Subject : EIE2108 Lab 3 Report

Student ID: 19069748D

Name: Kwok Kevin

Background of Lab 3:

Block Vector Quantization Coding (BVQC) is a type of lossy image compression technique for greyscale images. It divides the original images into blocks to derive a codebook for each block and then vector-quantizes the sub-blocks of each block based on the codebook associated with the block.

- Its encoding process is as follows:

  Step1: Get the input image of size $L_1 \times L_2$ pixels. Normalize its intensity values such that each pixel value is bounded in [0,255].

  Step2: Partition the image into blocks. Each block is of size $d \times d$ pixels.

  Step3: Visit the blocks in the image in a raster scanning manner. For each block, do the following steps:

  Step 3a: Compute its mean μ and its standard deviation σ with

  $$\mu = \frac{1}{N}\sum_i x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{N}\sum_i (\mu - x_i)^2}$$

  where x_i is the intensity value of the i[th] pixel in the block and $N = d \times d$ is the total number of pixels in the block.

  Step 3b: Construct a codebook based on μ and σ as follows:

index	0	1	2	3
codeword	$c_0 = \begin{bmatrix} g_0 & g_0 \\ g_1 & g_1 \end{bmatrix}$	$c_1 = \begin{bmatrix} g_1 & g_1 \\ g_0 & g_0 \end{bmatrix}$	$c_2 = \begin{bmatrix} g_0 & g_1 \\ g_0 & g_1 \end{bmatrix}$	$c_3 = \begin{bmatrix} g_1 & g_0 \\ g_1 & g_0 \end{bmatrix}$

  where $g_0 = \max(0, \mu - \sigma)$ and $g_1 = \min(255, \mu + \sigma)$

  Step 3d: Divide the block into subblocks of size 2x2 and approximate each of them as the closest codeword based on their distances from the subblock. The distance is measured in terms of $J = \sum_{m=0}^{1}\sum_{n=0}^{1}(x(m,n) - c_i(m,n))^2$, where $x(m,n)$ is the $(m,n)^{th}$ value of the subblock and $c_i(m,n)$ is the $(m,n)^{th}$ element of codeword c_i. When there are more than one closest codewords, the one with the minimum index value is picked. This step makes use of the vector quantization technique to further compress the image.

  An example of vector quantization is given as follows. The distances of subblock $\begin{bmatrix} 0 & 1 \\ 3 & 4 \end{bmatrix}$ from codewords $c_0 = \begin{bmatrix} 0 & 0 \\ 4 & 4 \end{bmatrix}$, $c_1 = \begin{bmatrix} 4 & 4 \\ 0 & 0 \end{bmatrix}$, $c_2 = \begin{bmatrix} 0 & 4 \\ 0 & 4 \end{bmatrix}$ and $c_3 = \begin{bmatrix} 4 & 0 \\ 4 & 0 \end{bmatrix}$ are 2, 50, 18 and 34 respectively. The subblock is then approximated as codeword c_0 and index 0 is recorded for the subblock.

  Step 3e: Store (i) quantized μ and σ in 8 bit unsigned integer format and (ii) the indices of the subblocks.

- Its decoding process is as follows:

  Step 1: For each block of size $d \times d$, do the following steps:

  Step 1a: Compute parameters g_0 and g_1 based on the received μ and σ. Use their rounded values to reconstruct a codebook as in the step 3b performed in the encoder.

  Step 1b: Approximate individual subblocks of the block as the codewords associated with their corresponding indices.

  Step 2: Combine the blocks to form the final reconstructed image.

- An example of handling a block for the case when d=4 is given as follows:

  The original block is $\begin{bmatrix} 245 & 239 & 249 & 239 \\ 245 & 245 & 239 & 235 \\ 245 & 245 & 245 & 245 \\ 245 & 235 & 235 & 239 \end{bmatrix}$. Its mean and standard deviation is 241.875 and 4.357 respectively. Accordingly, we have g_0= 237.52 and g_1=246.23.

  The codebook for the block will have codewords $c_0 = \begin{bmatrix} 237.52 & 237.52 \\ 246.23 & 246.23 \end{bmatrix}$, $c_1 = \begin{bmatrix} 246.23 & 246.23 \\ 237.52 & 237.52 \end{bmatrix}$, $c_2 = \begin{bmatrix} 237.52 & 246.23 \\ 237.52 & 246.23 \end{bmatrix}$ and $c_3 = \begin{bmatrix} 246.23 & 237.52 \\ 246.23 & 237.52 \end{bmatrix}$. The distances of subblock $\begin{bmatrix} 245 & 239 \\ 245 & 245 \end{bmatrix}$ from c_0, c_1, c_2 and c_3 are 61.215, 165.785, 165.785 and 61.215 respectively, so index 0 is recorded for this subblock. As for other subblocks, their indices are all found to be 1.

