CS33713 – IMAGE PROCESSING

JPEG COMPRESSION

Chathura Gunasekara – 200193U

NOVEMBER 1, 2023

Introduction

Image compression refers to the process of reducing the size of digital images to save storage space and transmission bandwidth while maintaining an acceptable level of image quality.

Image compression is used for several reasons:

1. **Storage Space:** High-resolution images captured by modern cameras can consume a significant amount of storage space. By compressing these images, you can store more photos on devices with limited storage capacity.
2. **Transmission Efficiency:** When images need to be transmitted over the internet or other networks, compressed images require less bandwidth. This is crucial for websites, social media platforms, and mobile applications where users expect quick loading times.
3. **Faster Transmission:** Smaller image sizes lead to faster transmission, improving the user experience, especially in scenarios where images need to be loaded on webpages or apps.
4. **Cost Efficiency:** For businesses and organizations, reducing the size of images can lead to cost savings in terms of storage, bandwidth usage, and server infrastructure.
5. **Multi-Media Applications:** In multimedia applications, such as video streaming and online gaming, compressed images are essential components that contribute to efficient overall system performance.
6. **Data Integrity:** In some cases, compression algorithms also include error-detection and error-correction techniques, ensuring data integrity during transmission.

There are 2 forms image compression:

1. **Lossless Compression:** In lossless compression, no data is lost during the compression process. The original image can be perfectly reconstructed from the compressed version. Lossless compression is ideal for cases where preserving every detail is crucial, but it typically achieves lower compression ratios compared to lossy compression.
2. **Lossy Compression:** In lossy compression, some data is discarded during the compression process. While this results in a smaller file size, it also leads to a loss of image quality. The extent of quality loss can be controlled, allowing a balance between image quality and compression ratio.

In this report we will be analyzing the **JPEG Compression** algorithm

JPEG Compression

A widely used lossy compression standard for photographs and realistic images. It achieves good compression ratios with acceptable image quality.

JPEG Compression Pipeline – High Level Overview

JPEG Compression Pipeline - Detailed Overview

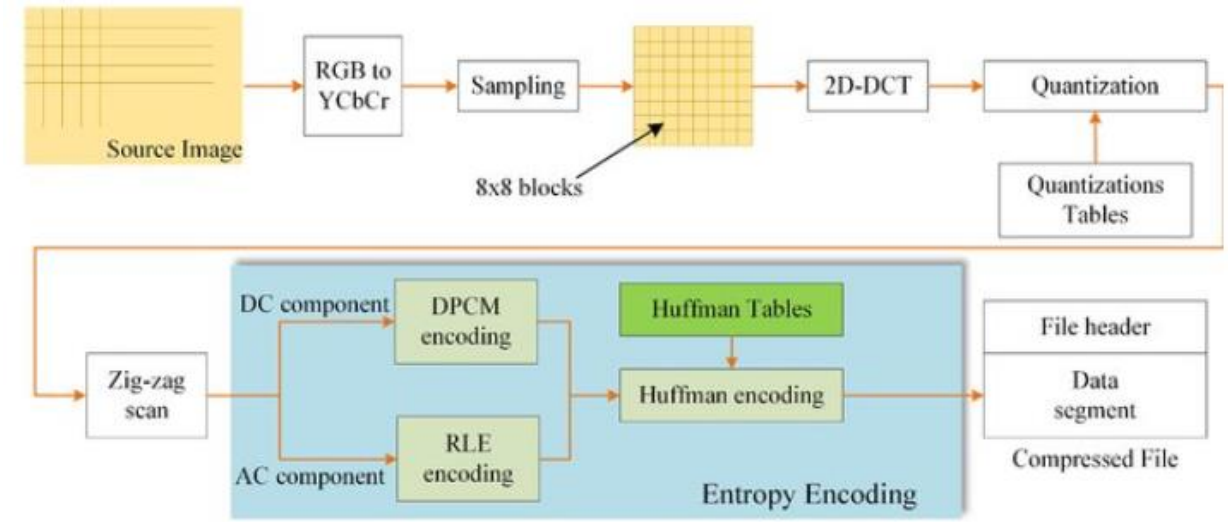

Figure 1- JPEG Compression Pipeline [1]

