ÉCOLE CENTRALE DE NANTES

2021 / 2022

Project Report for "PROJECT JPEG"

Presented by

Shutong JIN

On 23/04/2021

"Compression and Decompression for JPEG"

Evaluator: Professor Diana Mateus

1 Introduction

JPEG is a commonly used method for lossy compression for digital images, particularly for those images produced by digital photography[1]. The method depends on the fact that we don't see color quite as well as we do gray scale. It also deals with the fact that we don't see high-frequency changes in image intensity very well[2]. The main idea of this project is try to shrink the storage down as small as possible and extract as much as possible on the way out.

2 Process

A flow chart for this project is drawn below, as shown in figure 2.1

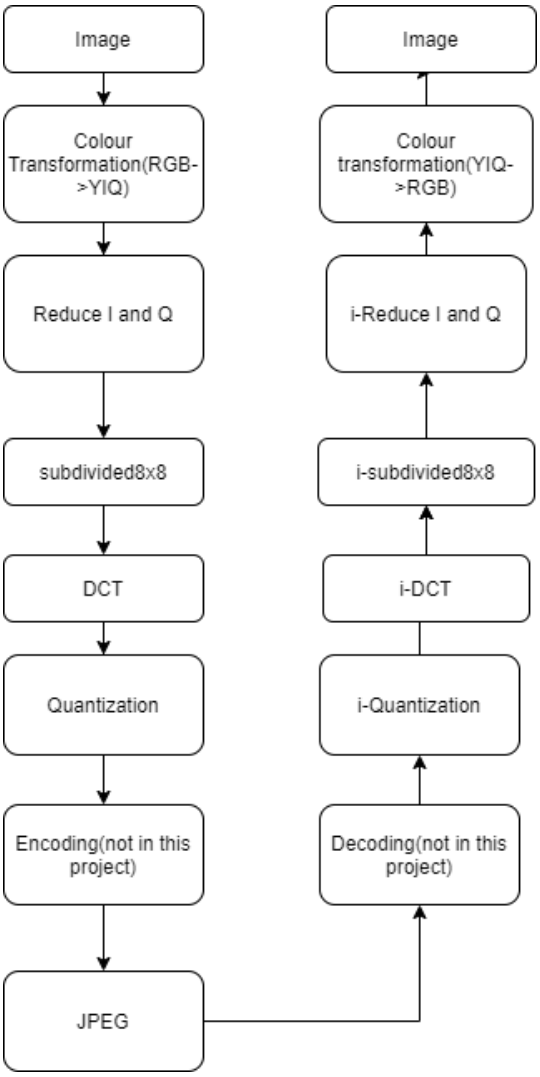

Figure 2.1: Process

3 Compression

3.1 RGB to YIQ

At the beginning, I resized the image to 128*128, because this is the multiple of 8, which facilitate the subdivision. Also, smaller image can decrease the running time. The original image is shown in figure 3.1 In this part, the image is transformed from RGB into YIQ. As in the YIQ form, the luminance(the first layer and) and chrominance(the second and third layer). And as for this transformation, I apply an existing formula[3], as shown below:

$$trans = \begin{bmatrix} 0.257 & 0.564 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \tag{3.1}$$

$$shift = \begin{bmatrix} 16 & 128 & 128 \end{bmatrix} \tag{3.2}$$

$$YIQ = trans * RGB + shift \tag{3.3}$$

Figure 3.1: original image

This is the image transformed into YIQ:

Figure 3.2: RGB2YIQ

3.2 Reduce I and Q

As I have mentioned in the introduction part, human eyes are less sensitive to chrominance, therefore, the size of I and Q channels are downsized. This can save quite a lot of space, without reducing the quality of the image distinctly. The result is shown in figure 3.3.

Figure 3.3: Reduce I and Q

3.3 Divide into 8x8

In order to be prepared for the coming DCT and quantization, which used arrays in 8 x 8, the image must be divided into 8 x 8 pieces, as shown in figure 3.4.
One thing needs to be explained is that I use a series of 3-d array to save the divided image, for example:

```
def i_subdivided8x8(out2):
    out3 = np.zeros((w, h, 3))
    a = np.zeros((8, 8, 3))
    k = 0
    for i in range(int(w/8)):
        for j in range(int(h/8)):
            a = out2[k]
            out3[i*8:(i+1)*8, j*8:(j+1)*8, :] = a
            k=k+1
    return out3
```

First of all, I defined a 3-d array, then use this array to assign the final output. And the reason why I am doing this is that I tried to use a 4-d array to save 64 3-d array at first, but failed, maybe it's because it's easier for human to understand 3-d. Then I tried out this way, by alining a series of 3-d array, I can save the image in the form I want. It's like a 1-d array, only the element is a 3-d array. And this idea of saving and reading the data is widely used in the whole project.