  At the decoding side, based on the received block mean and block standard deviation, parameters g_0 and g_1 are computed to reconstruct the codebook. The 4 codewords of the reconstructed codebook should be $c_0 = \begin{bmatrix} 238 & 238 \\ 246 & 246 \end{bmatrix}$, $c_1 = \begin{bmatrix} 246 & 246 \\ 238 & 238 \end{bmatrix}$, $c_2 = \begin{bmatrix} 238 & 246 \\ 238 & 246 \end{bmatrix}$ and $c_3 = \begin{bmatrix} 246 & 238 \\ 246 & 238 \end{bmatrix}$. Since the indices of the 4 subblocks are 0, 1, 1 and 1, the block is approximated to be $\begin{bmatrix} g_0 & g_0 & g_1 & g_1 \\ g_1 & g_1 & g_0 & g_0 \\ g_1 & g_1 & g_1 & g_1 \\ g_0 & g_0 & g_0 & g_0 \end{bmatrix} \approx \begin{bmatrix} 238 & 238 & 246 & 246 \\ 246 & 246 & 238 & 238 \\ 246 & 246 & 246 & 246 \\ 238 & 238 & 238 & 238 \end{bmatrix}$.

Mission of the task:

Build up two function BVQCencode and BVQCdecode for making data form from png to other data type

The content of BVQCencode:

- *in_image_filename* (in string) is the file name of the input image to be encoded.
- *out_encoding_result_filename* (in string) is the file name of the encoding output.

- *d* (in int) defines the block size that is determined as *d*×*d*. Its default value is 4 and it must be a power of 2.
- The function should return a data structure that carries the information of the encoding output. The data structure is a dictionary containing 3 elements:
  - M : a 2D numpy array of size $\left(\frac{L_1}{d}\right) \times \left(\frac{L_2}{d}\right)$, each element of which provides the mean of a specific block.
  - Sd : a 2D numpy array of size $\left(\frac{L_1}{d}\right) \times \left(\frac{L_2}{d}\right)$, each element of which provides the standard deviation of a specific block.
  - Idx : a 2D numpy array of size $\left(\frac{L_1}{2}\right) \times \left(\frac{L_2}{2}\right)$, each element of which provides the index of a specific subblock.

The content of BVQCdecode:

- *in_encoding_result_filename* (in string) is the file name of the file that contains the encoding result of BVQC. It should be an output of function ***BVQCencode***().
- *out_reconstructed_image_filename* (in string) is the image file name of the reconstructed image.
- The function should return the reconstructed grayscale image (in numpy array)

Mission of the task: (cont'd)

Function ***BVQCdecode***() should be able to read a specified file to get a data structure that carries the encoding result of an image in the format shown in Figure 1, decode it, display the reconstructed image and save the reconstructed image as a conventional image file.

Your main program should be able to do the following:

1. Prompt a user to input the file name of an image file through the command line. Hint should be provided to help the user to input an existing file that is in proper file format. It should be able to handle exceptions such as the case that the specified file does not exist.
2. Prompt the user to input parameter *d* through the command line. Hint should be provided to help the user to input a valid block size (must be an integer power of 2). When a null string is input, the default block size (=4) is used.
3. Do encoding with your encoding function by calling function ***BVQCencode***().
4. Do decoding with your decoding function by calling function ***BVQCdecode***().
5. Evaluate the quality of the reconstructed image in terms of mean square error (mse) and peak-to-peak signal-to-noise ratio (PPSNR). The mse and the PPSNR of a reconstructed image are defined as

$$mse = \frac{1}{L_1 L_2} \sum_{m=1}^{L_1} \sum_{n=1}^{L_2} (X(m,n) - \hat{X}(m,n))^2$$

where $X(m,n)$ and $\hat{X}(m,n)$ are, respectively, the $(m,n)^{th}$ pixel values of the original image X and the reconstructed image $\hat{X}$, and

$$PPSNR = 10log_{10}\left(\frac{255^2}{mse}\right).$$

6. Handle exceptions such as the case that the specified file does not exist.

Analysis and implementation of Codes:

Before analysis the code, I would like to discuss BVQCencode and BVQCdecode these two function.

