CC-402: Text, Image and Video Analytics

Unit 4: Image Compression Assignment

Name: Avani Brahmbhatt

Roll Number: 01

Course: Data Science

Semester: 07

Part B: Coding:

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. LZW Encoding

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.

A. Transform Coding (using DCT for forward transform)

In [1]:

```
import numpy as np
import cv2
from scipy.fftpack import dct, idct
from skimage.metrics import mean_squared_error

def dct_transform(image, block_size=8):
    h, w = image.shape
    dct_blocks = np.zeros_like(image, dtype=np.float32)

    # Apply DCT to each block
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = image[i:i+block_size, j:j+block_size]
            dct_block = dct(dct(block.T, norm='ortho').T, norm='ortho')
            dct_blocks[i:i+block_size, j:j+block_size] = dct_block
    return dct_blocks

def idct_transform(dct_blocks, block_size=8):
    h, w = dct_blocks.shape
    reconstructed_image = np.zeros_like(dct_blocks, dtype=np.float32)

    # Apply inverse DCT to each block
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = dct_blocks[i:i+block_size, j:j+block_size]
            idct_block = idct(idct(block.T, norm='ortho').T, norm='orth
            reconstructed_image[i:i+block_size, j:j+block_size] = idct_
    return np.clip(reconstructed_image, 0, 255).astype(np.uint8)

# Encode
image = cv2.imread('cato.jpg', cv2.IMREAD_GRAYSCALE)
dct_encoded = dct_transform(image)

# Decode
dct_decoded = idct_transform(dct_encoded)

# Compression Ratio and RMSE
compression_ratio = image.size / dct_encoded.size
rmse = np.sqrt(mean_squared_error(image, dct_decoded))


print('Compression Ratio',compression_ratio)
print('RMSE:',rmse)
```

```
C:\ProgramData\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarn
ing: A NumPy version >=1.16.5 and <1.23.0 is required for this version of
SciPy (detected version 1.26.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"

Compression Ratio 1.0
RMSE: 0.6609112467430145
```

B. Huffman Encoding

In [2]:

```python
import cv2
from collections import Counter
import heapq

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

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

def build_huffman_tree(frequency_dict):
    heap = [HuffmanNode(symbol, freq) for symbol, freq in frequency_dic
    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_code_map(node, path='', code_map={}):
    if node.symbol is not None:
        code_map[node.symbol] = path
    else:
        if node.left:
            huffman_code_map(node.left, path + '0', code_map)
        if node.right:
            huffman_code_map(node.right, path + '1', code_map)
    return code_map

def huffman_encode(image):
    # Flatten the image and calculate frequency
    symbols = image.flatten()
    frequency_dict = dict(Counter(symbols))
    huffman_tree_root = build_huffman_tree(frequency_dict)
    code_map = huffman_code_map(huffman_tree_root)

    # Encode
    encoded_image = ''.join(code_map[symbol] for symbol in symbols)
    return encoded_image, code_map

def huffman_decode(encoded_image, code_map, shape):
    inverse_code_map = {v: k for k, v in code_map.items()}
    current_code = ''
    decoded_image = []

    for bit in encoded_image:
        current_code += bit
        if current_code in inverse_code_map:
            decoded_image.append(inverse_code_map[current_code])
            current_code = ''

    return np.array(decoded_image).reshape(shape)
```

```

# Encode
encoded_image, code_map = huffman_encode(image)

# Decode
decoded_image = huffman_decode(encoded_image, code_map, image.shape)

# Compression Ratio and RMSE
compression_ratio = len(encoded_image) / (image.size * 8)  # assuming 8
rmse = np.sqrt(mean_squared_error(image, decoded_image))


print('Compression Ratio',compression_ratio)
print('RMSE:',rmse)
```

```
Compression Ratio 0.946908333295 91
RMSE: 0.0
```

C. LZW Encoding

In [3]:

```
import numpy as np
import cv2
def lzw_encode(image):
    pixels = image.flatten()
    dictionary = {bytes([i]): i for i in range(256)}
    dict_size = 256
    p = bytes([pixels[0]])
    encoded = []

    for c in pixels[1:]:
        pc = p + bytes([c])
        if pc in dictionary:
            p = pc
        else:
            encoded.append(dictionary[p])
            dictionary[pc] = dict_size
            dict_size += 1
            p = bytes([c])

    encoded.append(dictionary[p])
    return encoded

def lzw_decode(encoded):
    dictionary = {i: bytes([i]) for i in range(256)}
    dict_size = 256
    p = bytes([encoded[0]])
    decoded = [p]

    for k in encoded[1:]:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = p + p[:1]
        else:
            raise ValueError("Bad encoded k")

        decoded.append(entry)
        dictionary[dict_size] = p + entry[:1]
        dict_size += 1
        p = entry

    return np.frombuffer(b''.join(decoded), dtype=np.uint8)

# Encode
encoded_image = lzw_encode(image)

# Decode
decoded_image = lzw_decode(encoded_image).reshape(image.shape)

# Compression Ratio and RMSE
compression_ratio = len(encoded_image) / image.size
rmse = np.sqrt(mean_squared_error(image, decoded_image))


print('Compression Ratio',compression_ratio)
print('RMSE:',rmse)
```

```
Compression Ratio 0.1110135193124481
RMSE: 0.0
```