3.4 DCT

In this part, DCT is applied, because this can concentrate the information to the top left part of the matrix, and the rest is a small value near zero. Also, humans are less sensitive to the high

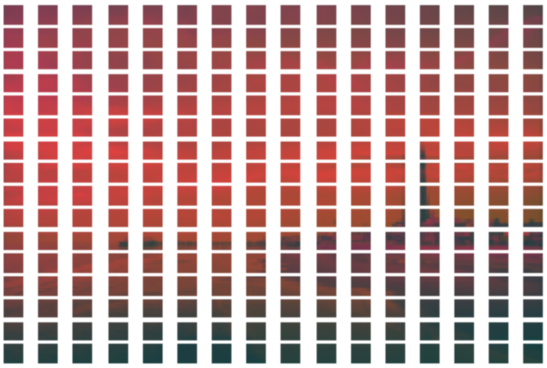

Figure 3.4: subdivision

frequencies information, in turn, this can save a lot of storage.
Another detail to note is that the data should be centralized before transformation, therefore, all of them should be subtracted by 128.

3.5 Quantization

In this part, two quantization table are used for different layers. And by doing this, the data can be more centralized. After being divided by these table, the data are transformed to integers compulsively. For now, most of the compression is finished. I will use the data of one layer of one patch to illustrate these:

```
[[ 1.01000000e+02  1.02250000e+02  1.04000000e+02  1.03500000e+02
   1.02250000e+02  1.01250000e+02  1.00750000e+02  1.00750000e+02]
 [-1.19999626e+01 -1.27843242e+01 -1.24903553e+01 -1.37718130e+01
  -1.37661300e+01 -1.42565227e+01 -1.58780368e+01 -1.75647285e+01]
 [-1.03596491e+00 -1.55394737e+00 -1.68924640e+00 -1.03596491e+00
  -7.09324174e-01 -5.60426911e-02 -5.17982457e-01 -5.60426911e-02]
 [-2.12024756e+00 -1.63553790e+00 -1.70451275e+00 -2.15450086e+00
  -2.47135709e+00 -2.05562228e+00 -1.44914673e+00 -2.23919126e+00]
 [-2.00000000e+00 -2.25000000e+00 -2.00000000e+00 -2.50000000e+00
  -2.75000000e+00 -2.75000000e+00 -2.75000000e+00 -2.25000000e+00]
 [-2.09100438e+00 -1.46645930e+00 -1.81321926e+00 -1.51254682e+00
  -2.57469348e+00 -2.29690836e+00 -1.59100190e+00 -9.67790244e-01]
 [-1.19447758e+00 -1.79171637e+00 -1.46507563e+00 -1.19447758e+00
  -1.05917856e+00 -7.88580507e-01 -5.97238791e-01 -7.88580507e-01]
 [-7.95705666e-01 -6.96827088e-01 -8.93250827e-01 -6.38353038e-01
   8.37372013e-02 -1.38079597e-02 -8.14493870e-02 -8.83015266e-01]]
```

Figure 3.5: After DCT

4 Decompression

The realization of decompression is just the inverse of compression. So the explanation for this part is relatively short.

```
[[33 34 20 11  7  6  6  6]
 [-3 -3 -2 -1  0 -1 -1 -1]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
```

Figure 3.6: After quantization

4.1 i-Quantization

I will use the same layer as an example. This is the data after i-quantization, as shown in figure 4.1.

```
[[ 99 102 100  99  91  90  90  90]
 [ -9 -12 -12 -11   0 -12 -12 -12]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]]
```

Figure 4.1: After i-quantization

4.2 i-DCT

I will use the same layer as an example. This is the data after i-DCT, as shown in figure 4.2