For function BVQCencode

It could make png file to a encoding result((i.e. the index planes and the quantized * and + for each block) in a file. The structure of the file format follows the one shown in Figure 1. Table 2 defines the structure of the file header. The function returns a data structure that carries the encoding result.

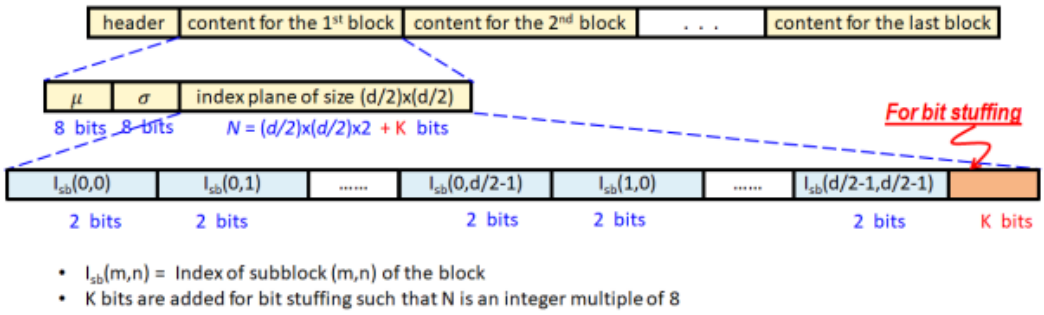


Figure 1 The structure of the file format

Byte(s)	No. of bits	Information
1	8	It specifies the header length in terms of number of bytes
2	8	It specifies the value of d, where $d \times d$ is the block size
3-4	16	It specifies the number of columns of blocks in the image
5-6	16	It specifies the number of rows of blocks in the image

Table 2 structure of the header

For function BVQCdecode

This is a function to do the opposite thing of BVQCencode. It could read the specified file to take a data structure that carries the encoding result of an image in the format shown in Figure 1, then decoding and display the reconstructed image and save the reconstructed image as a conventional image file.

Analysis and implementation of Codes: (cont'd)

```
def BVQCencode(in_image_filename, out_encoding_result_filename, d):

    in_image_filename = mpimg.imread(in_image_filename)
    intensity = np.array(in_image_filename) * 255

    # Define general variable
    halved_d = int(d/2) # Length of Subblock
    block_row = int(intensity.shape[0] / d)
    block_column = int(intensity.shape[1] / d)
    byte = int(d * d / 16)
    if(d == 2): # Bit stuffing
       byte = 1

    # Define the Dictionary Items
    dict_M = np.zeros([block_row, block_column], dtype = 'uint8')
    dict_Sd = np.zeros([block_row, block_column], dtype = 'uint8')
    dict_Idx = np.zeros([block_row * block_column, byte, 4], dtype = 'uint8')

    # Create content of each block
    for i in range(0,intensity.shape[0], d): # Column: Intensity
        for j in range(0,intensity.shape[1], d): # Row: Intensity
            # Define general variable
            block = intensity[i:i+d, j:j+d] # Get the block using d as interval
            mean = np.mean(block)
            std = np.std(block) # Standard Deviation

            # Define Codebook
            g0 = max(0, mean - std)
            g1 = min(255, mean + std)
            codebook = np.array(([[g0, g0],[g1, g1]], [[g1, g1],[g0, g0]], # c[0], c[1]
                                 [[g0, g1],[g0, g1]], [[g1, g0],[g1, g0]])) # c[2], c[3]

            # Define Subblock (2x2)
            subblock = np.zeros([halved_d,halved_d,2,2])
            for f in range(halved_d): # Column: block
                for g in range(halved_d): # Row: block
                    subblock[f][g] = block[f*2:f*2+2, g*2:g*2+2]

            # Find index by shortest diseance
            bit = 0
            byte_carry= 0
            for a in range(halved_d): # Column: subblock
                for b in range(halved_d): # Row: subblock
                    distance = np.array([]) # Empty the distance array
                    for c in range(4):
                        distance = np.append(distance, np.sum((subblock[a][b] - codebook[c]) ** 2))
                    dict_Idx[int(i/d * block_column + j/d), byte_carry, bit] = np.binary_repr(np.argmin(
                    bit += 1
                    if(bit == 4): # 4 number(4 x 2bits) to 1 byte --> Carrying system
                        byte_carry += 1
                        bit = 0

            # Record the Dictionary (Mean & SD)
            dict_M[int(i/d)][int(j/d)] = np.uint8(mean + 0.5)
            dict_Sd[int(i/d)][int(j/d)] = np.uint8(std + 0.5)
```

