untitled31

October 30, 2024

1. Implement functions for encoding and decoding an image using the following methods:

A. Transform Coding (using DCT for forward transform)

B. Huffman Encoding

C. LZWEncoding

D. Run-Length Encoding

E. Arithmetic Coding

For each method, display the Compression Ratio and calculate the Root Mean Square Error (RMSE) between the original and reconstructed image to quantify any loss of information.

[1]:
```
# Importing Libraries

import numpy as np
import cv2
import heapq
from collections import Counter, defaultdict
import math
from scipy.fftpack import dct, idct
from sklearn.metrics import mean_squared_error
```

[2]:
```
# Load image
def load_image(path, grayscale=True):
    image = cv2.imread(path, cv2.IMREAD_GRAYSCALE if grayscale else cv2.
↪IMREAD_COLOR)
    return image
```

[3]:
```
# RMSE Calculation
def calculate_rmse(original, reconstructed):
    return np.sqrt(mean_squared_error(original, reconstructed))
```

[4]:
```
# Compression Ratio Calculation
def calculate_compression_ratio(original_size, compressed_size):
    return original_size / compressed_size
```

A. Transform Coding (Using DCT for Forward Transform)

[5]:
```python
def dct_transform(image, block_size=8):
    height, width = image.shape
    dct_image = np.zeros_like(image, dtype=np.float32)

    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]
            dct_image[i:i+block_size, j:j+block_size] = dct(dct(block, axis=0,␣
 ↪norm='ortho'), axis=1, norm='ortho')
    return dct_image
```

[6]:
```python
def idct_transform(dct_image, block_size=8):
    height, width = dct_image.shape
    reconstructed_image = np.zeros_like(dct_image, dtype=np.float32)

    for i in range(0, height, block_size):
        for j in range(0, width, block_size):
            block = dct_image[i:i+block_size, j:j+block_size]
            reconstructed_image[i:i+block_size, j:j+block_size] =␣
 ↪idct(idct(block, axis=0, norm='ortho'), axis=1, norm='ortho')
    return np.clip(reconstructed_image, 0, 255).astype(np.uint8)
```

[7]:
```python
# Evaluate
image = load_image(r"/content/pexels-sulimansallehi-1704488.jpg")
dct_encoded = dct_transform(image)
dct_decoded = idct_transform(dct_encoded)
```

[8]:
```python
compression_ratio_dct = calculate_compression_ratio(image.size, dct_encoded.
 ↪size)
rmse_dct = calculate_rmse(image, dct_decoded)
```

[9]:
```python
print(f"DCT Compression Ratio: {compression_ratio_dct:.2f}")
print(f"DCT RMSE: {rmse_dct:.2f}")
```

```
DCT Compression Ratio: 1.00
DCT RMSE: 0.65
```

B. Huffman Encoding

[10]:
```python
class HuffmanNode:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None

    def __lt__(self, other):
```

```python
        return self.frequency < other.frequency

def build_huffman_tree(data):
    frequency = Counter(data)
    heap = [HuffmanNode(symbol, freq) for symbol, freq in frequency.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = HuffmanNode(None, node1.frequency + node2.frequency)
        merged.left = node1
        merged.right = node2
        heapq.heappush(heap, merged)

    return heap[0]

def huffman_encoding(data):
    root = build_huffman_tree(data)
    huffman_code = {}

    def generate_codes(node, code=""):
        if node.symbol is not None:
            huffman_code[node.symbol] = code
        else:
            generate_codes(node.left, code + "0")
            generate_codes(node.right, code + "1")

    generate_codes(root)
    encoded_data = "".join(huffman_code[symbol] for symbol in data)
    return encoded_data, huffman_code

def huffman_decoding(encoded_data, huffman_code):
    reverse_code = {v: k for k, v in huffman_code.items()}
    decoded_data = []
    buffer = ""
    for bit in encoded_data:
        buffer += bit
        if buffer in reverse_code:
            decoded_data.append(reverse_code[buffer])
            buffer = ""
    return np.array(decoded_data, dtype=np.uint8)
```

[11]:
```python
# usage and evaluation
flattened_image = image.flatten()
encoded_data, huffman_code = huffman_encoding(flattened_image)
```

```
decoded_image = huffman_decoding(encoded_data, huffman_code).reshape(image.
    ↪shape)
```

[12]:
```
compression_ratio_huffman = calculate_compression_ratio(len(flattened_image) *␣
    ↪8, len(encoded_data))
rmse_huffman = calculate_rmse(image, decoded_image)
```

[13]:
```
print(f"Huffman Compression Ratio: {compression_ratio_huffman:.2f}")
print(f"Huffman RMSE: {rmse_huffman:.2f}")
```

```
Huffman Compression Ratio: 1.17
Huffman RMSE: 0.00
```

