DIGITAL IMAGE PROCESSING
ELL715
ASSIGNMENT 3

08 October 2023

Group Members

1. Bhavik Shankla (2020MT60873)
2. Ritika Soni (2020MT10838)
3. Sai Kiran Gunnala (2020MT60889)
4. Sai Niketh Varanasi (2020MT60895)
5. Yash Pravin Shirke (2020MT60986)

Question 1

The pixels in an image are scanned from left to the right and from the top to the bottom. Each new pixel is predicted by the average of the pixel above and the one to the left. Let f and F represent the original and the predicted values, and e = f − F is the prediction error. The prediction error is quantized to “0”, “B”, or “-B” according to:

$$\hat{e} = \begin{cases} -B & e < -T \\ 0 & -T \le e \le T \\ B & e > T \end{cases}$$

0	0	1	5	6
0	0	1	5	6
2	2	4	7	8
3	3	7	4	2
6	6	5	1	0

Find the optimum weights while predicting the image such that mean square error is minimum.
Repeat the process if you use all nearest neighbor to predict the pixel value.
Repeat the process on any image of your choice

Solution

2-Neighbour Approach

In this approach, we utilize previously predicted pixel values from the top and left neighbors (‘$F[i][j-1]$‘ and ‘$F[i-1][j]$‘) while considering weights ‘$w1$‘ and ‘$w2$‘ to forecast the new pixel value at position ‘(i, j)‘. The objective is to minimize the mean squared error (MSE) between the original image and the image predicted using these weighted combinations. The constraint ‘$w1 + w2 = 1$‘ ensures weight normalization, resulting in a linear combination of the top and left neighbors.

4-Neighbour Approach

In this approach, we incorporate information from four neighboring pixels (above, below, left, and right) in addition to the original pixel to predict the new pixel value. We adjust the influence of these neighbors using weights ‘w_above‘, ‘w_below‘, ‘w_left‘, and ‘w_right‘. The optimization seeks to determine optimal weights that minimize the MSE between the original image and the predicted image.

Both approaches employ a quantization function, ‘quantize_error‘, to map prediction errors to three levels (-1, 0, 1). The optimization process aims to find weights that minimize the quantized MSE.

Optimization Process

We employ the SciPy ‘minimize‘ function to find the optimal weights for both approaches. For the 2-neighbour approach, we optimize ‘$w1$‘ and ‘$w2$‘, and for the 4-neighbour approach, we optimize ‘w_above‘, ‘w_below‘, ‘w_left‘, and ‘w_right‘. The optimization is subject to the constraint that the sum of these weights equals 1, ensuring that the prediction is a linear combination of neighboring pixels.

Python Code

```python
from PIL import Image
import numpy as np

image_path = '/content/harry.jpg'
image = Image.open(image_path)

# Converting the image to grayscale
if image.mode != 'L':
    image = image.convert('L')

# Converting the grayscale image to a NumPy matrix
image_matrix = np.array(image)

print("Image Matrix Shape:", image_matrix.shape)


import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

f = image_matrix

# Define the objective function to calculate MSE and then quantize error
def calculate_mse_and_quantize(weights, original_image):
    w1, w2 = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = (int)(original_image[i][j])
            elif i == 0:
                # F[i][j] = w1*F[i][j-1]
                F[i][j] = (int)(original_image[i][j])
            elif j == 0:
                # F[i][j] = w2*F[i-1][j]
                F[i][j] = (int)(original_image[i][j])
            else:
                F[i][j] = (int)(w1 * original_image[i][j-1] + w2 * original_image[i-1][j])

    prediction_errors = original_image - F
```

```
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
    mse = np.mean(quantized_errors**2)

    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective (minimize quantized MSE)
def objective(weights):
    return calculate_mse_and_quantize(weights, image_matrix)[0]  # Use image_matrix instead of predefined matrix f

# Initial guess for weights (w1, w2)
initial_weights = [0.5, 0.5]

# Define bounds for weights
bounds = [(0, 1), (0, 1)]

# Add x1 + x2 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] - 1},)

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds, constraints=constraints)

# Extracting the optimized weights
optimized_weights = result.x
w1_optimized, w2_optimized = optimized_weights

# Print the optimized weights and quantized MSE
print("Optimized Weights (w1, w2):", optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix = calculate_mse_and_quantize(optimized_weights, image_matrix)

predicted_image = Image.fromarray(predicted_matrix)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image, cmap='gray')
plt.title("Predicted Image")
```

```python

plt.show()


# In the below code we tried to experiment it using the updated values to predict the new pixel values

# Define the objective function to calculate MSE and then quantize error
def calculate_mse_and_quantize(weights, original_image):
    w1, w2 = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = (int)(original_image[i][j])
            elif i == 0:
                # F[i][j] = w1*F[i][j-1]
                F[i][j] = (int)(original_image[i][j])
            elif j == 0:
                # F[i][j] = w2*F[i-1][j]
                F[i][j] = (int)(original_image[i][j])
            else:
                F[i][j] = (int)(w1 * F[i][j-1] + w2 * F[i-1][j])

    prediction_errors = original_image - F
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
    mse = np.mean(quantized_errors**2)

    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective
def objective(weights):
    return calculate_mse_and_quantize(weights, image_matrix)[0]  # Use image_matrix instead of predefined matrix f

# Initial guess for weights (w1, w2)
initial_weights = [0.5, 0.5]

# Define bounds for weights
bounds = [(0, 1), (0, 1)]  # weights between 0 and 1

# Add x1 + x2 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] - 1},)

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds, constraints=constraints)
```

```python

# Extract the optimized weights
optimized_weights = result.x
w1_optimized, w2_optimized = optimized_weights
# print("optimized weights:",result.x)

# Print the optimized weights and quantized MSE
print("Optimized Weights (w1, w2):", optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix = calculate_mse_and_quantize(optimized_weights,
    image_matrix)

predicted_image = Image.fromarray(predicted_matrix)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image, cmap='gray')
plt.title("Predicted Image")

plt.show()


# 4 neighbours to predict the pixel value

def calculate_mse_and_quantize_neighbors(weights, original_image):
    w_above, w_below, w_left, w_right = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = original_image[i][j]
            elif i == 0:
                # F[i][j] = (w_left * F[i][j - 1] + original_image[i][j]) /
                    (w_left + 1)
                F[i][j] = original_image[i][j]
            elif j == 0:
                # F[i][j] = (w_above * F[i - 1][j] + original_image[i][j]) /
                    (w_above + 1)
                F[i][j] = original_image[i][j]
            else:
                neighbors = [
                    w_above * original_image[i - 1][j],         # Above
                    w_below * original_image[i + 1][j] if i + 1 < M else
                        original_image[i][j],  # Below (with boundary check)
                    w_left * original_image[i][j - 1],          # Left
                    w_right * original_image[i][j + 1] if j + 1 < N else
                        original_image[i][j]  # Right (with boundary check)
                ]
                F[i][j] = sum(neighbors)

    prediction_errors = original_image - F
```

```
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
    mse = np.mean(quantized_errors**2)
    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective (minimize quantized MSE)
def objective(weights):
    return calculate_mse_and_quantize_neighbors(weights, image_matrix)[0]  # Use image_matrix instead of predefined matrix f

# Initial guess for weights (w_above, w_below, w_left, w_right)
initial_weights = [0.25, 0.25, 0.25, 0.25]

# Add x1 + x2 + x3 + x4 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] + weights[2] + weights[3] - 1},)

# Define bounds for weights
bounds = [(0, 1), (0, 1), (0, 1), (0, 1)]  # weights between 0 and 1

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds, constraints=constraints)

# Extract the optimized weights
optimized_weights = result.x
w_above_opt, w_below_opt, w_left_opt, w_right_opt = optimized_weights

# Print the optimized weights and quantized MSE
print("Optimized Weights (w_above, w_below, w_left, w_right):", optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix1 = calculate_mse_and_quantize_neighbors(optimized_weights, image_matrix)

predicted_image1 = Image.fromarray(predicted_matrix1)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image1, cmap='gray')
plt.title("Predicted Image")
```

```

plt.show()
```