D. Run-Length Encoding

In [4]:

```
import numpy as np
import cv2
import numpy as np
from skimage.metrics import mean_squared_error

def rle_encode(image):
    pixels = image.flatten()
    encoded = []
    prev_pixel = pixels[0]
    count = 1

    for pixel in pixels[1:]:
        if pixel == prev_pixel:
            count += 1
        else:
            encoded.append((prev_pixel, count))
            prev_pixel = pixel
            count = 1
    # Append the last run
    encoded.append((prev_pixel, count))
    return encoded

def rle_decode(encoded, shape):
    decoded_pixels = []
    for pixel, count in encoded:
        decoded_pixels.extend([pixel] * count)
    return np.array(decoded_pixels).reshape(shape)

# Encode
image = cv2.imread('cato.jpg', cv2.IMREAD_GRAYSCALE)
encoded_image = rle_encode(image)

# Decode
decoded_image = rle_decode(encoded_image, image.shape)

# Compression Ratio and RMSE
compression_ratio = len(encoded_image) / image.size  # since each run u
rmse = np.sqrt(mean_squared_error(image, decoded_image))

print("Compression Ratio (RLE):", compression_ratio)
print("RMSE (RLE):", rmse)

```

```
Compression Ratio (RLE): 0.35363026521683594
RMSE (RLE): 0.0
```

E. Arithmetic Coding

In [5]:

```
from collections import Counter
from itertools import accumulate

# Step 1: Calculate frequency/probability of each pixel value
def calculate_probabilities(image):
    pixels = image.flatten()
    total_pixels = len(pixels)
    frequency = Counter(pixels)
    probabilities = {k: v / total_pixels for k, v in frequency.items()}
    return probabilities

# Step 2: Generate cumulative probability intervals for each symbol
def generate_intervals(probabilities):
    keys = list(probabilities.keys())
    values = list(probabilities.values())
    cumulative_probs = list(accumulate(values))
    intervals = {keys[i]: (cumulative_probs[i] - values[i], cumulative_
    return intervals

# Step 3: Encode using the probability intervals
def arithmetic_encode(image, intervals):
    pixels = image.flatten()
    low, high = 0.0, 1.0

    for pixel in pixels:
        range_width = high - low
        low = low + range_width * intervals[pixel][0]
        high = low + range_width * (intervals[pixel][1] - intervals[pi

    return (low + high) / 2  # Final encoded value

# Step 4: Decode the encoded message
def arithmetic_decode(encoded_value, intervals, num_pixels):
    decoded_pixels = []
    for _ in range(num_pixels):
        for pixel, (low, high) in intervals.items():
            if low <= encoded_value < high:
                decoded_pixels.append(pixel)
                encoded_value = (encoded_value - low) / (high - low)
                break
    return np.array(decoded_pixels)

# Encode
probabilities = calculate_probabilities(image)
intervals = generate_intervals(probabilities)
encoded_value = arithmetic_encode(image, intervals)

# Decode
decoded_image = arithmetic_decode(encoded_value, intervals, image.size)

# Compression Ratio and RMSE
compression_ratio = image.size / len(probabilities)  # Assuming a singl
rmse = np.sqrt(mean_squared_error(image, decoded_image))

print("Compression Ratio (Arithmetic Coding):", compression_ratio)
print("RMSE (Arithmetic Coding):", rmse)
```

```
Compression Ratio (Arithmetic Coding): 64131.072
RMSE (Arithmetic Coding): 74.79429391753378
```

Github Link:

[https://github.com/Avani-Brahmbhatt/CC-402-Text-Image-and-Video-Analytics-Assignment.git (https://github.com/Avani-Brahmbhatt/CC-402-Text-Image-and-Video-Analytics-Assignment.git)](https://github.com/Avani-Brahmbhatt/CC-402-Text-Image-and-Video-Analytics-Assignment.git)

In [ ]:
```

```