image-processing-coding

November 3, 2024

[10]:
```
import numpy as np
from scipy.fftpack import dct, idct
from PIL import Image
import heapq
from collections import defaultdict
import math
import struct

def dct_compression(image, quality_factor=30):
    """Transform Coding using DCT with improved compression"""
    height, width = image.shape
    block_size = 8
    compressed_data = []

    for i in range(0, height, block_size):
        for j in range(0, width, block_size):
            block = image[i:i+block_size, j:j+block_size]
            if block.shape[0] != block_size or block.shape[1] != block_size:
                padded = np.zeros((block_size, block_size))
                padded[:block.shape[0], :block.shape[1]] = block
                block = padded

            dct_block = dct(dct(block.T, norm='ortho').T, norm='ortho')
            q_matrix = np.ones((block_size, block_size)) * quality_factor
            q_matrix[0:4, 0:4] = quality_factor // 2

            quantized = np.round(dct_block / q_matrix)
            compressed_data.append(quantized)

    reconstructed = np.zeros_like(image)
    block_idx = 0

    for i in range(0, height, block_size):
        for j in range(0, width, block_size):
            dct_block = compressed_data[block_idx] * q_matrix
            block = idct(idct(dct_block.T, norm='ortho').T, norm='ortho')
            h = min(block_size, height-i)
```

```
            w = min(block_size, width-j)
            reconstructed[i:i+h, j:j+w] = block[:h, :w]
            block_idx += 1

    return np.uint8(np.clip(reconstructed, 0, 255))

def huffman_encoding(image):
    """Huffman Encoding with chunk processing"""
    def chunks(data, size=1024):
        for i in range(0, len(data), size):
            yield data[i:i+size]

    frequencies = defaultdict(int)
    for pixel in image.flatten():
        frequencies[pixel] += 1

    heap = [[freq, [pixel, ""]] for pixel, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    huffman_dict = dict(heap[0][1:])

    # Process in chunks
    encoded_bytes = bytearray()
    current_byte = 0
    bit_count = 0

    for chunk in chunks(image.flatten()):
        for pixel in chunk:
            code = huffman_dict[pixel]
            for bit in code:
                current_byte = (current_byte << 1) | (1 if bit == '1' else 0)
                bit_count += 1
                if bit_count == 8:
                    encoded_bytes.append(current_byte)
                    current_byte = 0
                    bit_count = 0

    # Handle remaining bits
```

```python
    if bit_count > 0:
        current_byte = current_byte << (8 - bit_count)
        encoded_bytes.append(current_byte)

    return encoded_bytes, (huffman_dict, bit_count, image.shape)

def lzw_encoding(image):
    """LZW Encoding with chunk processing"""
    chunk_size = 1024
    dictionary = {bytes([i]): i for i in range(256)}
    dictionary_size = 256
    compressed = bytearray()

    def encode_chunk(chunk):
        nonlocal dictionary_size
        current = bytes([chunk[0]])
        result = bytearray()

        for byte in chunk[1:]:
            byte = bytes([byte])
            temp = current + byte
            if temp in dictionary:
                current = temp
            else:
                result.extend(dictionary[current].to_bytes(2, byteorder='big'))
                if dictionary_size < 65536:
                    dictionary[temp] = dictionary_size
                    dictionary_size += 1
                current = byte

        if current:
            result.extend(dictionary[current].to_bytes(2, byteorder='big'))

        return result

    # Process image in chunks
    data = image.tobytes()
    for i in range(0, len(data), chunk_size):
        chunk = data[i:i+chunk_size]
        compressed.extend(encode_chunk(chunk))

    return compressed, image.shape

def rle_encoding(image):
    """RLE with chunk processing"""
    chunk_size = 1024
    encoded = bytearray()
```

```
    def encode_chunk(chunk):
        result = bytearray()
        count = 1
        current = chunk[0]

        for pixel in chunk[1:]:
            if pixel == current and count < 255:
                count += 1
            else:
                result.append(count)
                result.append(current)
                count = 1
                current = pixel

        result.append(count)
        result.append(current)
        return result

    # Process image in chunks
    flat_image = image.flatten()
    for i in range(0, len(flat_image), chunk_size):
        chunk = flat_image[i:i+chunk_size]
        encoded.extend(encode_chunk(chunk))

    return encoded, image.shape

def arithmetic_encoding(image):
    """Arithmetic Encoding with reduced precision"""
    PRECISION = 16  # Reduced precision to avoid overflow
    ONE = 2 ** PRECISION
    HALF = ONE >> 1
    QUARTER = HALF >> 1

    frequencies = defaultdict(int)
    total = 0
    for symbol in image.flatten():
        frequencies[symbol] += 1
        total += 1

    cumul = {}
    acc = 0
    for symbol in sorted(frequencies):
        cumul[symbol] = (acc, acc + frequencies[symbol])
        acc += frequencies[symbol]

    low, high = 0, ONE
```

```python
    encoded = bytearray()
    bits_to_follow = 0  # Initialize bits_to_follow here

    def output_bit(bit):
        nonlocal encoded, bits_to_follow  # Add bits_to_follow to nonlocal␣
↪declaration
        encoded.append(bit)
        while bits_to_follow > 0:
            encoded.append(1 - bit)
            bits_to_follow -= 1

    # Process each symbol
    for symbol in image.flatten():
        range_size = high - low
        high = low + (range_size * cumul[symbol][1]) // total
        low = low + (range_size * cumul[symbol][0]) // total

        while True:
            if high < HALF:
                output_bit(0)
            elif low >= HALF:
                output_bit(1)
                low -= HALF
                high -= HALF
            elif low >= QUARTER and high < 3*QUARTER:
                bits_to_follow += 1
                low -= QUARTER
                high -= QUARTER
            else:
                break
            low *= 2
            high *= 2
            if low >= ONE:
                low -= ONE
            if high >= ONE:
                high -= ONE

    # Output final bits
    bits_to_follow += 1
    if low < QUARTER:
        output_bit(0)
    else:
        output_bit(1)

    return encoded, (frequencies, total, image.shape)

def calculate_metrics(original, compressed_size):
```

```python
    """Calculate Compression Ratio"""
    original_size = original.nbytes
    return original_size / compressed_size

def main():
    # Load image
    image = np.array(Image.open('image.png').convert('L'))

    # Test compressions
    reconstructed_dct = dct_compression(image)
    cr_dct = calculate_metrics(image, reconstructed_dct.nbytes)
    rmse_dct = np.sqrt(np.mean((image - reconstructed_dct) ** 2))

    compressed_huff, _ = huffman_encoding(image)
    cr_huff = calculate_metrics(image, len(compressed_huff))

    compressed_lzw, _ = lzw_encoding(image)
    cr_lzw = calculate_metrics(image, len(compressed_lzw))

    compressed_rle, _ = rle_encoding(image)
    cr_rle = calculate_metrics(image, len(compressed_rle))

    compressed_arith, _ = arithmetic_encoding(image)
    cr_arith = calculate_metrics(image, len(compressed_arith))

    # Print results
    print(f"DCT Compression:")
    print(f"Compression Ratio: {cr_dct:.2f}")
    print(f"RMSE: {rmse_dct:.2f}\n")

    print(f"Huffman Compression:")
    print(f"Compression Ratio: {cr_huff:.2f}")
    print(f"RMSE: 0.00\n")

    print(f"LZW Compression:")
    print(f"Compression Ratio: {cr_lzw:.2f}")
    print(f"RMSE: 0.00\n")

    print(f"RLE Compression:")
    print(f"Compression Ratio: {cr_rle:.2f}")
    print(f"RMSE: 0.00\n")

    print(f"Arithmetic Compression:")
    print(f"Compression Ratio: {cr_arith:.2f}")
    print(f"RMSE: 0.00\n")

if __name__ == "__main__":
```

```
    main()
```

```
DCT Compression:
Compression Ratio: 1.00
RMSE: 2.41

Huffman Compression:
Compression Ratio: 1.44
RMSE: 0.00

LZW Compression:
Compression Ratio: 2.17
RMSE: 0.00

RLE Compression:
Compression Ratio: 1.07
RMSE: 0.00

Arithmetic Compression:
Compression Ratio: 0.18
RMSE: 0.00
```