Results

We report the optimized weights and the quantized MSE for each approach, demonstrating the effectiveness of the optimization process in minimizing prediction errors. Finally, both the original and predicted images are displayed side by side for visual comparison.

In summary, our code showcases two distinct approaches for pixel value prediction in an image, each with its own consideration of neighboring pixels. The optimization process determines optimal weights to minimize the quantized MSE between the original and predicted images, resulting in varying levels of prediction accuracy for the two approaches.

Question 2

Take a black & white typeset document, encode the document using Runlength encoding and than G3 fax encoder, compare the results.

Solution

Python Code

```python
import cv2
from PIL import Image
import numpy as np

# Load the input image
input_image = Image.open('doc.jpg')

# Convert the input image to binary (black and white)
binary_image = input_image.convert('1')

# Save the binary image
binary_image.save('result.jpg')

# Get the size of the binary image
image_size = binary_image.size

# Function to perform run-length encoding on a binary image
def rle_encode(image_array):
    shape = image_array.shape
    image_array = image_array.flatten()

    if len(image_array) == 0:
        return "0 0"

    encoded_data = f"{shape[0]} {shape[1]} "
    current_pixel = image_array[0]
    current_length = 1

    for i in range(1, len(image_array)):
        if image_array[i] != current_pixel:
            if current_pixel == True:
                encoded_data += f"{1} {current_length} "
            else:
                encoded_data += f"{0} {current_length} "
            current_pixel = image_array[i]
            current_length = 1
        else:
            current_length += 1

    if current_pixel == True:
        encoded_data += f"{1} {current_length}"
    else:
        encoded_data += f"{0} {current_length}"
    return encoded_data

# Function to decode a run-length encoded string back to an image
def rle_decode(encoded_data):
    arr = [int(x) for x in encoded_data.split()]
    rows = arr[0]
    cols = arr[1]
    data = arr[2:]

    decoded_data = []
```

```python

    for i in range(0, len(data), 2):
        pixel_value = data[i]
        run_length = data[i + 1]
        decoded_data.extend([pixel_value] * run_length)

    decoded_data = np.array(decoded_data, dtype=np.uint8)
    decoded_image = Image.fromarray(decoded_data.reshape(rows, cols))
    return decoded_image

# Function to encode the binary image using run-length encoding
def rle_encode_image(binary_image):
    image_array = np.array(binary_image)
    encoded_image_data = rle_encode(image_array)
    return encoded_image_data

# Load the input image
input_image = Image.open('doc.jpg')

# Convert the input image to binary (black and white)
binary_image = input_image.convert('1')

# Encode the binary image using run-length encoding
rle_encoded_data = rle_encode_image(binary_image)
# print('RLE encoded image data:', rle_encoded_data)

# Save the encoded data to a text file
with open("encoded_data.txt", "w") as encoded_file:
    encoded_file.write(rle_encoded_data)

# Class to perform G3Fax encoding on a binary image
class G3FaxEncoder:

    def __init__(self, image):
        self.image = image
        self.width = image.width
        self.height = image.height

    # Encode the binary image using G3Fax encoding
    def encode(self):
        bitstream = bytearray()

        # Start of page (SOP) marker
        bitstream.extend([0xFF, 0x00])

        image_data = list(self.image.getdata())

        for y in range(self.height):
            line = image_data[y * self.width: (y + 1) * self.width]
            run_length = 0

            for pixel in line:
                if pixel == 0:  # Black pixel
                    run_length += 1
                else:  # White pixel
                    if run_length > 0:
                        bitstream.extend(self.encode_run_length(run_length))
                    run_length = 0

            # End of line (EOL) marker
            bitstream.extend([0x00, 0x00])

        bitstream.extend([0x01, 0x00])
```

```python
        return bitstream

    # Encode run length for G3Fax
    def encode_run_length(self, run_length):
        encoded = []

        while run_length >= 0x80:
            encoded.append(0x80 | (run_length & 0x7F))
            run_length >>= 7
        encoded.append(run_length)
        return encoded

# Create an instance of the G3FaxEncoder
g3fax_encoder = G3FaxEncoder(binary_image)

# Encode the image using G3Fax and save it to a binary file
g3fax_bitstream = g3fax_encoder.encode()

with open('g3fax_bitstream.bin', 'wb') as f:
    f.write(g3fax_bitstream)
```

In the course of our analysis, we have examined two distinct compression methods, namely Run-Length Encoding (RLE) and G3 Fax Encoding. These methods were applied to an original image with a file size of 414KB to evaluate their respective compression efficiencies.