4.3 i-Divide into 8x8

In this part, the patches are put together, as shown in figure 4.3

4.4 i-Reduce I and Q

In this part, the reduced channels I and Q are recovered, as shown in figure 4.4

```
[[61.36851715 62.24897027 59.70952829 61.51338002 60.52440226 64.64485832
  64.48684997 63.10409674]
 [62.43561768 62.66242828 60.95950243 62.17066949 60.91233575 64.98968809
  64.91788717 63.62134139]
 [63.63037962 63.29093357 62.49217071 63.24971658 61.62914336 65.62685041
  65.71434008 64.57708487]
 [64.08833808 63.84722475 63.3916251  64.39466899 62.56569764 66.45934311
  66.75495595 65.82582392]
 [63.87003269 64.11114603 63.65119145 65.29575259 63.57941669 67.36042671
  67.88131045 67.17744932]
 [63.70307173 64.04251778 63.92572641 65.81578557 64.51597097 68.1929194
  68.92192631 68.42618836]
 [64.01406413 63.78725354 64.57462516 66.01106313 65.23277859 68.83008172
  69.71837922 69.38193184]
 [64.45624524 63.57579212 65.19967987 66.04343318 65.62071207 69.17491149
  70.14941643 69.89917649]]
```

Figure 4.2: After i-DCT

Figure 4.3: After i-Divide

Figure 4.4: YIQ

4.5 YIQ to RGB

In this part, the YIQ image is transformed back to RGB, as shown in figue 4.5.

5 Bonus

DCT is just a particular form of DFT. It's known that $X[k] = \sum_{n=0}^{N-1} x[n](cos(\frac{2\pi kn}{N}) - jsin(\frac{2\pi kn}{N}))$, and if the input signal is a real even function, like cosine, the transform is equal to $\sqrt{\frac{2}{N}} * \sum_{n=0}^{N-1} x'[n]cos(\frac{(n+\frac{1}{2}\pi k)}{N})$.
The 2d-DCT can be simplified to[4]:

Figure 4.5: RGB

- Obtain the matrix f(x,y).
- Calculate the coefficient function A.
- Transpose A and obtain A^T.
- Calculate DCT based on Ax[f(x,y)]xA^T.

The code to realize these:

```
def my_dct_2d(im):
    A = np.zeros((8,8))
    shape = im.shape[1]
    for i in range(8):
        for j in range(8):
            if(i == 0):
                x = np.sqrt(1/shape)
            else:
                x = np.sqrt(2/shape)
            A[i, j] = x*np.cos(np.pi*(j + 0.5)*i/shape)
    A_T = A.transpose()
    out = np.zeros((8,8,3))
    for i in range(3):
        Y1 = np.dot(A,im[:,:,i])
        out[:,:,i] = np.dot(Y1,A_T)
    return out
```

It turns out that the storage got even smaller, but it also means that some visible information are neglected. The outcome for DCT and quantization are shown in figure 5.1 and figure 5.2 respectively.

6 Analysis

The jpegCompress and jpeg Decompress function are called on the following images: before and after applying a gaussian filter, on images with high content, on images with low content,

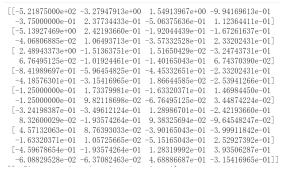

Figure 5.1: myDCT

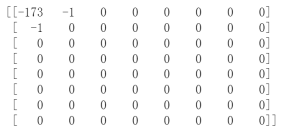

Figure 5.2: myDCT-quantization

the outcomes are shown in 6.1.

By comparing the original image and the image after a gaussian filter, it's obvious that the