Forward DCT

The DCT Transform considers the input image as a *time varying signal* and decomposes the signal into regular cosine waves with different frequencies. The process involves 3 steps

1. **Convert image to YCrCb format** – The RGB color space is converted to YCbCr to separate luminance (Y) and chrominance (Cb and Cr) components. This conversion exploits the human eye's higher sensitivity to brightness (luminance) compared to color information.

   ***Importance**: Separating color information allows for more efficient compression, as chrominance components can be subsampled without significant loss in image quality.*

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

```python
# Convert RGB to YCbCr
def RGB2YCbCr(image: np.ndarray) -> np.ndarray:
    # Convert RGB to YCrCb
    offset = np.array([16, 128, 128])
    ycbcr_transform = np.array(
        [
            [0.257, 0.504, 0.098],
            [-0.148, -0.291, 0.439],
            [0.439, -0.368, -0.071],
        ]
    )

    ycbcr_image = np.zeros(image.shape)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            ycbcr_image[i, j] = np.round(np.dot(ycbcr_transform,
image[i, j]) + offset)

    return ycbcr_image
```

2. **Sample image into 8x8 blocks** - The image is divided into non-overlapping 8x8 blocks for further processing. Sampling ensures that each 8x8 block represents a portion of the image, allowing for localized transformations.

   ***Importance**: Working with small blocks simplifies the compression process. Each block can be processed independently, enabling parallelization and facilitating various compression techniques.*

   Sample the resulting image into 8x8 blocks. For illustration purposes, we will use the following 8x8 block to visualize the results of the preceding algorithms

```
Image:
52      55      61      66      70      61      64      73
63      59      55      90      109     85      69      72
62      59      68      113     144     104     66      73
63      58      71      122     154     106     70      69
67      61      68      104     126     88      68      70
79      65      60      70      77      68      58      75
85      71      64      59      55      61      65      83
87      79      69      68      65      76      78      94
```

Figure 2- Sample 8x8 image used

3. **Perform Discrete Cosine Transform(DCT) –** DCT is applied to each 8x8 block, transforming spatial domain pixel values into frequency domain coefficients. DCT helps concentrate most of the image information into a few low-frequency coefficients, making subsequent quantization and compression more effective.

   ***Importance**: DCT reduces spatial redundancy and prepares the data for quantization, making it suitable for lossy compression while minimizing perceptual loss.*

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where

- u is the horizontal spatial frequency, for the integers $0 \leq u < 8$.
- v is the vertical spatial frequency, for the integers $0 \leq v < 8$.
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases}$ is a normalizing scale factor to make the transformation orthonormal
- $g_{x,y}$ is the pixel value at coordinates (x, y)
- $G_{u,v}$ is the DCT coefficient at coordinates (u, v).

Figure 3 - DCT Equations [2]

Performing the DCT transform on a sample 8x8 matrix, is shown below

```python
# DCT function
def dct(image_block):
    dct_block = np.zeros_like(image_block, dtype=np.float32)
    for u in range(8):
        for v in range(8):
            cu = 1 if u == 0 else np.sqrt(2)
            cv = 1 if v == 0 else np.sqrt(2)
            sum_val = 0
```

```
            for x in range(8):
                for y in range(8):
                    sum_val += (
                        image_block[x, y]
                        * np.cos((2 * x + 1) * u * np.pi / 16)
                        * np.cos((2 * y + 1) * v * np.pi / 16)
                    )
            dct_block[u, v] = 0.25 * cu * cv * sum_val

    return dct_block
```