The results of our assessment are as follows:

1. Run-Length Encoding (RLE):
   - The RLE-encoded data produced an output size of 320KB.
   - Compression Ratio: 414KB/320KB = 1.29
2. G3 Fax Encoding:
   - The G3 Fax-encoded data yielded an output size of 44KB.
   - Compression Ratio: 414KB/44KB = 9.41

As higher the compression ratio, the smaller the file. So from the above observations, it becomes evident that the RLE method achieved a relatively lower compression ratio. This outcome is primarily attributed to the nature of RLE, which is effective for eliminating consecutive duplicate pixels but may not perform optimally for complex images.

In contrast, the G3 Fax Encoding method demonstrated significantly superior compression capabilities with a compression ratio. This remarkable result is a testament to the efficiency of G3 Fax Encoding in handling bi-level (black and white) images, particularly in scenarios such as scanned text documents.

In conclusion, while Run-Length Encoding (RLE) offers simplicity and ease of encoding and decoding, it falls short of achieving substantial compression for certain types of images. On the other hand, G3 Fax Encoding, although more complex to implement, excels in compression, making it a preferred choice for scenarios where efficient data compression is crucial.

Question 3

Compress the image used in Question 2 using

a) Huffman coding

b) DCT coding

c) KL transform based coding

d) use Haar wavelet and compress it

Compare the results in terms of compression

Solution

Python Code

```
import cv2
import numpy as np
from scipy.fftpack import dct, idct
import matplotlib.pyplot as plt
import pywt
import huffman

original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

dct_image = dct(dct(original_image.T, norm='ortho').T, norm='ortho')

quantization_factor = 0.001
quantized_dct_image = np.round(dct_image / quantization_factor)

encoded_data = quantized_dct_image.flatten().astype(np.int16)
encoded_data.tofile('encoded_image.bin')

decoded_dct_image = idct(idct(quantized_dct_image.T, norm='ortho').T,
    norm='ortho').astype(np.uint8)

# Display the original and decoded images
plt.subplot(1, 3, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(quantized_dct_image, cmap='gray')
plt.title('Encoded Image')

plt.subplot(1, 3, 3)
plt.imshow(decoded_dct_image, cmap='gray')
plt.title('Decoded Image')

plt.show()


# Load an image using OpenCV
original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

# Perform Haar wavelet transform
coeffs = pywt.dwt2(original_image, 'haar')

# Get the approximation and details coefficients
cA, (cH, cV, cD) = coeffs

```

```python
# Display the coefficients or perform further processing as needed
cv2.imshow('Approximation (cA)', cA)
cv2.imshow('Horizontal Detail (cH)', cH)
cv2.imshow('Vertical Detail (cV)', cV)
cv2.imshow('Diagonal Detail (cD)', cD)

cv2.waitKey(0)
cv2.destroyAllWindows()


# Load the image using OpenCV
original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

# Calculate pixel frequencies
pixel_frequencies = {}
for row in original_image:
    for pixel_value in row:
        if pixel_value in pixel_frequencies:
            pixel_frequencies[pixel_value] += 1
        else:
            pixel_frequencies[pixel_value] = 1

# Build the Huffman tree
huff_tree = huffman.build_tree(pixel_frequencies)

# Generate Huffman codes
huff_codes = huffman.get_codes(huff_tree)

# Encode the image using Huffman codes
encoded_image = []
for row in original_image:
    encoded_row = [huff_codes[pixel] for pixel in row]
    encoded_image.append(encoded_row)

with open('encoded_image.txt', 'w') as f:
    for row in encoded_image:
        f.write(' '.join(row) + '\n')
```

KL Transform based coding MATLAB Code

```matlab
clc;
close all;
clear all;
I=imread('cameraman.tif');
I=im2double(I);
m=1;
for i=1:8:256
    for j=1:8:256
        for x=0:7
            for y=0:7
                img(x+1,y+1)=I(i+x,j+y);
            end
        end

        k=0;
        for l=1:8
            img_expect{k+1}=img(:,l)*img(:,l)';
            k=k+1;
        end

        imgexp=zeros(8:8);
        for l=1:8
```

```
            imgexp=imgexp+(1/8)*img_expect{l};
            %expectation of E[xx']
        end

        img_mean=zeros(8,1);
        for l=1:8
            img_mean=img_mean+(1/8)*img(:,l);
        end

        img_mean_trans=img_mean*img_mean';
        img_covariance=imgexp - img_mean_trans;
        [v{m},d{m}]=eig(img_covariance);
        temp=v{m};
        m=m+1;

        for l=1:8
            v{m-1}(:,l)=temp(:,8-(l-1));
        end

        for l=1:8
            trans_img1(:,l)=v{m-1}*img(:,l);
        end

        for x=0:7
            for y=0:7
                transformed_img(i+x,j+y)=trans_img1(x+1,y+1);
            end
        end

        mask=[1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1 ];

        trans_img=trans_img1.*mask;
        for l=1:8
            inv_trans_img(:,l)=v{m-1}'*trans_img(:,l);
        end

        for x=0:7
            for y=0:7
                inv_transformed_img(i+x,j+y)=inv_trans_img(x+1,y+1);
            end
        end

    end
end
imshow(transformed_img);

figure
imshow(inv_transformed_img);
```

```python
from PIL import Image
import numpy as np

image_path = '/content/harry.jpg'
image = Image.open(image_path)

# Converting the image to grayscale
if image.mode != 'L':
    image = image.convert('L')

# Converting the grayscale image to a NumPy matrix
image_matrix = np.array(image)

print("Image Matrix Shape:", image_matrix.shape)
```

