Image Processing Assignment 3

Introduction

- Goal:
  - Experiment with the various components of a JPEG codec
  - Your implementation should include both an encoder and a decoder.
  - Evaluate your results both objectively and subjectively
  - Start your experiments with gray-scale images. If you are successful at compress and reconstruct gray-scale images, then move on to color images
  - Use at least 4 images from those supplied with the second assignment
- Environment: Python 3.8.8
- Code: https://github.com/maikurufeza/NCTU-Image-Processing-2021/blob/main/Assignment/Assignment3.ipynb

Implemented method

I. JPEG codec
   A. Block based transform coding: Discrete Cosine Transfer (DCT)
   B. Quantization of Discrete Cosine Transform (DCT) coefficients
   C. Run-length coding of the AC coefficients
   D. Chromatic subsampling
   E. Huffman coding
   F. SNR and RMSE

Organization

- Section 1 (Experiments): I show all the method I practice and its effect.
- Section 2 (Observation and Discussions):
  I show some my observation. And because the problem of over-segmentation, I try my best to improve the result and discussion how I improve.
- Section 3 (Code Analysis): This part shows the code of the method above.

※ **Note**:

Because all images from those supplied with the second assignment are jpg file. That is, all images in the second assignment have been compressed by JPEG. Therefore, I implement from all images in the second assignment in Section2. Then, I will use other un-compressed png image in Section3 to show the detailed difference between original images and decoded images.

Section 1: Experiments

I. gray-scale images

Method: (block size = 8)

Encode: gray image → DCT → Quantization → Huffman encode → JPEG

Decode: JPEG → Huffman decode → Deqantization → IDCT → gray image

Experiment:

Original image

Decoded image

Result:

Original size	JPEG size	SNR	RMSE	Compression ratio
640KB	206KB	1219.6702	3.6213	32.18%

II. color-scale images

Method: (block size = 8)

Encode: color image → DCT → Quantization → Huffman encode → JPEG

Decode: JPEG → Huffman decode → Deqantization → IDCT → color image

Experiment:

Original image

Decoded image

Result:

Original size	JPEG size	SNR	RMSE	Compression ratio
557KB	267KB	559.8229	7.6475	47.93%

III. With Run-length coding of the AC coefficients

Method: (block size = 8)

Encode: color image→DCT&Quantize→Run-length→Huffman encode→JPEG

Decode: JPEG → Huffman decode → De-Run-length → Deqantization → IDCT → color image

Experiment:

Original image

Decoded image

Result:

Original size	JPEG size	SNR	RMSE	Compression ratio
351KB	99KB	194.3404	18.5196	28.21%

IV. With Chromatic subsampling

Method: (block size = 8)

Encode: color image → color space conversion → Chromatic subsampling → DCT → Quantization → Run-length→Huffman encode→JPEG

Decode: JPEG → Huffman decode → De-Run-length → Deqantization → IDCT → Chromatic resizing → color space conversion → color image

Experiment:

Original image

Decoded image

Result:

Original size	JPEG size	SNR	RMSE	Compression ratio
279KB	40KB	804.5861	10.2409	14.34%

V. Summary

Run-length coding and Chromatic subsampling can get better compression ratio. Meanwhile, the decoded image doesn't differ from the original image a lot for Human eyes.

Because original images are jpg files, that is, the original images have been compressed. Therefore, I think the metrics of SNR and RMSE is not proper in this section.

Section 2: Observation and Discussion

1. Where is the difference between original images and decoded images?

Original image

Decoded image

They look the same! Where is the difference? Let's zoomed-in to see the detail

Original image

Decoded image

Then we can tell where the difference is. We can also evaluate the result objectively by using SNR and RMSE.

Original size	JPEG size	SNR	RMSE	Compression ratio
1386KB	109KB	1195.6235	6.5166	7.86%

We can see that SNR is big (bigger better), RMSE is small (smaller better). Meanwhile, Compression ratio (smaller better) is very small. It means we use less file size to store very similar image.

