CS3713 - IMAGE PROCESSING

JPEG Image Compression

Contents

Introduction

JPEG, short for Joint Photographic Experts Group, is a widely adopted image compression standard. It plays a crucial role in reducing the storage and transmission requirements for images, making them more manageable in the digital world. By employing a lossy compression technique, JPEG minimizes file sizes by removing less perceptible image details. This trade-off between image quality and file size makes JPEG a go-to choose for various applications.

Basic Steps in JPEG Compression

1. Color Space Conversion
2. Discrete Cosine Transform (DCT)
3. Quantization of DCT Coefficients
4. Zigzag Scanning and Coefficient Reordering
5. Run-Length Encoding (RLE)
6. Huffman Encoding for Entropy Coding

Color Space Conversion

Color space conversion is the process of transforming an image's representation from one color space to another. In this context, RGB images are commonly converted to the YCrCb color space, which separates the image into three components:

- Y (Luminance): Represents the image's brightness and grayscale information.
- Cr (Chrominance Red): Describes the deviation from gray in the red channel.
- Cb (Chrominance Blue): Describes the deviation from gray in the blue channel.

Conversion involves linear transformations of the RGB channels. The motivation is to align with human visual perception, which is more sensitive to changes in brightness than color.

```
# Function to convert an RGB image to the YCrCb color space
def rgb_to_YCrCb(image, height, width):
    ycrcb_image = np.zeros((height, width, 3), dtype=np.uint8)
    for i in range(height):
        for j in range(width):
            r, g, b = image[i, j]
            y = 0.299 * r + 0.587 * g + 0.114 * b
            cr = int(128 + 0.5 * r - 0.418688 * g - 0.081312 * b)
            cb = int(128 - 0.168736 * r - 0.331264 * g + 0.5 * b)
            ycrcb_image[i, j] = [y, cr, cb]
    return ycrcb_image
```

Discrete Cosine Transform (DCT)

Dividing an image into 8x8 blocks and applying the Discrete Cosine Transform (DCT) to each block is a common technique used in image processing, notably in JPEG compression. The primary goals of this process are to segment the image into manageable chunks and transform spatial information into frequency information.

- The image is partitioned into multiple non-overlapping 8x8 blocks. Each block is precisely 8 pixels wide and 8 pixels high. These blocks are chosen without any overlap, ensuring that every pixel in the image is encompassed within exactly one 8x8 block.
- For each 8x8 block, the Discrete Cosine Transform (DCT) is applied. The DCT converts the pixel values from the spatial domain to frequency coefficients in the frequency domain. This transformation allows the image data to be represented in terms of different frequency components:
    - Lower Frequencies: Represent the general image structure or color variations.
    - Higher Frequencies: Represent finer details or sharp transitions within the image.
- The DCT provides a way to compact most of the image's information into a few coefficients, useful for image compression while preserving perceived image quality.

The equation for Discrete Cosine Transform (DCT) is as follows:

$$\mathrm{DCT}(u,v) = \tfrac{1}{2}C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right)$$

where:

- $f(i,j)$ represents the pixel values in an 8×8 block.
- u and v are the frequencies in the horizontal and vertical directions, respectively.
- $C(u)$ and $C(v)$ are normalization constants $\frac{1}{\sqrt{2}}$ for $u = 0$ or $v = 0$, and 1 for $u \neq 0$ and $v \neq 0$.

```
# Function to extract an 8x8 block from an image at position (x, y)
def block_8x8(image, x, y):
    block = []
    for i in range(8):
        row = []
        for j in range(8):
            row.append(image[y + i][x + j])
        block.append(row)
    return block

# Function to select a specific channel (Y, Cr, or Cb) from an 8x8 block
def select_channel(block, channel):
    block_channel = []
```

```
    index = 0 if channel == 'Y' else (1 if channel == 'Cr' else 2)
    for i in range(8):
        row = []
        for j in range(8):
            row.append(block[i][j][index])
        block_channel.append(row)
    return block_channel

# Function to calculate the 2D Discrete Cosine Transform (DCT) of an 8x8
block
def dct(block):
    N = 8
    dct_result = np.zeros((N, N), dtype=float)

    for u in range(N):
        for v in range(N):
            sum_val = 0

            for i in range(N):
                for j in range(N):
                    cu = 1 if u == 0 else math.sqrt(2)  # Adjust cu
                    cv = 1 if v == 0 else math.sqrt(2)  # Adjust cv
                    cos_u = math.cos((2 * i + 1) * u * math.pi / (2 * N))
                    cos_v = math.cos((2 * j + 1) * v * math.pi / (2 * N))

                    sum_val += cu * cv * block[i][j] * cos_u * cos_v

            dct_result[u][v] = 0.25 * sum_val

    return dct_result
```

Quantization of DCT Coefficients

Quantization is a crucial step in JPEG compression that significantly reduces the amount of data needed to represent an image while balancing image quality. During quantization, DCT coefficients are divided by specific quantization matrices, reducing their precision.

JPEG uses standard quantization matrices for luminance (Y) and chrominance (Cr and Cb) channels:

- Luminance Quantization Matrix: Preserves more detail in the grayscale (Y) channel, which is more sensitive to brightness variations.
- Chrominance Quantization Matrix: Reduces the precision of color information in the Cr and Cb channels while maintaining acceptable visual quality.