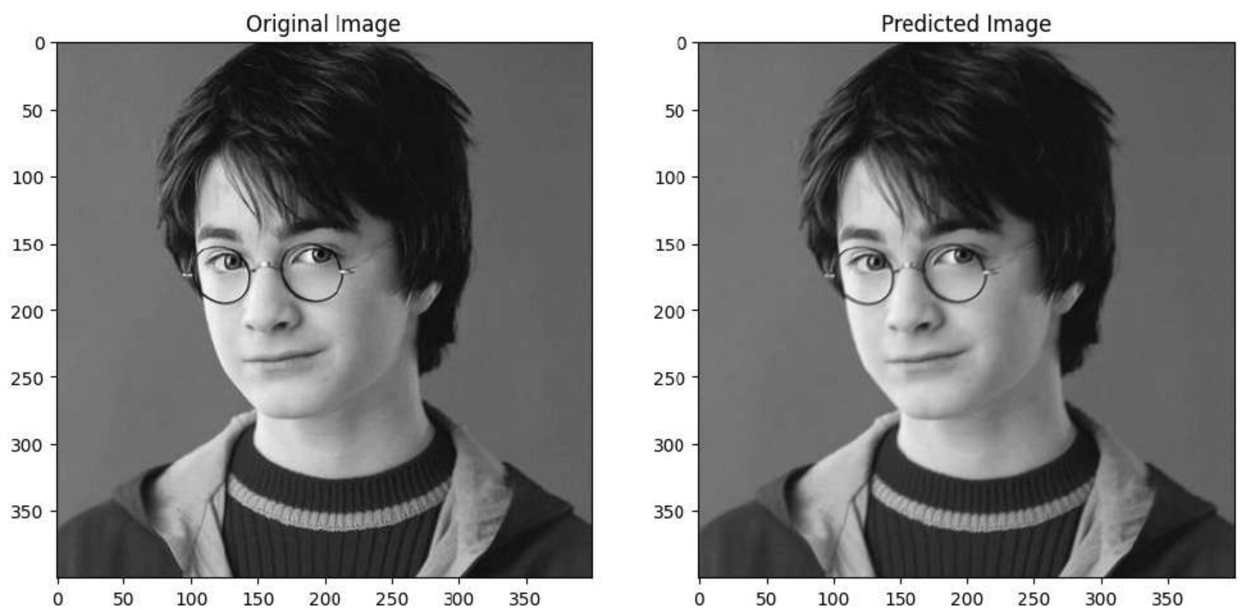
```
Image Matrix Shape: (400, 400)
```

```python
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

f = image_matrix

# Define the objective function to calculate MSE and then quantize
error
def calculate_mse_and_quantize(weights, original_image):
    w1, w2 = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = (int)(original_image[i][j])
            elif i == 0:
                # F[i][j] = w1*F[i][j-1]
                F[i][j] = (int)(original_image[i][j])
            elif j == 0:
                # F[i][j] = w2*F[i-1][j]
                F[i][j] = (int)(original_image[i][j])
            else:
                F[i][j] = (int)(w1 * original_image[i][j-1] + w2 *
original_image[i-1][j])

    prediction_errors = original_image - F
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
```

```
    mse = np.mean(quantized_errors**2)

    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective (minimize quantized MSE)
def objective(weights):
    return calculate_mse_and_quantize(weights, image_matrix)[0]  # Use
image_matrix instead of predefined matrix f

# Initial guess for weights (w1, w2)
initial_weights = [0.5, 0.5]

# Define bounds for weights
bounds = [(0, 1), (0, 1)]

# Add x1 + x2 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] +
weights[1] - 1},)

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds,
constraints=constraints)

# Extracting the optimized weights
optimized_weights = result.x
w1_optimized, w2_optimized = optimized_weights

# Print the optimized weights and quantized MSE
print("Optimized Weights (w1, w2):", optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix =
calculate_mse_and_quantize(optimized_weights, image_matrix)

predicted_image = Image.fromarray(predicted_matrix)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
```

```
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image, cmap='gray')
plt.title("Predicted Image")

plt.show()
```

```
Optimized Weights (w1, w2): [0.5 0.5]
Optimized Quantized MSE: 0.346225
```

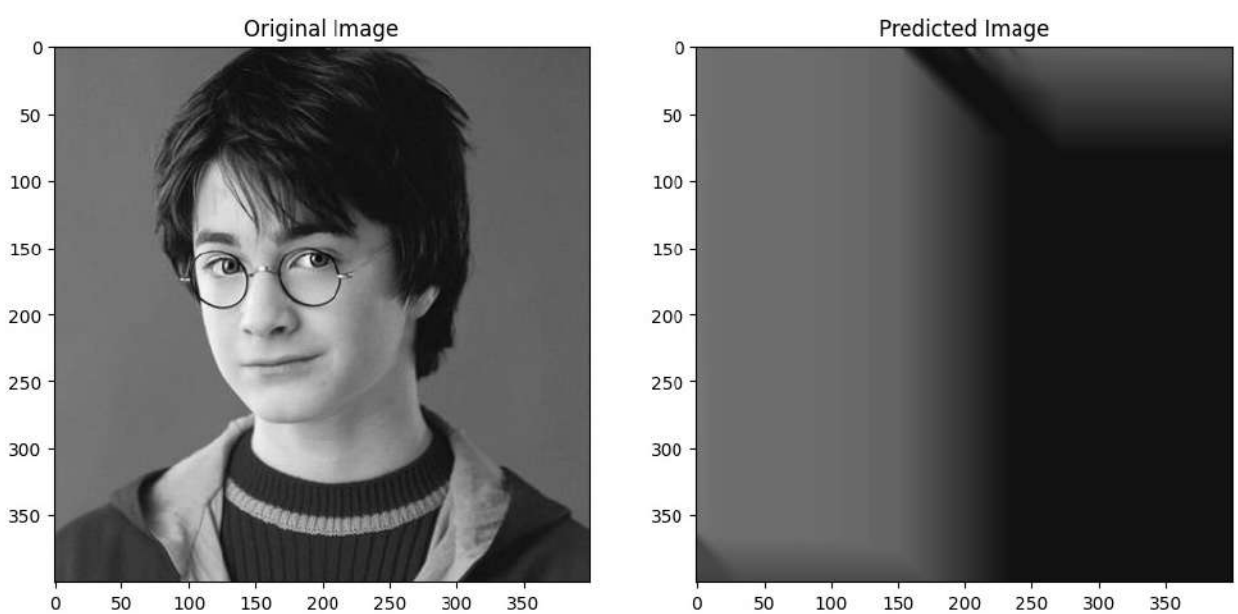



```
# In the below code we tried to experiment it using the updated values
to predict the new pixel values

# Define the objective function to calculate MSE and then quantize
error
def calculate_mse_and_quantize(weights, original_image):
    w1, w2 = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = (int)(original_image[i][j])
            elif i == 0:
                # F[i][j] = w1*F[i][j-1]
```

```
                F[i][j] = (int)(original_image[i][j])
            elif j == 0:
                # F[i][j] = w2*F[i-1][j]
                F[i][j] = (int)(original_image[i][j])
            else:
                F[i][j] = (int)(w1 * F[i][j-1] + w2 * F[i-1][j])

    prediction_errors = original_image - F
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
    mse = np.mean(quantized_errors**2)

    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective
def objective(weights):
    return calculate_mse_and_quantize(weights, image_matrix)[0]  # Use
image_matrix instead of predefined matrix f

# Initial guess for weights (w1, w2)
initial_weights = [0.5, 0.5]

# Define bounds for weights
bounds = [(0, 1), (0, 1)]  # weights between 0 and 1

# Add x1 + x2 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] +
weights[1] - 1},)

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds,
constraints=constraints)

# Extract the optimized weights
optimized_weights = result.x
w1_optimized, w2_optimized = optimized_weights
# print("optimized weights:",result.x)
```

```
# Print the optimized weights and quantized MSE
print("Optimized Weights (w1, w2):", optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix =
calculate_mse_and_quantize(optimized_weights, image_matrix)

predicted_image = Image.fromarray(predicted_matrix)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image, cmap='gray')
plt.title("Predicted Image")

plt.show()
```

```
Optimized Weights (w1, w2): [0.5 0.5]
Optimized Quantized MSE: 0.9601125
```

