# JPEG Compression

KURUPPU K.D.R.J
200334F
31/10/2023

The Joint Photographic Experts Group (JPEG) compression technique is a popular way to reduce digital image file sizes without losing image quality.

The key steps of JPEG compression for a 24bpp image.

1. Color space conversion
2. Discrete Cosine Transform
3. Quantization
4. Zig Zac scanning
5. Run Length and Huffman coding

1.Color space conversion

First step of the JPEG compression is color space conversion. In this process the 24bpp RGB image is converted into YCrCb format. Here it takes RGB values for every single pixel and calculate the new values luminance, blue chrominance and red chrominance using the following formulars.

$Y \ = 0.299R + 0.587G + 0.114B$

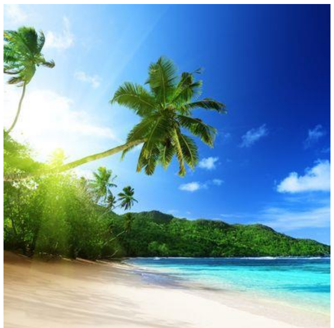
$Cb = 128 - 0.168736R - 0.331264G + 0.5B$

$Cr \ = 128 + 0.5R - 0.418688G - 0.081312$

Human eyes are far more receptive to the brightness and the darkness of an image therefore the

Here is the python function to convert a RGB pixel to YCrCb.

```
# Convert BGR to YCrCb
def ConvertToYCrCb(bgr_pixel):
    B, G, R = bgr_pixel
    Y = 0.299 * R + 0.587 * G + 0.114 * B
    Cb = 128 - 0.168736 * R - 0.331264 * G + 0.5 * B
    Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B
    return [Y, Cb, Cr]

bgr_image = cv2.imread('/content/beach.jpg')
height, width = len(bgr_image), len(bgr_image[0])
ycbcr_image = np.zeros((height, width, 3), dtype=np.uint8)

for i in range(height):
    for j in range(width):
        ycbcr_image[i, j] = ConvertToYCrCb(bgr_image[i][j])

cv2.imwrite('YCrCbImage2.jpg', ycbcr_image)
```



400 X 400 RGB image

YCrCb image

2.Discrete Cosine Transformation

First step of this is to divide the entire image into 8x8 tiles called blocks, each with 64 pixels with values from 0 to 255 that represent the luminance of every pixel. Then Discrete Cosine Transformation is applied on each block of data. After applying Discrete Cosine Transformation most of the data will be low frequency components.

$$b(u,v)=\frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}a(x,y)\cos\left(\frac{\pi u(2x+1)}{2N}\right)\cos\left(\frac{\pi v(2y+1)}{2N}\right)$$

Following code shows how the image is divided into 8x8 blocks.

```
# Iterate through the image and extract 8x8 blocks
blocks = []
for i in range(0, height, 8):
    for j in range(0, width, 8):
        block = ycbcr_image[i:i + 8, j:j + 8]
        blocks.append(block)
```

This is the function for Discrete Cosine Transformation.

```
def dctTransform(block):
  m=8
  n=8
  pi = 3.142857
  dct_block = [[0.0] * 8 for _ in range(8)]
  for u in range(8):
        for v in range(8):
            sum_val = 0.0

            for x in range(8):
```

```
                for y in range(8):
                    if u == 0:
                        cu = 1.0
                    else:
                        cu = 1.4142135623730951

                    if v == 0:
                        cv = 1.0
                    else:
                        cv = 1.4142135623730951
                    cos_x = np.cos((2 * x + 1) * u * np.pi / 16)
                    cos_y = np.cos((2 * y + 1) * v * np.pi / 16)
                    #print(block[x][y])
                    sum_val += block[x][y] * cos_x * cos_y
                    #print (sum_val)

            dct_block[u][v] = 0.25 * cu * cv * sum_val
    return dct_block
```

3.Quantization

Quantization is used to reduce the amount of data needed to represent an image. Quantization involves dividing DCT coefficients by a set of values in a quantization matrix.

$$\textbf{Quantized Value(i, j)} = \frac{\textbf{DCT(i, j)}}{\textbf{Quantum(i, j)}} \quad \textbf{Rounded to the nearest integer}$$

To get the quantum values standard luminance and chrominance quantization matrices are used.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

(a) Luminance quantization matrix

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

(b) Chrominance quantization matrix

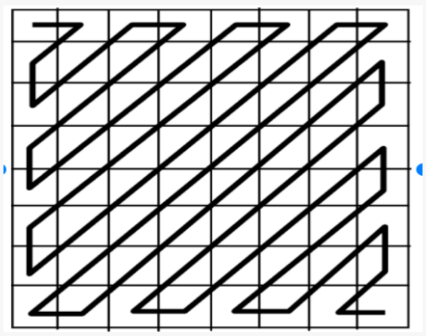
Here is the code for to perform Quantization

```
quantized_blocks = []
for dct_block in dct_blocks:
    quantized_block = []

    for u in range(8):
        quantized_row = []

        for v in range(8):
            quantized_coeff
=Quantization(dct_block[u][v],quantization_table_y[u][v],quantization_table_c
[u][v])
            quantized_row.append(quantized_coeff)

        quantized_block.append(quantized_row)

    quantized_blocks.append(quantized_block)
```

```
def Quantization(YCrCb_pixel,luminance_value,chrominance_value):
  Y, Cr, Cb = YCrCb_pixel
  X=round(Y/luminance_value)
  Y=round(Cr/chrominance_value)
  Z=round(Cb/chrominance_value)
  return [X, Y, Z]
```

5.Zigzag Scanning

Zigzag scanning is applied to the quantized DCT coefficient blocks to convert them into a linear array before further compression.