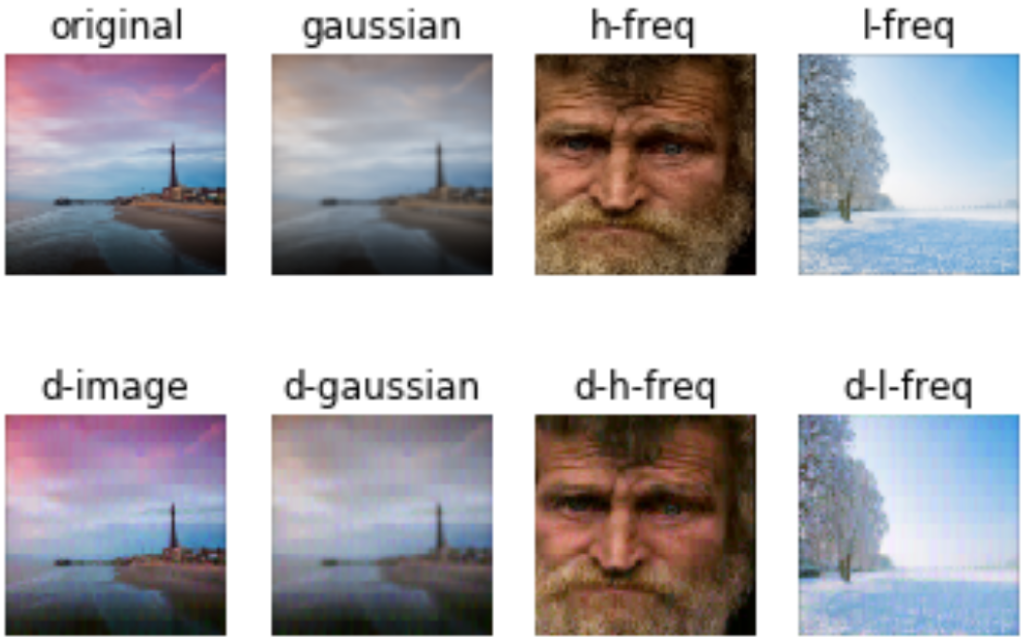

Figure 6.1: various kinds of images

gaussian image is vaguer than the original one, which is quite normal. And by observing those decompressed images, I found out that the information loss on gaussian image is less than on the original image. It's maybe because the origianl gaussian image's border is relatively dimmer. And we distinguish the difference mainly by borders, therefore, the gaussian image

has less difference between the original and the decompressed.
By comparing the high-frequency and the low-frequency, I find out that the compression has less damaging influence on low-frequency image, probably because the high-frequency composition are omitted to a large extent.
The frequency representation of the original image and the compressed image is shown in figure 6.2. By analysing this, the central frequency didn't change much, which implies that the image stay roughly the same.

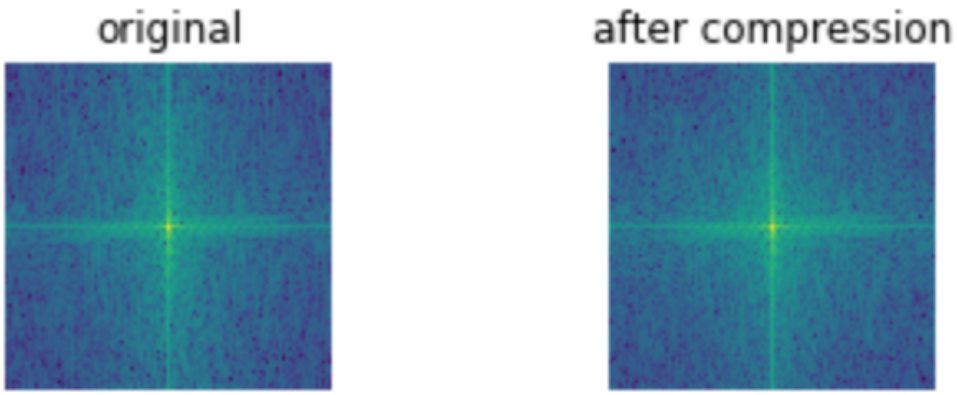

Figure 6.2: Frequency analysis

7 Conclusion

By doing this project, I have a thorough insight into JPEG and JFIF. Apart from that, my grammatical knowledge for python and basic imaging processing skills, like subdivision, fourier transform ..., are improved. This is really meaningful and thank you so much for bring me this chance of practical exercise.

References

[1] Wikimedia Foundation. Jpeg. https://en.wikipedia.org/wiki/JPEG. Lasted accessed April 21, 2021.

[2] Computerphile. Jpeg 'files' colour (jpeg pt1). https://www.youtube.com/watch?v=n_uNPbdenRs. Lasted accessed April 21, 2015.

[3] . Rgbycbcr. https://zhuanlan.zhihu.com/p/88933905. Lasted accessed October 28, 2019.

[4] konmu. dct. https://www.cnblogs.com/Konmu/p/12555802.html. Lasted accessed Mar 23, 2020.