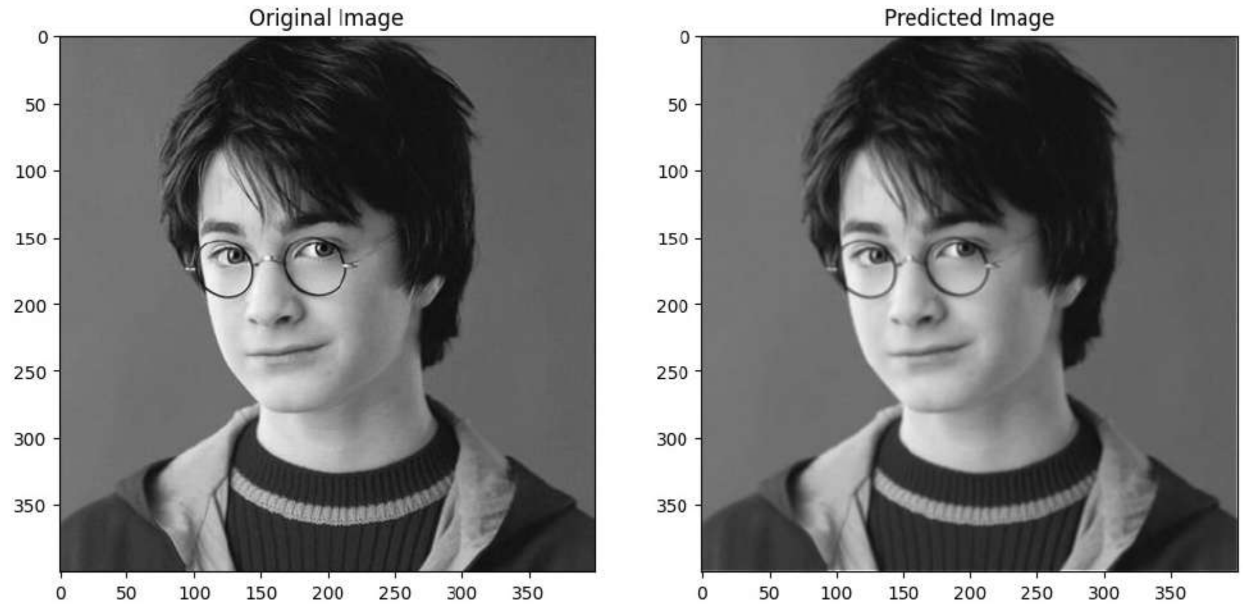



```
# 4 neighbours to predict the pixel value

def calculate_mse_and_quantize_neighbors(weights, original_image):
```

```
    w_above, w_below, w_left, w_right = weights
    M, N = original_image.shape
    F = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            if i == 0 and j == 0:
                F[i][j] = original_image[i][j]
            elif i == 0:
                # F[i][j] = (w_left * F[i][j - 1] + original_image[i]
[j]) / (w_left + 1)
                F[i][j] = original_image[i][j]
            elif j == 0:
                # F[i][j] = (w_above * F[i - 1][j] + original_image[i]
[j]) / (w_above + 1)
                F[i][j] = original_image[i][j]
            else:
                neighbors = [
                    w_above * original_image[i - 1][j],       # Above
                    w_below * original_image[i + 1][j] if i + 1 < M
else original_image[i][j],  # Below (with boundary check)
                    w_left * original_image[i][j - 1],        # Left
                    w_right * original_image[i][j + 1] if j + 1 < N
else original_image[i][j]  # Right (with boundary check)
                ]
                F[i][j] = sum(neighbors)

    prediction_errors = original_image - F
    flatten_array = prediction_errors.flatten()
    quantized_errors = np.zeros(flatten_array.size)
    # print(flatten_array.size)
    for i in range(flatten_array.size):
      quantized_errors[i] = quantize_error(flatten_array[i])
    mse = np.mean(quantized_errors**2)
    return mse, F

# Define the quantization function for prediction errors
def quantize_error(error):
    if error < -1:
        return -1
    elif error >= -1 and error <= 1:
        return 0
    else:
        return 1

# Define the optimization objective (minimize quantized MSE)
def objective(weights):
    return calculate_mse_and_quantize_neighbors(weights, image_matrix)
[0]  # Use image_matrix instead of predefined matrix f
```

```python
# Initial guess for weights (w_above, w_below, w_left, w_right)
initial_weights = [0.25, 0.25, 0.25, 0.25]

# Add x1 + x2 + x3 + x4 = 1 as an equality constraint
constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] +
weights[1] + weights[2] + weights[3] - 1},)

# Define bounds for weights
bounds = [(0, 1), (0, 1), (0, 1), (0, 1)]  #  weights between 0 and 1

# Optimize weights to minimize quantized MSE
result = minimize(objective, initial_weights, bounds=bounds,
constraints=constraints)

# Extract the optimized weights
optimized_weights = result.x
w_above_opt, w_below_opt, w_left_opt, w_right_opt = optimized_weights

# Print the optimized weights and quantized MSE
print("Optimized Weights (w_above, w_below, w_left, w_right):",
optimized_weights)
print("Optimized Quantized MSE:", result.fun)

# Calculating the predicted image using the optimized weights
optimal_mse, predicted_matrix1 =
calculate_mse_and_quantize_neighbors(optimized_weights, image_matrix)

predicted_image1 = Image.fromarray(predicted_matrix1)

# Display both the original and predicted images side by side
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(predicted_image1, cmap='gray')
plt.title("Predicted Image")

plt.show()
```

```
Optimized Weights (w_above, w_below, w_left, w_right): [0.25 0.25 0.25
0.25]
Optimized Quantized MSE: 0.30706875
```


Original Image
Predicted Image
0
50
100
150
200
250
300
350

q2-a4

October 8, 2023

[5]:
```
import cv2
from PIL import Image
import numpy as np
from google.colab.patches import cv2_imshow
```

[6]:
```
image = Image.open('doc.jpg')
image_binary = image.convert('1')
image_binary.save('result.jpg')
image_binary.size
```

[6]:
```
(2481, 3508)
```

[18]:
```
from pickle import STRING
import collections
from PIL import Image
import numpy as np

def rle_encode(image_array):
  shape = image_array.shape
  image_array = image_array.flatten()  # flatteing the array to get 1-D array
  if len(image_array) == 0: return "0 0"
  enc = f"{shape[0]} {shape[1]} "
  cur_char = image_array[0]
  cur_len = 1
  for i in range(1, len(image_array)):
    if image_array[i]!=cur_char:
      if cur_char==True:
        enc+= f"{1} {cur_len} "
      else:
        enc+= f"{0} {cur_len} "
      cur_char = image_array[i]
      cur_len = 1
    else:
      cur_len+=1
  if cur_char==True:
    enc+= f"{1} {cur_len}"
  else:
```

```
    enc+= f"{0} {cur_len}"
  return enc


def rle_decode(string):
  arr = [int(x) for x in string.split()]
  row = arr[0]
  col = arr[1]

  res = []
  arr = arr[2:]
  for i in range(0, len(arr),2):
    res.extend([arr[i]]*arr[i+1])
  res = np.array(res)
  print("="*10,len(res), row, col, row*col)
  res.reshape((row, col))
  decoded_image = Image.fromarray(res)
  return decoded_image

def rle_encode_image(image):

  # Convert the image to a NumPy array.
  image_array = np.array(image_binary)


  # Encode the flattened image array using RLE.
  encoded_image_data = rle_encode(image_array)

  # Return the encoded image data.
  return encoded_image_data

image = Image.open('doc.jpg')

rle_encoded_image_data = rle_encode_image(image)
print('RLE encoded image data:', rle_encoded_image_data)

file = open("encode.txt", "w")
file.write(rle_encoded_image_data)  # storing the encoding in the form of
 ↪string to a .txt file
file.close()
```

```
RLE encoded image data: 3508 2481 1 790121 0 1 1 1 0 1 1 2407 0 5 1 65 0 5 1 215
0 9 1 2181 0 7 1 64 0 6 1 212 0 14 1 2178 0 7 1 64 0 5 1 211 0 17 1 2177 0 8 1
63 0 6 1 173 0 3 1 33 0 18 1 2176 0 9 1 63 0 5 1 173 0 5 1 32 0 6 1 5 0 8 1 2175
0 4 1 1 0 5 1 241 0 4 1 32 0 4 1 9 0 7 1 2174 0 4 1 1 0 5 1 240 0 5 1 32 0 2 1
12 0 6 1 2173 0 5 1 1 0 5 1 241 0 4 1 46 0 6 1 2173 0 4 1 3 0 5 1 239 0 5 1 47 0
6 1 2171 0 5 1 3 0 5 1 22 0 1 1 22 0 1 1 36 0 1 1 31 0 1 1 29 0 1 1 15 0 1 1 27
```

6 1 13 0 4 1 24 0 4 1 7 0 4 1 7 0 6 1 9 0 4 1 8 0 4 1 11 0 4 1 6 0 5 1 9 0 5 1
2248 0 4 1 11 0 4 1 7 0 4 1 10 0 5 1 8 0 4 1 11 0 4 1 11 0 4 1 7 0 5 1 13 0 3 1
25 0 4 1 8 0 4 1 7 0 4 1 10 0 5 1 7 0 4 1 11 0 4 1 6 0 4 1 11 0 5 1 2247 0 3 1
12 0 4 1 7 0 4 1 11 0 3 1 10 0 4 1 10 0 4 1 11 0 4 1 7 0 4 1 13 0 5 1 24 0 4 1 8
0 4 1 7 0 4 1 11 0 3 1 8 0 4 1 11 0 4 1 6 0 4 1 12 0 3 1 2247 0 5 1 11 0 4 1 7 0
4 1 10 0 5 1 9 0 4 1 9 0 4 1 12 0 4 1 7 0 4 1 14 0 4 1 24 0 4 1 8 0 4 1 7 0 4 1
10 0 5 1 6 0 4 1 13 0 3 1 5 0 5 1 11 0 5 1 2246 0 21 1 5 0 4 1 12 0 4 1 8 0 5 1
9 0 16 1 1 0 4 1 5 0 5 1 14 0 4 1 18 0 1 1 1 0 2 1 1 0 1 1 1 0 3 1 7 0 5 1 7 0 4
1 11 0 4 1 6 0 21 1 4 0 4 1 13 0 3 1 2247 0 20 1 7 0 4 1 11 0 4 1 9 0 4 1 9 0 20
1 7 0 4 1 14 0 4 1 14 0 14 1 8 0 4 1 6 0 4 1 12 0 4 1 6 0 20 1 5 0 5 1 11 0 5 1
2246 0 20 1 7 0 4 1 11 0 4 1 9 0 4 1 9 0 20 1 7 0 4 1 14 0 4 1 13 0 15 1 8 0 3 1
8 0 4 1 10 0 5 1 6 0 20 1 5 0 4 1 13 0 3 1 2248 0 3 1 23 0 4 1 10 0 5 1 8 0 4 1
10 0 4 1 23 0 3 1 14 0 4 1 13 0 6 1 1 0 1 1 4 0 4 1 8 0 4 1 7 0 4 1 11 0 3 1 7 0
4 1 21 0 4 1 12 0 5 1 2246 0 5 1 21 0 5 1 11 0 3 1 10 0 4 1 9 0 5 1 21 0 5 1 14
0 4 1 11 0 5 1 8 0 5 1 6 0 5 1 7 0 4 1 11 0 4 1 6 0 5 1 21 0 4 1 11 0 4 1 2247 0
4 1 23 0 4 1 11 0 4 1 9 0 4 1 10 0 3 1 23 0 4 1 14 0 4 1 10 0 5 1 9 0 4 1 8 0 4
1 7 0 4 1 11 0 4 1 7 0 3 1 21 0 5 1 12 0 4 1 2247 0 4 1 22 0 3 1 12 0 4 1 9 0 4
1 9 0 5 1 22 0 4 1 14 0 4 1 10 0 4 1 10 0 4 1 8 0 4 1 6 0 5 1 11 0 4 1 6 0 5 1
21 0 4 1 11 0 4 1 2248 0 4 1 22 0 4 1 11 0 4 1 8 0 5 1 10 0 4 1 21 0 5 1 14 0 4
1 10 0 4 1 10 0 4 1 8 0 4 1 7 0 3 1 11 0 5 1 7 0 4 1 21 0 4 1 11 0 5 1 2247 0 5
1 20 0 5 1 10 0 5 1 9 0 4 1 10 0 5 1 21 0 3 1 14 0 5 1 10 0 5 1 9 0 4 1 7 0 4 1
8 0 4 1 11 0 3 1 8 0 5 1 20 0 5 1 9 0 5 1 10 0 1 1 2238 0 4 1 12 0 1 1 8 0 4 1
11 0 4 1 9 0 4 1 11 0 4 1 12 0 1 1 8 0 4 1 14 0 4 1 11 0 3 1 9 0 5 1 8 0 4 1 7 0
4 1 11 0 4 1 8 0 4 1 12 0 1 1 7 0 5 1 8 0 7 1 7 0 5 1 2236 0 6 1 7 0 5 1 7 0 4 1
11 0 3 1 10 0 5 1 4 0 1 1 5 0 6 1 7 0 5 1 6 0 5 1 14 0 5 1 4 0 1 1 4 0 6 1 5 0 7
1 8 0 4 1 6 0 5 1 11 0 4 1 8 0 6 1 7 0 5 1 7 0 6 1 4 0 8 1 8 0 5 1 2237 0 16 1 8
0 4 1 11 0 4 1 9 0 10 1 6 0 16 1 8 0 4 1 14 0 10 1 5 0 17 1 8 0 4 1 7 0 4 1 10 0
5 1 9 0 16 1 9 0 13 1 1 0 4 1 7 0 5 1 2238 0 15 1 7 0 5 1 11 0 4 1 10 0 9 1 7 0
15 1 8 0 4 1 15 0 9 1 6 0 11 1 2 0 3 1 7 0 5 1 7 0 3 1 12 0 3 1 11 0 15 1 10 0
11 1 2 0 3 1 8 0 5 1 2240 0 10 1 1 0 1 1 9 0 3 1 11 0 4 1 13 0 7 1 9 0 10 1 1 0
1 1 8 0 4 1 18 0 7 1 8 0 7 1 4 0 3 1 8 0 3 1 8 0 4 1 11 0 4 1 12 0 11 1 13 0 8 1
4 0 4 1 7 0 4 1 2244 0 1 1 1 0 1 1 1 0 1 1 46 0 1 1 16 0 1 1 1 0 1 1 38 0 1 1 14
0 1 1 12 0 1 1 51 0 1 1 1 0 1 1 21 0 1 1 1 0 1 1 3527305

[19]:
```python
import numpy as np
from PIL import Image

class G3FaxEncoder:

    def __init__(self, image):

        self.image = image
        self.width = image.width
        self.height = image.height

    def encode(self):
```

```
        # Initialize the bitstream
        bitstream = bytearray()

        # Start of page (SOP) marker
        bitstream.extend([0xFF, 0x00])

        # Convert image data to a list
        image_data = list(self.image.getdata())

        # Iterate over image lines
        for y in range(self.height):
            line = image_data[y * self.width: (y + 1) * self.width]

            # Run-length encoding (RLE) for black and white pixels
            run_length = 0
            for pixel in line:
                if pixel == 0:  # Black pixel
                    run_length += 1
                else:  # White pixel
                    if run_length > 0:
                        bitstream.extend(self.encode_run_length(run_length))
                    run_length = 0

            # End of line (EOL) marker
            bitstream.extend([0x00, 0x00])

        bitstream.extend([0x01, 0x00])

        return bitstream

    def encode_run_length(self, run_length):

        encoded = []
        while run_length >= 0x80:
            encoded.append(0x80 | (run_length & 0x7F))
            run_length >>= 7
        encoded.append(run_length)

        return encoded


image = Image.open('doc.jpg').convert('1') # converting it to a black and white␣
 ↪image

# Create the G3FaxEncoder instance
encoder = G3FaxEncoder(image)
```

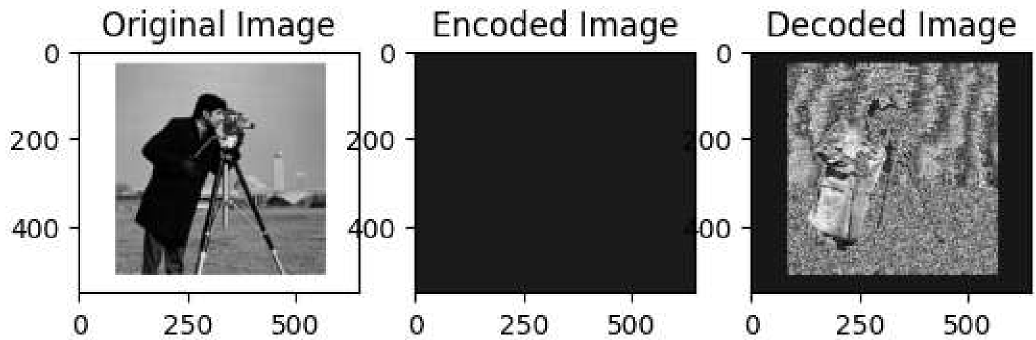
```python
# Encode the image and save it to a file
bitstream = encoder.encode()
with open('bitstream.bin', 'wb') as f:  # outputting it to a .bin file
    f.write(bitstream)
```

ell715q3

October 10, 2023

[5]:
```
import cv2
import numpy as np
from scipy.fftpack import dct, idct
import matplotlib.pyplot as plt
import pywt
import huffman
```

[3]:
```
original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

dct_image = dct(dct(original_image.T, norm='ortho').T, norm='ortho')

quantization_factor = 0.001
quantized_dct_image = np.round(dct_image / quantization_factor)

encoded_data = quantized_dct_image.flatten().astype(np.int16)
encoded_data.tofile('encoded_image.bin')

decoded_dct_image = idct(idct(quantized_dct_image.T, norm='ortho').T,␣
 ↪norm='ortho').astype(np.uint8)

# Display the original and decoded images
plt.subplot(1, 3, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(quantized_dct_image, cmap='gray')
plt.title('Encoded Image')

plt.subplot(1, 3, 3)
plt.imshow(decoded_dct_image, cmap='gray')
plt.title('Decoded Image')

plt.show()
```



[6]:
```
# Load an image using OpenCV
original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

# Perform Haar wavelet transform
coeffs = pywt.dwt2(original_image, 'haar')

# Get the approximation and details coefficients
cA, (cH, cV, cD) = coeffs

# Display the coefficients or perform further processing as needed
cv2.imshow('Approximation (cA)', cA)
cv2.imshow('Horizontal Detail (cH)', cH)
cv2.imshow('Vertical Detail (cV)', cV)
cv2.imshow('Diagonal Detail (cD)', cD)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

[ ]:
```
# Load the image using OpenCV
original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)

# Calculate pixel frequencies
pixel_frequencies = {}
for row in original_image:
    for pixel_value in row:
        if pixel_value in pixel_frequencies:
            pixel_frequencies[pixel_value] += 1
        else:
            pixel_frequencies[pixel_value] = 1

# Build the Huffman tree
huff_tree = huffman.build_tree(pixel_frequencies)

# Generate Huffman codes
```

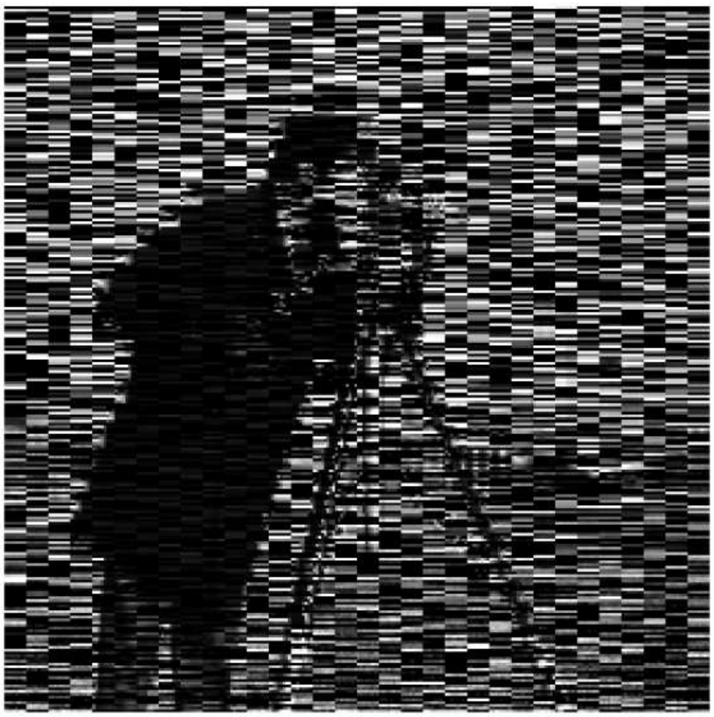
```python
huff_codes = huffman.get_codes(huff_tree)

# Encode the image using Huffman codes
encoded_image = []
for row in original_image:
    encoded_row = [huff_codes[pixel] for pixel in row]
    encoded_image.append(encoded_row)

with open('encoded_image.txt', 'w') as f:
    for row in encoded_image:
        f.write(' '.join(row) + '\n')
```

[ ]:

```python

```

```
clc;
close all;
clear all;
I=imread('cameraman.tif');
I=im2double(I);
m=1;
for i=1:8:256
    for j=1:8:256
        for x=0:7
            for y=0:7
            img(x+1,y+1)=I(i+x,j+y);
            end
        end
            k=0;
            for l=1:8
                img_expect{k+1}=img(:,l)*img(:,l)';
                k=k+1;
            end
            imgexp=zeros(8:8);
            for l=1:8
                imgexp=imgexp+(1/8)*img_expect{l};%expectation of E[xx']
            end
            img_mean=zeros(8,1);
            for l=1:8
                img_mean=img_mean+(1/8)*img(:,l);
            end
            img_mean_trans=img_mean*img_mean';
            img_covariance=imgexp - img_mean_trans;
            [v{m},d{m}]=eig(img_covariance);
            temp=v{m};
             m=m+1;
            for l=1:8
                v{m-1}(:,l)=temp(:,8-(l-1));
            end
             for l=1:8
           trans_img1(:,l)=v{m-1}*img(:,l);
             end
           for x=0:7
               for  y=0:7
                   transformed_img(i+x,j+y)=trans_img1(x+1,y+1);
               end
           end
mask=[1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 ];
```

```
trans_img=trans_img1.*mask;
          for l=1:8
          inv_trans_img(:,l)=v{m-1}'*trans_img(:,l);
          end
           for x=0:7
              for  y=0:7
                 inv_transformed_img(i+x,j+y)=inv_trans_img(x+1,y+1);
              end
          end

         end
end
imshow(transformed_img);
```



```
figure
imshow(inv_transformed_img);
```