Input Matrix:

```
Image:
52      55      61      66      70      61      64      73
63      59      55      90      109     85      69      72
62      59      68      113     144     104     66      73
63      58      71      122     154     106     70      69
67      61      68      104     126     88      68      70
79      65      60      70      77      68      58      75
85      71      64      59      55      61      65      83
87      79      69      68      65      76      78      94
```

Output Matrix:

```
DCT block:
1217    -60     -122    54      112     -40     -5      1
9       -44     -122    21      26      -14     -17     10
-94     15      154     -49     -58     20      11      -11
-97     24      68      -30     -20     13      4       4
24      -13     -26     -8      -4      3       -6      6
-15     6       5       -12     -5      2       9       4
-2      0       1       -5      -2      -6      8       -1
0       0       -2      -8      -2      0       1       3
```

Quantization

Quantization involves dividing DCT coefficients by a quantization matrix. This step introduces loss by mapping a wide range of values to a limited set. Higher frequencies, which are less perceptible to the human eye, are quantized more heavily, contributing to data reduction.

***Importance**: Quantization is a key lossy step, enabling significant data reduction. The choice of quantization matrix balances compression ratio and image quality.*

```python
# Quantizing constants
luminance_quantization_matrix = 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],
    ],
    dtype=np.float32,
)

chrominance_quantization_matrix = 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],
    ],
    dtype=np.float32,
)


def quantize_block(block, quantization_matrix):
    return np.round(block / quantization_matrix)


# Quantization function
def quantize(dct_block):
    quantized_luminance_block = quantize_block(
        dct_block[..., 0], luminance_quantization_matrix
    )
    quantized_chrominance_blocks = [
        quantize_block(dct_block[..., i], chrominance_quantization_matrix)
        for i in range(1, 3)
    ]

    return quantized_luminance_block, quantized_chrominance_blocks
```

Note: *The human eye is more sensitive to changes in luminance (brightness) than to changes in chrominance (color). As a result, higher levels of quantization can be applied to chrominance components without significantly affecting the perceived image quality. This perceptual characteristic is leveraged in JPEG compression to achieve higher compression ratios while maintaining acceptable image quality. This is why 2 different quantization matrices are used*

Input Matrix:

```
DCT block:
1217    -60     -122    54      112     -40     -5      1
9       -44     -122    21      26      -14     -17     10
-94     15      154     -49     -58     20      11      -11
-97     24      68      -30     -20     13      4       4
24      -13     -26     -8      -4      3       -6      6
-15     6       5       -12     -5      2       9       4
-2      0       1       -5      -2      -6      8       -1
0       0       -2      -8      -2      0       1       3
```

Output Matrix:

```
Quantized Luminance block
76      1       -9      -6      1       0       0       0
101     1       -7      -5      1       0       0       0
87      1       -6      -4      1       0       0       0
87      1       -4      -3      0       0       0       0
68      0       -3      -2      0       0       0       0
51      0       -2      -2      0       0       0       0
25      0       -1      -1      0       0       0       0
17      0       -1      -1      0       0       0       0
```

Encoding

Entropy encoding is a data compression technique used to represent data in the most efficient way possible, reducing the amount of data needed to convey information. It takes advantage of the statistical properties of the input data, assigning shorter codes to more frequently occurring symbols and longer codes to less common symbols.

In JPEG Compression, Entropy encoding is done in 3 steps

1. **ZigZag reordering of DCT coefficients -** Zigzag scanning reorganizes quantized DCT coefficients into a linear array. This process groups coefficients with similar frequencies together, increasing the efficiency of run-length encoding and entropy coding.