This is the beginning part of encoding .

```
def BVQCdecode(in_encoding_result_filename, out_reconstructed_image_filename):

    # Read the general variable
    file = open(in_encoding_result_filename, "rb")
    d = file.read(2)[1] # Size of block
    halved_d = int(d/2)
    byte = int(d * d / 16)
    if(d == 2): # Bit stuffing
        byte = 1
    # Amount of block in row
    block_row_hex_fh = np.base_repr(file.read(1)[0], base = 16) # Read the first half
    block_row_hex_sh = np.base_repr(file.read(1)[0], base = 16) # Read the second half
    block_row = int(block_row_hex_fh + block_row_hex_sh, 16) # Combine the first and se
    # Amount of block in column
    block_column_hex_fh = np.base_repr(file.read(1)[0], base = 16) # Read the first hal
    block_column_hex_sh = np.base_repr(file.read(1)[0], base = 16) # Read the second ha
    block_column = int(block_column_hex_fh + block_column_hex_sh, 16) # Combine the fi

    # Define the whole picture
    intensity = np.zeros([d * block_row, d* block_column])

    # Decode content of each block
    for i in range(0, d * block_row, d): # Column: Intensity
        for j in range(0, d * block_column, d): # Row: Intensity

            # Read the Mean and SD of each block
            mean = file.read(1)[0]
            std = file.read(1)[0]

            # Re-generate the codebook
            g0 = max(0, mean - std)
            g1 = min(255, mean + std)
            codebook = np.array(([[g0, g0],[g1, g1]], [[g1, g1],[g0, g0]],
                                 [[g0, g1],[g0, g1]], [[g1, g0],[g1, g0]]))

            # Reconstruct block by index
            block = np.zeros([halved_d,halved_d,2,2])
            for m in range(byte): # Read 1/4/16/... bit accroding to byte
                Idx = file.read(1)[0]
                Idx = '{0:08b}'.format(Idx) # Read the number in BINARY
                Idx_array = np.zeros([4])
                # Get the binary number from index
                bit = 0
                for n in range(4): # 8 bit number([0]00/ [1]00/ [2]00/ [3]00)
                    Idx_array[n] = int(Idx[bit:bit+2],2)
                    bit += 2 # [0:2] --> [2:4] --> [4:6] --> [6:8]
                # Reconstruct the subblock by index and codebook
                index_no = 0
                for o in range(halved_d): # Column: subblock
                    for p in range(halved_d): # Row: subblock
                        block[o][p] = codebook[int(Idx_array[index_no])]
                        index_no += 1
                        if(index_no == 4): # Reset the index number when count to 4
                            index_no = 0

            # Save the block into intensity
            for y in range(halved_d): # Column: Block
                for z in range(halved_d): # Row: Block
                    intensity[i+y*2:i+y*2+2, j+z*2:j+z*2+2] = block[y,z]
    file.close()
```

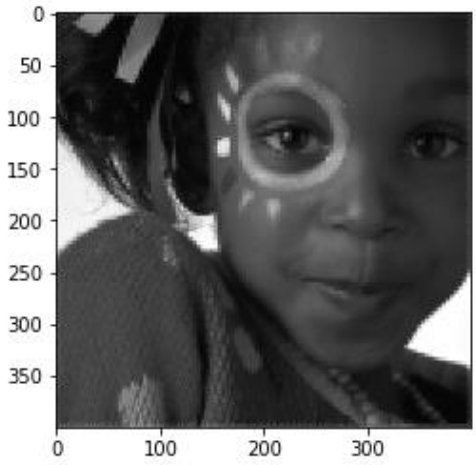
This is the beginning part of the decoding

This is a prototype to test the two functions working with the “myTimg.png”

```
fname_in = 'myTimg.png'
fname_out= 'myTimg_encoded.out'
d1=4
result = BVQCencode(fname_in,fname_out,d1)

fname_in = 'myTimg_encoded.out'
fname_out= 'myTimg_remake.png'
y = BVQCdecode(fname_in,fname_out)
print(fname_in)
print(fname_out)
```

Output :

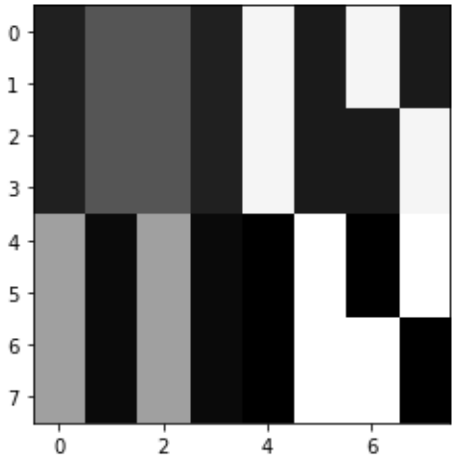


This is another prototype to test the two functions working with the 64bits “SImg8x8.png”

```
fname_in = 'SImg8x8.png'
fname_out= 'SImg8x8_encoded.out'
result = BVQCencode(fname_in,fname_out,d1)

fname_in = 'SImg8x8_encoded.out'
fname_out= 'SImg8x8_remake.png'
y = BVQCdecode(fname_in,fname_out)
print(fname_in)
print(fname_out)
"""
```

Output:



```
while True:
  try:
    img_in = input("Please input the file name ")
    open(img_in)
    break
  except:
    print("Error404: no that file")


while True:
  try:
    d = int(input("Please input random no "))
    break
  except:
    print("Error404: no that file")


replace = img_in.split(".")
bvqc_out = img_in.replace(replace[1], "bvqc")
img_out = replace[0] + '-bvqc-R.png'


BVQCencode(img_in, bvqc_out, d)
BVQCdecode(bvqc_out, img_out)
```

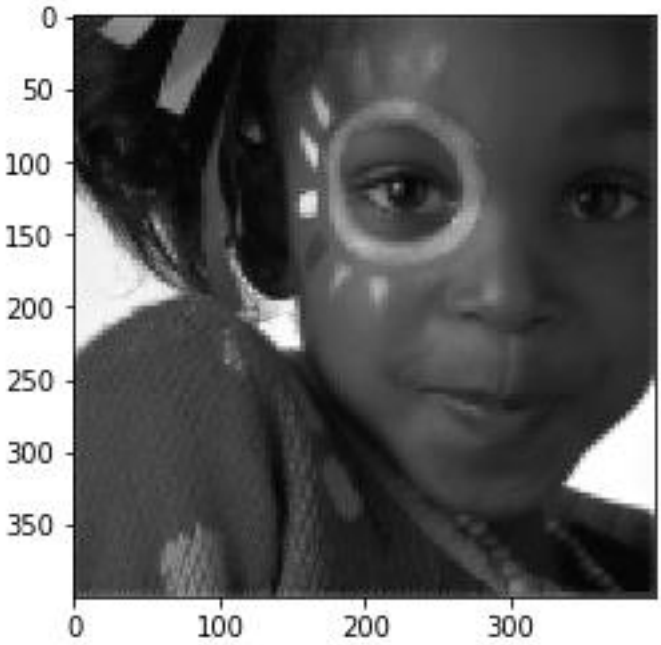
This is the part with asking user to input the file/photo name and the number

User working stage

```
Please input the file name myTimg.png

Please input random no 4
```

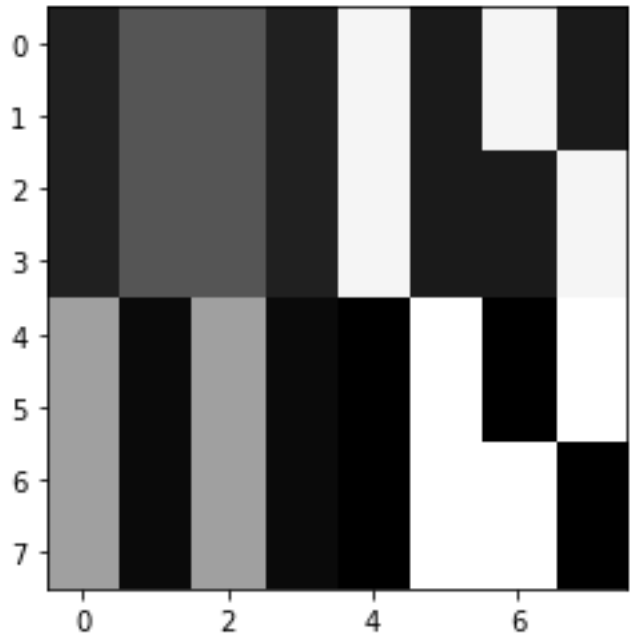
Output of the photot of myTimg.png




```
Please input the file name 4
Error404: no that file
```

message of wrong input name

Output of the photot of SImg8x8.png




```
Please input the file name SImg8x8.png

Please input random no 4
```