Here is the code for Zigzag scanning for the luminance values

```
#  zigzag scan for a given 8x8 quantized block
def zigzag_scan(quantized_block):
    rows, cols = len(quantized_block), len(quantized_block[0])
    result = []
```

```python
    for i in range(rows + cols - 1):
        if i % 2 == 0:
            # Move up the diagonal
            row = max(0, i - cols + 1)
            col = min(i, cols - 1)
            while col >= 0 and row < rows:
                result.append(quantized_block[row][col][0])
                row += 1
                col -= 1
        else:
            # Move down the diagonal
            row = min(i, rows - 1)
            col = max(0, i - rows + 1)
            while row >= 0 and col < cols:
                result.append(quantized_block[row][col][0])
                row -= 1
                col += 1

    return result

#Create 1D list of the elements in zigzag order
zigzag_encoded = []
for quantized_block in quantized_blocks:
    zigzag_result = zigzag_scan(quantized_block)
    zigzag_encoded.append(zigzag_result)
```

Sample output format:
(This is a one 8x8 block (quantized_block) for Luminance values)
[240, -1, -4, -1, 0, 1, 1, -2, 0, 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, 0, 0, 0, 0, 0, 0]

6.Run Length Encoding

Run length encoding represents repeated consecutive data values as a single value and its count.
Here is the Function for run length encoding.

```python
# Run length encoding
def run_length_encode(data):
    if not data:
        return []

    encoded_data = []
    current = data[0]
    count = 1
```

```
    for item in data[1:]:
        if item == current:
            count += 1
        else:
            encoded_data.append((current, count))
            current = item
            count = 1

    # Append the last run
    encoded_data.append((current, count))

    return encoded_data
```

Sample input:
(This is a one 8x8 block for Luminance vales)
[240, -1, -4, -1, 0, 1, 1, -2, 0, 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, 0, 0, 0, 0, 0, 0]

Sample output:
[(240, 1), (-1, 1), (-4, 1), (-1, 1), (0, 1), (1, 2), (-2, 1), (0, 1), (1, 1), (0, 54)]

Get the frequency of each symbol

```
# get the frequence of each symbol
symbol_frequencies = {}

for symbol, count in rl_encoded:
     if symbol in symbol_frequencies:
         symbol_frequencies[symbol] += count
     else:
         symbol_frequencies[symbol] = count
print(symbol_frequencies)
```

Sample input:
[(240, 1), (-1, 1), (-4, 1), (-1, 1), (0, 1), (1, 2), (-2, 1), (0, 1), (1, 1), (0, 54)]
Sample output:
`{240: 1, -1: 2, -4: 1, 0: 56, 1: 3, -2: 1}`

## 7.Huffman coding

This generates variable-length codes for frequent sequences of data. Finding the frequency of each symbol in the input data is the first step in the Huffman coding process. Here shorter codes are given to symbols with higher frequencies and longer codes to symbols with lower frequencies. Huffman coding builds a binary tree, where the leaves of the tree represent the symbols, and the binary codes are derived from the paths from the root to each leaf. Each symbol is given a distinct binary code once the Huffman tree has been built, which is determined by the path that connects the symbol's root to its matching leaf node. In order to create the codes, one usually goes around the tree from root to leaf, marking '0' for a left branch and '1' for a right branch. The input symbols are changed to their corresponding Huffman codes in order to compress the data. These variable-length binary codes are then used to represent all of the input data.

```
class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(frequencies):
    heap = [HuffmanNode(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        parent = HuffmanNode(None, left.freq + right.freq)
        parent.left = left
        parent.right = right
        heapq.heappush(heap, parent)

    return heap[0]

def build_huffman_encoding_table(node, current_code="", huffman_table=None):
    if huffman_table is None:
        huffman_table = {}

    if node.char is not None:
        huffman_table[node.char] = current_code
    if node.left is not None:
        build_huffman_encoding_table(node.left, current_code + "0",
huffman_table)
    if node.right is not None:
        build_huffman_encoding_table(node.right, current_code + "1",
huffman_table)

def huffman_encode(data):
    # Calculate symbol frequencies from the input data
    frequencies = defaultdict(int)
```

```
    for symbol, count in data:
      for i in symbol:
        if i in symbol_frequencies:
          frequencies[i] += count
        else:
          frequencies[i] = count
    #print(frequencies)


    # Build the Huffman tree
    root = build_huffman_tree(frequencies)

    # Build the Huffman encoding table
    huffman_table = {}
    build_huffman_encoding_table(root, huffman_table=huffman_table)

    # Encode the data using the Huffman table
    huffman_codes = []
    for symbol, count in data:
      for char in symbol:
        huffman_codes.append(huffman_table[char])

    huffman_encoded = "".join(huffman_codes)

    return huffman_encoded


huffman_encoded = huffman_encode(rl_encoded)

print(huffman_encoded)
```

This code output a string with 0 s and 1 s

End.