***Importance**: Zigzag scanning prepares the data for run-length encoding, ensuring that non-zero coefficients are efficiently represented.*

```
# zigzag scan
def zigzag_scan(block):
    zigzag = []
    for i in range(15):
        if i < 8:
            for j in range(i + 1):
                if i % 2 == 0:
                    zigzag.append(block[i - j][j])
                else:
                    zigzag.append(block[j][i - j])
        else:
            for j in range(15 - i):
                if i % 2 == 0:
                    zigzag.append(block[7 - j][j + (i - 7)])
                else:
                    zigzag.append(block[j + (i - 7)][7 - j])

    return np.array(zigzag)
```

Input Matrix:

```
Quantized Luminance block
76      1       -9      -6      1       0       0       0
101     1       -7      -5      1       0       0       0
87      1       -6      -4      1       0       0       0
87      1       -4      -3      0       0       0       0
68      0       -3      -2      0       0       0       0
51      0       -2      -2      0       0       0       0
25      0       -1      -1      0       0       0       0
17      0       -1      -1      0       0       0       0
```

Output Matrix:

```
ZigZag scan
[ 76.   1. 101.  87.   1.  -9.  -6.  -7.   1.  87.  68.   1.  -6.  -5.
   1.  -0.   1.  -4.  -4.   0.  51.  25.   0.  -3.  -3.   1.  -0.  -0.
  -0.  -0.  -0.   0.  -2.  -2.   0.  17.   0.  -1.  -2.   0.  -0.  -0.
  -0.  -0.  -0.  -0.   0.  -1.  -1.  -1.   0.  -0.  -0.  -0.  -0.  -0.
  -0.   0.  -0.  -0.  -0.  -0.  -0.  -0.]
```

2. **Apply DPCM encoding for the DC Component and RLE encoding for the AC Components -** DPCM captures the relative changes in brightness for the DC component, ensuring efficient representation and lossless compression. RLE targets sequences of zeros in the AC components, enabling the compression of sparse data structures and eliminating redundancy.

   Importance*: These techniques enhance storage, transmission, and processing efficiency while preserving the original data quality, making them fundamental components of many image compression algorithms.*

```python
# DPCM encoding for DC component

def dpcm_encode_dc(dc_coefficients):
    dpcm_encoded_dc = [dc_coefficients[0]]
    for i in range(1, len(dc_coefficients)):
        diff = dc_coefficients[i] - dc_coefficients[i - 1]
        dpcm_encoded_dc.append(diff)
    return dpcm_encoded_dc


# RLE for AC components
def run_length_encode_ac(ac_coefficients):
    rle_encoded_ac = []
    run_length = 1
    prev_chr = ac_coefficients[0]
    for ac_coefficient in ac_coefficients[1:]:
        if ac_coefficient == prev_chr:
            run_length += 1
        else:
            # Append (run length, value) pair to the encoded list
            rle_encoded_ac.extend([prev_chr, run_length])
            prev_chr = ac_coefficient
            run_length = 1

    # Add the last entry
    rle_encoded_ac.extend([prev_chr, run_length])
    return rle_encoded_ac
```

Input Array:

```
ZigZag scan
[ 76   1 101  87   1  -9  -6  -7   1  87  68   1  -6  -5   1   0   1  -4
  -4   0  51  25   0  -3  -3   1   0   0   0   0   0   0  -2  -2   0  17
   0  -1  -2   0   0   0   0   0   0   0   0  -1  -1  -1   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
```

Output Array:

```
Encoded values:
[ 76   1   1 101   1  87   1   1   1  -9   1  -6   1  -7   1   1   1  87
   1  68   1   1   1  -6   1  -5   1   1   1   0   1   1   1  -4   2   0
   1  51   1  25   1   0   1  -3   2   1   1   0   6  -2   2   0   1  17
   1   0   1  -1   1  -2   1   0   8  -1   3   0  14]
```

3. **Huffman Encoding -** Huffman encoding is a fundamental technique in data compression, leveraging the frequency distribution of characters to generate efficient variable-length codes.

   Importance: *This method achieves compact data representation, enabling significant reduction in storage space and transmission bandwidth*

   To perform Huffman encoding, first the frequency distribution must be analyzed, and the frequency dictionary should be constructed

```python
import heapq
from collections import defaultdict

# Step 1: Find frequency dictionary
def build_frequency_dict(data):
    frequency_dict = defaultdict(int)
    for char in data:
        frequency_dict[char] += 1
    return frequency_dict
```

   Next, perform the Huffman algorithm to generate the Huffman tree using the frequency dictionary created above.

```python
# Step 2: Huffman algorithm
def build_huffman_tree(frequency_dict):
    priority_queue = [[weight, [char, ""]] for char, weight in
frequency_dict.items()]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        lo = heapq.heappop(priority_queue)
        hi = heapq.heappop(priority_queue)
        for pair in lo[1:]:
            pair[1] = "0" + pair[1]
        for pair in hi[1:]:
            pair[1] = "1" + pair[1]
```

```
        heapq.heappush(priority_queue, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return priority_queue[0]
```

Next create the Huffman code from the Huffman tree

Input Array:

```
Encoded values:
[ 76   1   1 101   1  87   1   1   1  -9   1  -6   1  -7   1   1   1  87
   1  68   1   1   1  -6   1  -5   1   1   1   0   1   1   1  -4   2   0
   1  51   1  25   1   0   1  -3   2   1   1   0   6  -2   2   0   1  17
   1   0   1  -1   1  -2   1   0   8  -1   3   0  14]
```

Output Dictionary:

```
Frequency dictionary:
{76: 1, 1: 33, 101: 1, 87: 2, -9: 1, -6: 2, -7: 1, 68: 1, -5: 1, 0: 8
, -4: 1, 2: 3, 51: 1, 25: 1, -3: 1, 6: 1, -2: 2, 17: 1, -1: 2, 8: 1,
3: 1, 14: 1}
```

Huffman Tree:

```
Huffman Tree
[67, [1, '0'], [-6, '10000'], [-5, '100010'], [-4, '100011'], [-3, '100100']
, [3, '100101'], [-2, '10011'], [-1, '10100'], [6, '101010'], [8, '101011'],
 [14, '101100'], [17, '101101'], [25, '101110'], [51, '101111'], [0, '110'],
 [68, '111000'], [76, '111001'], [87, '11101'], [2, '11110'], [101, '111110'
], [-9, '1111110'], [-7, '1111111']]
```

Huffman Codes:

```
Huffman Codes:
{1: '0', -6: '10000', -5: '100010', -4: '100011', -3: '100100', 3: '100101', -2: '10011', -1
: '10100', 6: '101010', 8: '101011', 14: '101100', 17: '101101', 25: '101110', 51: '101111',
 0: '110', 68: '111000', 76: '111001', 87: '11101', 2: '11110', 101: '111110', -9: '1111110'
, -7: '1111111'}
```

Finally, generate the encoded string using the generated Huffman codes

```
    # Huffman encoding

def huffman_encode(rle, codes):
    encoded_str = ""
    for sym in rle:
        encoded_str += codes[sym]

    return encoded_str
```

Input Array:

```
Encoded values:
[ 76   1   1 101   1  87   1   1   1  -9   1  -6   1  -7   1   1   1  87
   1  68   1   1   1  -6   1  -5   1   1   1   0   1   1   1  -4   2   0
   1  51   1  25   1   0   1  -3   2   1   1   0   6  -2   2   0   1  17
   1   0   1  -1   1  -2   1   0   8  -1   3   0  14]
```

Huffman Encoded String:

```
Huffman Encoded String:
11100100111110011101000111111001000001111111000111010111000000100000100010000110000100011111
10110010111101011100110010010011110001101010101001111101100101101011001010001001101101010111
10100100101110101100
```

Conclusion

This is a demonstration of how the JPEG Compression on one channel of a image that is sampled into 8x8 blocks. This algorithm is performed on all the channels of the image and finally the entire image will be encoded.

References

[1] Zhu, Fushun & Yan, Hua. (2022). An efficient parallel entropy coding method for JPEG compression based on GPU. The Journal of Supercomputing. 78. 1-28. 10.1007/s11227-021-03971-6.

[2] https://stackoverflow.com/questions/54295413/change-code-from-dct-to-inverse-discrete-cosine-transformation