2. <u>What happen when we change the block size?</u>

Original image

block size = 8

block size = 32

block size = 128

Original image

block size = 8

block size = 32

block size = 128

Block size	Original size	JPEG size	SNR	RMSE	Compression ratio
8	1386KB	108KB	1790.8857	5.3246	7.79%
32	1386KB	104KB	1785.9753	5.3319	7.50%
64	1386KB	138KB	1497.1041	5.8236	9.95%

From zoom-in images, we can see that the bigger block size is, the more blurred is. From the table above, we can see that the bigger block size is, the more different image is (SNR get smaller, RMSE get bigger). But the compression ratio seems not affected by the block size.

3. <u>What happen when we change the Chromatic subsampling size?</u>

Original image

subsampling 1/2

subsampling 1/8

subsampling 1/32

Original image

subsampling 1/2

subsampling 1/8

subsampling 1/32

Subsample	Original size	JPEG size	SNR	RMSE	Compression ratio
1/2	1386KB	104KB	1785.9753	5.3319	7.50%
1/8	1386KB	94KB	751.9026	8.2175	6.78%
1/32	1386KB	92KB	257.6314	14.0386	6.63%

From the result of the image, we can see that the color is slightly different.

From zoom-in image, we can see that the bigger subsampling factor is, the more blurred is. And we can see that when subsampling 1/32, the color is different from original image.

From the table above, we can see that the bigger subsampling factor is, the more different image is (SNR get smaller, RMSE get bigger). We can also see that the bigger subsampling factor is, the lower compression ratio is.

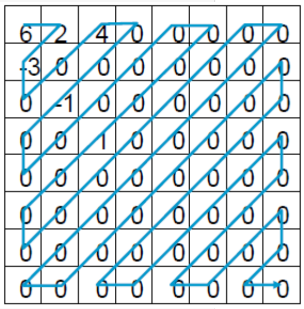
Code Analysis

JPEG encoder

```
class JPEGEncoder():
    def __init__(self, block_size, chromatic_subsampling = True, chromatic_factor = 2, run_length = True):
        self.block_size = block_size
        self.quantization_tables = self.load_quantization_table(block_size)
        self.zigzag_points = self.get_zigzag_points(block_size)
        self.chromatic_subsampling = chromatic_subsampling
        self.chromatic_factor = chromatic_factor
        self.run_length = run_length
```

```
    def load_quantization_table(self, block_size):
        q_lum = np.array([[16,11,10,16,24 ,40 ,51 ,61 ],
                          [12,12,14,19,26 ,58 ,60 ,55 ],
                          [14,13,16,24,40 ,57 ,69 ,56 ],
                          [14,17,22,29,51 ,87 ,80 ,62 ],
                          [18,22,37,56,68 ,109,103,77 ],
                          [24,35,55,64,81 ,104,113,92 ],
                          [49,64,78,87,103,121,120,101],
                          [72,92,95,98,112,100,103,99 ]])
        q_chrom = np.array([[17,18,24,47,99,99,99,99],
                          [18,21,26,66,99,99,99,99],
                          [24,26,56,99,99,99,99,99],
                          [47,66,99,99,99,99,99,99],
                          [99,99,99,99,99,99,99,99],
                          [99,99,99,99,99,99,99,99],
                          [99,99,99,99,99,99,99,99],
                          [99,99,99,99,99,99,99,99]])
        q_lum = cv2.resize(q_lum.astype('uint8'), dsize=(block_size,block_size), interpolation=cv2.INTER_CUBIC)
        q_chrom = cv2.resize(q_chrom.astype('uint8'), dsize=(block_size,block_size), interpolation=cv2.INTER_CUBIC)
        return {'lum': q_lum, 'chrom': q_chrom}
```

```
    def get_zigzag_points(self, block_size):
        x, y = 0, 0
        output = [(x,y)]
        move_up = True
        for i in range(block_size * block_size - 1):
            if move_up:
                if 0 <= x-1 < block_size and 0 <= y+1 < block_size: # can move up right?
                    x, y = x-1, y+1 # move up right
                else:
                    move_up = False
                    if 0 <= y+1 < block_size: # can move right?
                        y = y + 1 # move right
                    else:
                        x = x + 1 # move down
            else:
                if 0 <= x+1 < block_size and 0 <= y-1 < block_size: # can move down left?
                    x, y = x+1, y-1 # move down left
                else:
                    move_up = True
                    if 0 <= x+1 < block_size: # can move down?
                        x = x + 1 # move down
                    else:
                        y = y + 1 # move right
            output.append((x,y))
        return output
```




```
    def dct_2d(self, image):
        return fftpack.dct(fftpack.dct(image.T, norm='ortho').T, norm='ortho')

    def quantize(self, block, component):
        q = self.quantization_tables[component]
        return (block / q).round().astype(np.int32) #(?) round?
```

```
    def run_length_encode(self, arr):
        last_nonzero = -1
        for i, elem in enumerate(arr):
            if elem != 0:
                last_nonzero = i

        run_len = 0
        output = []
        for i, elem in enumerate(arr):
            if i > last_nonzero:
                output.append((0, 0))
                break
            elif elem == 0:
                run_len += 1
            else:
                output.append((run_len, elem))
                run_len = 0

        if last_nonzero == len(arr)-1:
            output.append((0, 0))
        return output
```

```python
    def quant_and_dct2D(self, image, component):
        # image padding
        pad_hight = (self.block_size - image.shape[0]%self.block_size)%self.block_size
        pad_width = (self.block_size - image.shape[1]%self.block_size)%self.block_size
        image = np.pad(image, ((0, pad_hight), (0, pad_width)), 'constant', constant_values=0)
        # parameter setting
        image_hight, image_width= image.shape[0], image.shape[1]
        blocks_count = math.ceil(image_hight/self.block_size) * math.ceil(image_width/self.block_size)
        block_area = self.block_size * self.block_size
        dc = np.empty((blocks_count), dtype=np.int32)
        ac = np.empty((blocks_count, block_area-1), dtype=np.int32)

        block_index = -1
        ac_run_len = []
        for i in range(0, image_hight, self.block_size):
            for j in range(0, image_width, self.block_size):
                block_index += 1
                block = image[i:i+self.block_size, j:j+self.block_size] # split block_sizexblock_size block

                dct_matrix = self.dct_2d(block) # get DCT of the block
                quant_matrix = self.quantize(dct_matrix, component) # quantize
                zz = np.array([quant_matrix[p] for p in self.zigzag_points]) # zigzag index
                dc[block_index] = zz[0]
                ac[block_index, :] = zz[1:]
                ac_run_len.append(self.run_length_encode(zz[1:]))
        return dc, ac, ac_run_len
```

Padding

DCT

Quantilization

Run-length

```python
    def fit(self, image):
        # image preprocessing
        image_is_nparray = (type(image).__module__ == np.__name__)
        if self.chromatic_subsampling: # color space conversion
            if image_is_nparray: image = Image.fromarray(image.astype('uint8'))
            image = image.convert('YCbCr')
            if image_is_nparray: image = np.array(image)
        if not image_is_nparray: image = np.array(image, dtype='int32')  # any type --> ndarray
        image_is_gray = True if len(image.shape) == 2 else False # dim ? --> dim 3
        image = image.reshape((image.shape[0], image.shape[1], 1 if image_is_gray else image.shape[2]))
        image = image - 128   # [0, 255] --> [-128, 127]
        image_hight, image_width, image_channel= image.shape
        self.image_shape = image.shape
        image = [image[:,:,c] for c in range(image_channel)]

        if self.chromatic_subsampling:
            for c in range(1,image_channel):
                chrom_image = image[c] + 128
                chrom_image = cv2.resize(chrom_image.astype('uint8'), \
                                         dsize=(image_width//self.chromatic_factor,image_hight//self.chromatic_factor), interpola
                chrom_image = chrom_image.astype('int32')-128
                image[c] = chrom_image
```

Color space conversion

Chromatic subsampling

```python
        self.dc = []
        self.ac = []
        self.dc_huffman_table = []
        self.ac_huffman_table = []
        huffman = Huffman()
        for c in range(image_channel): # quantization and DCT
            dct_dc, dct_ac, dct_ac_run_len = self.quant_and_dct2D(image[c], 'lum' if c == 0 else 'chrom')
            huffman_dc, table_dc = huffman.encode(dct_dc)
            self.dc.append(huffman_dc)
            self.dc_huffman_table.append(table_dc)

            if self.run_length:
                huffman_ac, table_ac = huffman.encode(flatten(flatten(dct_ac_run_len)))
            else:
                huffman_ac, table_ac = huffman.encode(flatten(dct_ac))
            self.ac.append(huffman_ac)
            self.ac_huffman_table.append(table_ac)

        return len(dct_dc)
```

Huffman coding

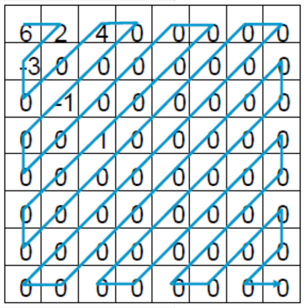
```python
    def write_to_file(self, filepath):
        return JPGfile(img_channel = self.image_shape[2],
                       is_run_len = self.run_length,
                       is_chromatic_subsampling = self.chromatic_subsampling,
                       chromatic_factor = self.chromatic_factor,
                       block_size = self.block_size,
                       img_hight=self.image_shape[0],
                       img_width=self.image_shape[1],
                       dc = self.dc,
                       dc_table = self.dc_huffman_table,
                       ac = self.ac,
                       ac_table = self.ac_huffman_table)
```

JPEG decoder

```python
class JPEGDecoder():
    def __init__(self, jpg_image):
        self.jpg_image = jpg_image
        self.read_jpg(jpg_image)
        self.quantization_tables = self.load_quantization_table(self.block_size)
        self.zigzag_points = self.get_zigzag_points(self.block_size)
```

```python
def load_quantization_table(self, block_size):
    q_lum = np.array([[16,11,10,16,24 ,40 ,51 ,61 ],
                      [12,12,14,19,26 ,58 ,60 ,55 ],
                      [14,13,16,24,40 ,57 ,69 ,56 ],
                      [14,17,22,29,51 ,87 ,80 ,62 ],
                      [18,22,37,56,68 ,109,103,77 ],
                      [24,35,55,64,81 ,104,113,92 ],
                      [49,64,78,87,103,121,120,101],
                      [72,92,95,98,112,100,103,99 ]])
    q_chrom = np.array([[17,18,24,47,99,99,99,99],
                        [18,21,26,66,99,99,99,99],
                        [24,26,56,99,99,99,99,99],
                        [47,66,99,99,99,99,99,99],
                        [99,99,99,99,99,99,99,99],
                        [99,99,99,99,99,99,99,99],
                        [99,99,99,99,99,99,99,99],
                        [99,99,99,99,99,99,99,99]])
    q_lum = cv2.resize(q_lum.astype('uint8'), dsize=(block_size,block_size), interpolation=cv2.INTER_CUBIC)
    q_chrom = cv2.resize(q_chrom.astype('uint8'), dsize=(block_size,block_size), interpolation=cv2.INTER_CUBIC)
    return {'lum': q_lum, 'chrom': q_chrom}
```

```python
def get_zigzag_points(self, block_size):
    x, y = 0, 0
    output = [(x,y)]
    move_up = True
    for i in range(block_size * block_size - 1):
        if move_up:
            if 0 <= x-1 < block_size and 0 <= y+1 < block_size: # can move up right?
                x, y = x-1, y+1 # move up right
            else:
                move_up = False
                if 0 <= y+1 < block_size: # can move right?
                    y = y + 1 # move right
                else:
                    x = x + 1 # move down
        else:
            if 0 <= x+1 < block_size and 0 <= y-1 < block_size: # can move down left?
                x, y = x+1, y-1 # move down left
            else:
                move_up = True
                if 0 <= x+1 < block_size: # can move down?
                    x = x + 1 # move down
                else:
                    y = y + 1 # move right
        output.append((x,y))
    return output
```



```python
def zigzag_to_block(self, zigzag):
    block = np.empty((self.block_size, self.block_size), np.int32)
    for i, point in enumerate(self.zigzag_points):
        block[point] = zigzag[i]
    return block

def dequantize(self, block, component):
    q = self.quantization_tables[component]
    return block * q

def idct_2d(self, image):
    return fftpack.idct(fftpack.idct(image.T, norm='ortho').T, norm='ortho')
```

```python
def dequant_and_idct2D(self, dc, ac, component):
    if self.is_chromatic_subsampling and component == 'chrom':
        img_h = self.image_hight//self.chromatic_factor
        img_w = self.image_width//self.chromatic_factor
    else:
        img_h = self.image_hight
        img_w = self.image_width

    blocks_count_height = math.ceil(img_h/self.block_size)
    blocks_count_width = math.ceil(img_w/self.block_size)
    blocks_count = blocks_count_height * blocks_count_width
    block_area = self.block_size * self.block_size

    if not self.is_run_len:
        ac = [ac[i*(block_area-1):i*(block_area-1)+(block_area-1)] for i in range(blocks_count)]
    dct_matrix = np.hstack([np.array(dc).reshape((blocks_count),1), np.array(ac)])

    npmat = np.empty((blocks_count_height*self.block_size, blocks_count_width*self.block_size), dtype=np.int32)
    for b in range(blocks_count):
        block = self.zigzag_to_block(dct_matrix[b])
        block = self.dequantize(block, component) # dequantize
        block = self.idct_2d(block)  # idct

        block_i = (b//blocks_count_width)*self.block_size
        block_j = (b%blocks_count_width)*self.block_size
        npmat[block_i:block_i+self.block_size, block_j:block_j+self.block_size] = block
```

dequantilization

IDCT

```python
    # depadding
    pad_hight = (self.block_size - img_h%self.block_size)%self.block_size
    pad_width = (self.block_size - img_w%self.block_size)%self.block_size
    output_hight = blocks_count_height*self.block_size - pad_hight
    output_width = blocks_count_width*self.block_size - pad_width
    output = npmat[:output_hight, :output_width]
    return output
```

Inverse padding

```python
    def run_length_decode(self, inputs):
        output = []
        ac = []
        test_count = 0
        for i in range(0,len(inputs),2):
            run_len, elem = inputs[i], inputs[i+1]
            if run_len == 0 and elem == 0:
                for pad in range(self.block_size*self.block_size-1-len(ac)):
                    ac.append(0)
                output.append(ac)
                ac = []
                test_count+=1
                continue
            for z in range(run_len):
                ac.append(0)
            ac.append(elem)
        return output
```

```python
    def fit(self):
        huffman = Huffman()
        output = []
        for c in range(self.image_channel):
            dc_dehuffman = huffman.decode(self.dc[c], self.dc_table[c])
            ac_dehuffman = huffman.decode(self.ac[c], self.ac_table[c])
            if self.is_run_len:
                ac_dehuffman = self.run_length_decode(ac_dehuffman)

            image = self.dequant_and_idct2D(dc_dehuffman, ac_dehuffman, 'lum' if c == 0 else 'chrom')
            output.append(image)

        if self.is_chromatic_subsampling:
            for c in range(1,self.image_channel):
                chrom_image = output[c] + 128
                chrom_image = cv2.resize(chrom_image.astype('uint8'), dsize=(self.image_width, self.image_hight), interpolation=c
                chrom_image = chrom_image.astype('int32')-128
                output[c] = chrom_image

        if self.image_channel==1:
            output = output[0]
        else:
            output = np.dstack(output)
```

Huffman decode

Run-length decode

```python
        output += 128
        output = np.clip(output, 0, 255)

        # image space deconversion
        if self.is_chromatic_subsampling:
            output = Image.fromarray(output.astype('uint8'), mode="YCbCr")
            output = output.convert('RGB')
            output = np.array(output)
        return output.astype('uint8')
```

Metrics

```python
def MSE(oriImage, recImage):
    err = np.sum((oriImage.astype("float") - recImage.astype("float")) ** 2)
    err /= float(oriImage.shape[0] * oriImage.shape[1])
    return err


def RMSE(oriImage, recImage):
    return math.sqrt(MSE(oriImage, recImage))


def SNR(oriImage, recImage):
    denominator = np.sum((oriImage.astype("float") - recImage.astype("float")) ** 2)
    numerator = np.sum((oriImage.astype("float")) ** 2)
    return numerator/denominator
```