Quantization Process: The process applies the appropriate quantization matrix to each 8x8 DCT coefficient block. Each coefficient is divided by the corresponding value in the matrix

and rounded to an integer. The quantized coefficients are used for compression and encoding.

```
# Standard quantization matrices for luminance and chrominance in JPEG
compression
luminance_quantization_matrix = [
    [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]
]

chrominance_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]
]

# Function to apply quantization to an 8x8 block
def quantize_block(dct_block, quantization_matrix):
    quantized_block = [[0] * 8 for _ in range(8)]

    for i in range(8):
        for j in range(8):
            quantized_block[i][j] = round(dct_block[i][j] /
quantization_matrix[i][j])

    return quantized_block
```

Zigzag Scanning and Coefficient Reordering

Certainly, in JPEG compression, a zigzag pattern is used to store quantized Discrete Cosine Transform (DCT) coefficients efficiently. The generate_zigzag_pattern function arranges these coefficients in a one-dimensional list, optimizing data storage. It iterates through the upper and lower triangles of the DCT coefficient matrix, alternating directions to create the zigzag pattern. This ordered list is essential for subsequent run-length encoding and entropy coding, allowing for efficient compression while preserving crucial information. Your example demonstrates how this process transforms quantized coefficients into an ordered_list, simplifying data representation and aiding compression.

```
def generate_zigzag_pattern(input_matrix):
    dimension = len(input_matrix)
    zigzag_result = []

    # Iterate over the upper triangle of the matrix
    for i in range(dimension):
        for j in range(i + 1):
            if i % 2 == 0:
                zigzag_result.append(input_matrix[j][i - j])
            else:
                zigzag_result.append(input_matrix[i - j][j])

    # Iterate over the lower triangle of the matrix
    for i in range(1, dimension):
        for j in range(dimension - i):
            if (i + j) % 2 == 0:
               zigzag_result.append(input_matrix[dimension - 1 - j][i + j])
            else:
               zigzag_result.append(input_matrix[i + j][dimension - 1 - j])

    return zigzag_result
```

Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is a compression method that condenses consecutive identical values by replacing them with the value followed by a count of repetitions. This simplifies data storage by efficiently representing long runs of the same value. In JPEG compression, RLE is a vital step in reducing data size and enabling efficient encoding.

```
def run_length_encoding(data):
    encoded_data = []
    count = 1

    for i in range(1, len(data)):
        if data[i] == data[i - 1]:
            count += 1
        else:
```

```
            encoded_data.extend([data[i - 1], count])
            count = 1

    encoded_data.extend([data[-1], count])
    return encoded_data
```

Huffman Coding in JPEG Compression

Huffman coding is employed to efficiently encode data. In this code, it's applied to the Run-Length Encoding (RLE) output. The process begins with creating a frequency dictionary to track the occurrence of values. A Huffman tree is built based on these frequencies, and Huffman codes are assigned to each value. The encoded data is generated by converting RLE values into their corresponding Huffman codes. This step significantly reduces data size, making it a crucial component in the JPEG compression process. The example showcases the generation of Huffman codes and encoding, resulting in more compact data representation.

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

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

def calculate_symbol_frequencies(data):
    symbol_frequencies = defaultdict(int)
    for symbol in data:
        symbol_frequencies[symbol] += 1

    return symbol_frequencies

def build_huffman_tree_from_frequencies(symbol_frequencies):
    priority_queue = [HuffmanNode(freq, symbol) for symbol, freq in
symbol_frequencies.items()]
    heapq.heapify(priority_queue)

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

    return heapq.heappop(priority_queue)
```

```
def generate_huffman_codes_from_tree(huffman_tree):
    def assign_huffman_codes(node, code, huffman_codes):
        if node.symbol is not None:
            huffman_codes[node.symbol] = code
        else:
            assign_huffman_codes(node.left, code + "0", huffman_codes)
            assign_huffman_codes(node.right, code + "1", huffman_codes)

    huffman_codes = {}
    assign_huffman_codes(huffman_tree, "", huffman_codes)
    return huffman_codes

data = rle_result
symbol_frequencies = calculate_symbol_frequencies(data)
huffman_tree = build_huffman_tree_from_frequencies(symbol_frequencies)
huffman_codes = generate_huffman_codes_from_tree(huffman_tree)

def encode_data_with_huffman(data, huffman_codes):
    encoded_data = "".join(huffman_codes[symbol] for symbol in data)
    return encoded_data

encoded_data = encode_data_with_huffman(rle_result, huffman_codes)
```

Conclusion

In summary, JPEG compression employs a series of techniques to reduce the size of digital images while maintaining acceptable visual quality. This process begins with color space conversion to separate luminance and chrominance components. The application of the Discrete Cosine Transform (DCT) divides the image into 8x8 blocks and transforms spatial information into frequency components. Quantization further reduces data size by scaling DCT coefficients with quantization matrices. Zigzag scanning optimizes data storage, enabling efficient run-length encoding and entropy coding. Run-Length Encoding (RLE) simplifies data representation by grouping consecutive identical values, while Huffman coding efficiently encodes the RLE output. These techniques collectively strike a balance between file size reduction and image quality, making JPEG compression a cornerstone of digital image processing.