C. LZW Encoding

[14]:
```
def lzw_encoding(data):
    dictionary = {chr(i): i for i in range(256)}
    current = ""
    encoded_data = []
    dict_size = 256

    for symbol in data:
        combined = current + symbol
        if combined in dictionary:
            current = combined
        else:
            encoded_data.append(dictionary[current])
            dictionary[combined] = dict_size
            dict_size += 1
            current = symbol

    if current:
        encoded_data.append(dictionary[current])

    return encoded_data, dictionary
```

[15]:
```
def lzw_decoding(encoded_data, dictionary):
    reverse_dict = {v: k for k, v in dictionary.items()}
    current = encoded_data.pop(0)
    decoded_data = [reverse_dict[current]]

    for code in encoded_data:
        if code in reverse_dict:
            entry = reverse_dict[code]
        else:
            entry = reverse_dict[current] + reverse_dict[current][0]

        decoded_data.append(entry)
```

```
        reverse_dict[len(reverse_dict)] = reverse_dict[current] + entry[0]
        current = code

    return "".join(decoded_data)
```

[16]:
```
# usage
flattened_image_str = ''.join(map(chr, flattened_image))
encoded_data, dictionary = lzw_encoding(flattened_image_str)
decoded_image_str = lzw_decoding(encoded_data, dictionary)
decoded_image = np.array(list(map(ord, decoded_image_str))).reshape(image.shape)
```

[17]:
```
compression_ratio_lzw = calculate_compression_ratio(len(flattened_image_str) *␣
 ↪8, len(encoded_data) * 8)
rmse_lzw = calculate_rmse(image, decoded_image)
```

[18]:
```
print(f"LZW Compression Ratio: {compression_ratio_lzw:.2f}")
print(f"LZW RMSE: {rmse_lzw:.2f}")
```

```
LZW Compression Ratio: 5.78
LZW RMSE: 0.00
```

D. Run-Length Encoding (RLE)

[19]:
```
def rle_encoding(data):
    encoded_data = []
    count = 1

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

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

[20]:
```
def rle_decoding(encoded_data):
    decoded_data = []
    for value, count in encoded_data:
        decoded_data.extend([value] * count)
    return np.array(decoded_data, dtype=np.uint8)
```

[21]:
```
# usage
encoded_data_rle = rle_encoding(flattened_image)
decoded_image_rle = rle_decoding(encoded_data_rle).reshape(image.shape)
```

[22]:
```
compression_ratio_rle = calculate_compression_ratio(len(flattened_image) * 8,␣
↪len(encoded_data_rle) * 16)
rmse_rle = calculate_rmse(image, decoded_image_rle)
```

[23]:
```
print(f"RLE Compression Ratio: {compression_ratio_rle:.2f}")
print(f"RLE RMSE: {rmse_rle:.2f}")
```

```
RLE Compression Ratio: 0.82
RLE RMSE: 0.00
```

E. Arithmetic Coding

[26]:
```
from collections import Counter

# Encoding function for Arithmetic Coding
def arithmetic_encoding(data):
    frequency = Counter(data)
    total_symbols = sum(frequency.values())
    prob = {symbol: freq / total_symbols for symbol, freq in frequency.items()}

    # Start with the full range [0, 1)
    low, high = 0.0, 1.0

    for symbol in data:
        range_ = high - low
        high = low + range_ * sum(prob[s] for s in prob if s <= symbol)
        low = low + range_ * sum(prob[s] for s in prob if s < symbol)

    # The encoded value is any number within the final interval [low, high)
    encoded_value = (low + high) / 2
    return encoded_value, prob
```

[27]:
```
# Decoding function for Arithmetic Coding
def arithmetic_decoding(encoded_value, length, prob):
    low, high = 0.0, 1.0
    decoded_data = []

    for _ in range(length):
        range_ = high - low
        for symbol, p in prob.items():
            new_high = low + range_ * sum(prob[s] for s in prob if s <= symbol)
            new_low = low + range_ * sum(prob[s] for s in prob if s < symbol)

            if new_low <= encoded_value < new_high:
                decoded_data.append(symbol)
                low, high = new_low, new_high
                break
```

```
    return np.array(decoded_data, dtype=np.uint8)
```

[ ]:
```
# Example Usage and Evaluation
flattened_image = image.flatten()
encoded_value, probabilities = arithmetic_encoding(flattened_image)
decoded_image = arithmetic_decoding(encoded_value, len(flattened_image),␣
↪probabilities).reshape(image.shape)
```

[ ]:
```
# Compression Ratio (estimative, since actual encoding depends on binary␣
↪representation of `encoded_value`)
compressed_size_estimate = len(str(encoded_value)) * 8
compression_ratio_arithmetic = calculate_compression_ratio(flattened_image.size␣
↪* 8, compressed_size_estimate)
rmse_arithmetic = calculate_rmse(image, decoded_image)
```

[ ]:
```
print(f"Arithmetic Compression Ratio: {compression_ratio_arithmetic:.2f}")
print(f"Arithmetic RMSE: {rmse_arithmetic:.2f}")
```

[ ]: