

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:

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

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 w_1 and w_2 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 $w_1 + w_2 = 1$ 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

```
1 from PIL import Image
2 import numpy as np
3
4 image_path = '/content/harry.jpg'
5 image = Image.open(image_path)
6
7 # Converting the image to grayscale
8 if image.mode != 'L':
9     image = image.convert('L')
10
11 # Converting the grayscale image to a NumPy matrix
12 image_matrix = np.array(image)
13
14 print("Image Matrix Shape:", image_matrix.shape)
15
16
17 import numpy as np
18 from scipy.optimize import minimize
19 import matplotlib.pyplot as plt
20
21 f = image_matrix
22
23 # Define the objective function to calculate MSE and then quantize error
24 def calculate_mse_and_quantize(weights, original_image):
25     w1, w2 = weights
26     M, N = original_image.shape
27     F = np.zeros((M, N))
28
29     for i in range(M):
30         for j in range(N):
31             if i == 0 and j == 0:
32                 F[i][j] = (int)(original_image[i][j])
33             elif i == 0:
34                 # F[i][j] = w1 * F[i][j-1]
35                 F[i][j] = (int)(original_image[i][j])
36             elif j == 0:
37                 # F[i][j] = w2 * F[i-1][j]
38                 F[i][j] = (int)(original_image[i][j])
39             else:
40                 F[i][j] = (int)(w1 * original_image[i][j-1] + w2 *
41                                original_image[i-1][j])
42
43 prediction_errors = original_image - F
```

```

43     flatten_array = prediction_errors.flatten()
44     quantized_errors = np.zeros(flatten_array.size)
45     # print(flatten_array.size)
46     for i in range(flatten_array.size):
47         quantized_errors[i] = quantize_error(flatten_array[i])
48     mse = np.mean(quantized_errors**2)
49
50     return mse, F
51
52 # Define the quantization function for prediction errors
53 def quantize_error(error):
54     if error < -1:
55         return -1
56     elif error >= -1 and error <= 1:
57         return 0
58     else:
59         return 1
60
61 # Define the optimization objective (minimize quantized MSE)
62 def objective(weights):
63     return calculate_mse_and_quantize(weights, image_matrix)[0] # Use
64         image_matrix instead of predefined matrix f
65
66 # Initial guess for weights (w1, w2)
67 initial_weights = [0.5, 0.5]
68
69 # Define bounds for weights
70 bounds = [(0, 1), (0, 1)]
71
72 # Add x1 + x2 = 1 as an equality constraint
73 constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] -
74     1},)
75
76 # Optimize weights to minimize quantized MSE
77 result = minimize(objective, initial_weights, bounds=bounds,
78     constraints=constraints)
79
80 # Extracting the optimized weights
81 optimized_weights = result.x
82 w1_optimized, w2_optimized = optimized_weights
83
84 # Print the optimized weights and quantized MSE
85 print("Optimized Weights (w1, w2):", optimized_weights)
86 print("Optimized Quantized MSE:", result.fun)
87
88 # Calculating the predicted image using the optimized weights
89 optimal_mse, predicted_matrix = calculate_mse_and_quantize(optimized_weights,
90     image_matrix)
91
92 predicted_image = Image.fromarray(predicted_matrix)
93
94 # Display both the original and predicted images side by side
95 plt.figure(figsize=(12, 6))
96
97 # Original Image
98 plt.subplot(1, 2, 1)
99 plt.imshow(image_matrix, cmap='gray')
100 plt.title("Original Image")
101
102 # Predicted Image
103 plt.subplot(1, 2, 2)
104 plt.imshow(predicted_image, cmap='gray')
105 plt.title("Predicted Image")

```

```

102 plt.show()
103
104
105
106 # In the below code we tried to experiment it using the updated values to
    predict the new pixel values
107
108 # Define the objective function to calculate MSE and then quantize error
109 def calculate_mse_and_quantize(weights, original_image):
110     w1, w2 = weights
111     M, N = original_image.shape
112     F = np.zeros((M, N))
113
114     for i in range(M):
115         for j in range(N):
116             if i == 0 and j == 0:
117                 F[i][j] = (int)(original_image[i][j])
118             elif i == 0:
119                 # F[i][j] = w1*F[i][j-1]
120                 F[i][j] = (int)(original_image[i][j])
121             elif j == 0:
122                 # F[i][j] = w2*F[i-1][j]
123                 F[i][j] = (int)(original_image[i][j])
124             else:
125                 F[i][j] = (int)(w1 * F[i][j-1] + w2 * F[i-1][j])
126
127     prediction_errors = original_image - F
128     flatten_array = prediction_errors.flatten()
129     quantized_errors = np.zeros(flatten_array.size)
130     # print(flatten_array.size)
131     for i in range(flatten_array.size):
132         quantized_errors[i] = quantize_error(flatten_array[i])
133     mse = np.mean(quantized_errors**2)
134
135     return mse, F
136
137 # Define the quantization function for prediction errors
138 def quantize_error(error):
139     if error < -1:
140         return -1
141     elif error >= -1 and error <= 1:
142         return 0
143     else:
144         return 1
145
146 # Define the optimization objective
147 def objective(weights):
148     return calculate_mse_and_quantize(weights, image_matrix)[0] # Use
        image_matrix instead of predefined matrix f
149
150 # Initial guess for weights (w1, w2)
151 initial_weights = [0.5, 0.5]
152
153 # Define bounds for weights
154 bounds = [(0, 1), (0, 1)] # weights between 0 and 1
155
156 # Add x1 + x2 = 1 as an equality constraint
157 constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] -
    1},)
158
159 # Optimize weights to minimize quantized MSE
160 result = minimize(objective, initial_weights, bounds=bounds,
    constraints=constraints)

```

```

161
162 # Extract the optimized weights
163 optimized_weights = result.x
164 w1_optimized, w2_optimized = optimized_weights
165 # print("optimized weights:", result.x)
166
167 # Print the optimized weights and quantized MSE
168 print("Optimized Weights (w1, w2):", optimized_weights)
169 print("Optimized Quantized MSE:", result.fun)
170
171 # Calculating the predicted image using the optimized weights
172 optimal_mse, predicted_matrix = calculate_mse_and_quantize(optimized_weights,
    image_matrix)
173
174 predicted_image = Image.fromarray(predicted_matrix)
175
176 # Display both the original and predicted images side by side
177 plt.figure(figsize=(12, 6))
178
179 # Original Image
180 plt.subplot(1, 2, 1)
181 plt.imshow(image_matrix, cmap='gray')
182 plt.title("Original Image")
183
184 # Predicted Image
185 plt.subplot(1, 2, 2)
186 plt.imshow(predicted_image, cmap='gray')
187 plt.title("Predicted Image")
188
189 plt.show()
190
191
192 # 4 neighbours to predict the pixel value
193
194 def calculate_mse_and_quantize_neighbors(weights, original_image):
195     w_above, w_below, w_left, w_right = weights
196     M, N = original_image.shape
197     F = np.zeros((M, N))
198
199     for i in range(M):
200         for j in range(N):
201             if i == 0 and j == 0:
202                 F[i][j] = original_image[i][j]
203             elif i == 0:
204                 # F[i][j] = (w_left * F[i][j - 1] + original_image[i][j]) /
205                     (w_left + 1)
206                 F[i][j] = original_image[i][j]
207             elif j == 0:
208                 # F[i][j] = (w_above * F[i - 1][j] + original_image[i][j]) /
209                     (w_above + 1)
210                 F[i][j] = original_image[i][j]
211             else:
212                 neighbors = [
213                     w_above * original_image[i - 1][j], # Above
214                     w_below * original_image[i + 1][j] if i + 1 < M else
215                         original_image[i][j], # Below (with boundary check)
216                     w_left * original_image[i][j - 1], # Left
217                     w_right * original_image[i][j + 1] if j + 1 < N else
218                         original_image[i][j] # Right (with boundary check)
219                 ]
220                 F[i][j] = sum(neighbors)
221
222 prediction_errors = original_image - F

```

```

219     flatten_array = prediction_errors.flatten()
220     quantized_errors = np.zeros(flatten_array.size)
221     # print(flatten_array.size)
222     for i in range(flatten_array.size):
223         quantized_errors[i] = quantize_error(flatten_array[i])
224     mse = np.mean(quantized_errors**2)
225     return mse, F
226
227 # Define the quantization function for prediction errors
228 def quantize_error(error):
229     if error < -1:
230         return -1
231     elif error >= -1 and error <= 1:
232         return 0
233     else:
234         return 1
235
236 # Define the optimization objective (minimize quantized MSE)
237 def objective(weights):
238     return calculate_mse_and_quantize_neighbors(weights, image_matrix)[0] #
239     Use image_matrix instead of predefined matrix f
240
241 # Initial guess for weights (w_above, w_below, w_left, w_right)
242 initial_weights = [0.25, 0.25, 0.25, 0.25]
243
244 # Add x1 + x2 + x3 + x4 = 1 as an equality constraint
245 constraints = ({'type': 'eq', 'fun': lambda weights: weights[0] + weights[1] +
246     weights[2] + weights[3] - 1},)
247
248 # Define bounds for weights
249 bounds = [(0, 1), (0, 1), (0, 1), (0, 1)] # weights between 0 and 1
250
251 # Optimize weights to minimize quantized MSE
252 result = minimize(objective, initial_weights, bounds=bounds,
253     constraints=constraints)
254
255 # Extract the optimized weights
256 optimized_weights = result.x
257 w_above_opt, w_below_opt, w_left_opt, w_right_opt = optimized_weights
258
259 # Print the optimized weights and quantized MSE
260 print("Optimized Weights (w_above, w_below, w_left, w_right):",
261     optimized_weights)
262 print("Optimized Quantized MSE:", result.fun)
263
264 # Calculating the predicted image using the optimized weights
265 optimal_mse, predicted_matrix1 =
266     calculate_mse_and_quantize_neighbors(optimized_weights, image_matrix)
267
268 predicted_image1 = Image.fromarray(predicted_matrix1)
269
270 # Display both the original and predicted images side by side
271 plt.figure(figsize=(12, 6))
272
273 # Original Image
274 plt.subplot(1, 2, 1)
275 plt.imshow(image_matrix, cmap='gray')
276 plt.title("Original Image")
277
278 # Predicted Image
279 plt.subplot(1, 2, 2)
280 plt.imshow(predicted_image1, cmap='gray')
281 plt.title("Predicted Image")

```

```
277  
278 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

```
1 import cv2
2 from PIL import Image
3 import numpy as np
4
5 # Load the input image
6 input_image = Image.open('doc.jpg')
7
8 # Convert the input image to binary (black and white)
9 binary_image = input_image.convert('1')
10
11 # Save the binary image
12 binary_image.save('result.jpg')
13
14 # Get the size of the binary image
15 image_size = binary_image.size
16
17 # Function to perform run-length encoding on a binary image
18 def rle_encode(image_array):
19     shape = image_array.shape
20     image_array = image_array.flatten()
21
22     if len(image_array) == 0:
23         return "0 0"
24
25     encoded_data = f"{shape[0]} {shape[1]} "
26     current_pixel = image_array[0]
27     current_length = 1
28
29     for i in range(1, len(image_array)):
30         if image_array[i] != current_pixel:
31             if current_pixel == True:
32                 encoded_data += f"{1} {current_length} "
33             else:
34                 encoded_data += f"{0} {current_length} "
35             current_pixel = image_array[i]
36             current_length = 1
37         else:
38             current_length += 1
39
40     if current_pixel == True:
41         encoded_data += f"{1} {current_length}"
42     else:
43         encoded_data += f"{0} {current_length}"
44     return encoded_data
45
46 # Function to decode a run-length encoded string back to an image
47 def rle_decode(encoded_data):
48     arr = [int(x) for x in encoded_data.split()]
49     rows = arr[0]
50     cols = arr[1]
51     data = arr[2:]
52
53     decoded_data = []
```



```

54
55     for i in range(0, len(data), 2):
56         pixel_value = data[i]
57         run_length = data[i + 1]
58         decoded_data.extend([pixel_value] * run_length)
59
60     decoded_data = np.array(decoded_data, dtype=np.uint8)
61     decoded_image = Image.fromarray(decoded_data.reshape(rows, cols))
62     return decoded_image
63
64 # Function to encode the binary image using run-length encoding
65 def rle_encode_image(binary_image):
66     image_array = np.array(binary_image)
67     encoded_image_data = rle_encode(image_array)
68     return encoded_image_data
69
70 # Load the input image
71 input_image = Image.open('doc.jpg')
72
73 # Convert the input image to binary (black and white)
74 binary_image = input_image.convert('1')
75
76 # Encode the binary image using run-length encoding
77 rle_encoded_data = rle_encode_image(binary_image)
78 # print('RLE encoded image data:', rle_encoded_data)
79
80 # Save the encoded data to a text file
81 with open("encoded_data.txt", "w") as encoded_file:
82     encoded_file.write(rle_encoded_data)
83
84 # Class to perform G3Fax encoding on a binary image
85 class G3FaxEncoder:
86
87     def __init__(self, image):
88         self.image = image
89         self.width = image.width
90         self.height = image.height
91
92     # Encode the binary image using G3Fax encoding
93     def encode(self):
94         bitstream = bytearray()
95
96         # Start of page (SOP) marker
97         bitstream.extend([0xFF, 0x00])
98
99         image_data = list(self.image.getdata())
100
101         for y in range(self.height):
102             line = image_data[y * self.width: (y + 1) * self.width]
103             run_length = 0
104
105             for pixel in line:
106                 if pixel == 0: # Black pixel
107                     run_length += 1
108                 else: # White pixel
109                     if run_length > 0:
110                         bitstream.extend(self.encode_run_length(run_length))
111                         run_length = 0
112
113             # End of line (EOL) marker
114             bitstream.extend([0x00, 0x00])
115
116         bitstream.extend([0x01, 0x00])

```

```

117         return bitstream
118
119     # Encode run length for G3Fax
120     def encode_run_length(self, run_length):
121         encoded = []
122
123         while run_length >= 0x80:
124             encoded.append(0x80 | (run_length & 0x7F))
125             run_length >>= 7
126         encoded.append(run_length)
127         return encoded
128
129     # Create an instance of the G3FaxEncoder
130     g3fax_encoder = G3FaxEncoder(binary_image)
131
132     # Encode the image using G3Fax and save it to a binary file
133     g3fax_bitstream = g3fax_encoder.encode()
134
135     with open('g3fax_bitstream.bin', 'wb') as f:
136         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: $414\text{KB}/320\text{KB} = 1.29$

2. G3 Fax Encoding:

- The G3 Fax-encoded data yielded an output size of 44KB.
- Compression Ratio: $414\text{KB}/44\text{KB} = 9.41$

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

```
1 import cv2
2 import numpy as np
3 from scipy.fftpack import dct, idct
4 import matplotlib.pyplot as plt
5 import pywt
6 import huffman
7
8 original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)
9
10 dct_image = dct(dct(original_image.T, norm='ortho').T, norm='ortho')
11
12 quantization_factor = 0.001
13 quantized_dct_image = np.round(dct_image / quantization_factor)
14
15 encoded_data = quantized_dct_image.flatten().astype(np.int16)
16 encoded_data.tofile('encoded_image.bin')
17
18 decoded_dct_image = idct(idct(quantized_dct_image.T, norm='ortho').T,
19     norm='ortho').astype(np.uint8)
20
21 # Display the original and decoded images
22 plt.subplot(1, 3, 1)
23 plt.imshow(original_image, cmap='gray')
24 plt.title('Original Image')
25
26 plt.subplot(1, 3, 2)
27 plt.imshow(quantized_dct_image, cmap='gray')
28 plt.title('Encoded Image')
29
30 plt.subplot(1, 3, 3)
31 plt.imshow(decoded_dct_image, cmap='gray')
32 plt.title('Decoded Image')
33
34 plt.show()
35
36 # Load an image using OpenCV
37 original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)
38
39 # Perform Haar wavelet transform
40 coeffs = pywt.dwt2(original_image, 'haar')
41
42 # Get the approximation and details coefficients
43 cA, (cH, cV, cD) = coeffs
44
```

```

45 # Display the coefficients or perform further processing as needed
46 cv2.imshow('Approximation (cA)', cA)
47 cv2.imshow('Horizontal Detail (cH)', cH)
48 cv2.imshow('Vertical Detail (cV)', cV)
49 cv2.imshow('Diagonal Detail (cD)', cD)
50
51 cv2.waitKey(0)
52 cv2.destroyAllWindows()
53
54
55 # Load the image using OpenCV
56 original_image = cv2.imread('doc3.jpg', cv2.IMREAD_GRAYSCALE)
57
58 # Calculate pixel frequencies
59 pixel_frequencies = {}
60 for row in original_image:
61     for pixel_value in row:
62         if pixel_value in pixel_frequencies:
63             pixel_frequencies[pixel_value] += 1
64         else:
65             pixel_frequencies[pixel_value] = 1
66
67 # Build the Huffman tree
68 huff_tree = huffman.build_tree(pixel_frequencies)
69
70 # Generate Huffman codes
71 huff_codes = huffman.get_codes(huff_tree)
72
73 # Encode the image using Huffman codes
74 encoded_image = []
75 for row in original_image:
76     encoded_row = [huff_codes[pixel] for pixel in row]
77     encoded_image.append(encoded_row)
78
79 with open('encoded_image.txt', 'w') as f:
80     for row in encoded_image:
81         f.write(' '.join(row) + '\n')

```

KL Transform based coding MATLAB Code

```

1 clc;
2 close all;
3 clear all;
4 I=imread('cameraman.tif');
5 I=im2double(I);
6 m=1;
7 for i=1:8:256
8     for j=1:8:256
9         for x=0:7
10             for y=0:7
11                 img(x+1,y+1)=I(i+x,j+y);
12             end
13         end
14
15         k=0;
16         for l=1:8
17             img_expect{k+1}=img(:,l)*img(:,l)';
18             k=k+1;
19         end
20
21         imgexp=zeros(8:8);
22         for l=1:8

```

```

23         imgexp=imgexp+(1/8)*img_expect{1};
24         %expectation of  $E[xx']$ 
25     end
26
27     img_mean=zeros(8,1);
28     for l=1:8
29         img_mean=img_mean+(1/8)*img(:,l);
30     end
31
32     img_mean_trans=img_mean*img_mean';
33     img_covariance=imgexp - img_mean_trans;
34     [v{m},d{m}]=eig(img_covariance);
35     temp=v{m};
36     m=m+1;
37
38     for l=1:8
39         v{m-1}(:,l)=temp(:,8-(l-1));
40     end
41
42     for l=1:8
43         trans_img1(:,l)=v{m-1}*img(:,l);
44     end
45
46     for x=0:7
47         for y=0:7
48             transformed_img(i+x,j+y)=trans_img1(x+1,y+1);
49         end
50     end
51
52     mask=[1 1 1 1 1 1 1 1
53           1 1 1 1 1 1 1 1
54           1 1 1 1 1 1 1 1
55           1 1 1 1 1 1 1 1
56           1 1 1 1 1 1 1 1
57           1 1 1 1 1 1 1 1
58           1 1 1 1 1 1 1 1
59           1 1 1 1 1 1 1 1];
60
61     trans_img=trans_img1.*mask;
62     for l=1:8
63         inv_trans_img(:,l)=v{m-1}'*trans_img(:,l);
64     end
65
66     for x=0:7
67         for y=0:7
68             inv_transformed_img(i+x,j+y)=inv_trans_img(x+1,y+1);
69         end
70     end
71
72     end
73 end
74 imshow(transformed_img);
75
76 figure
77 imshow(inv_transformed_img);

```

```

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)

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

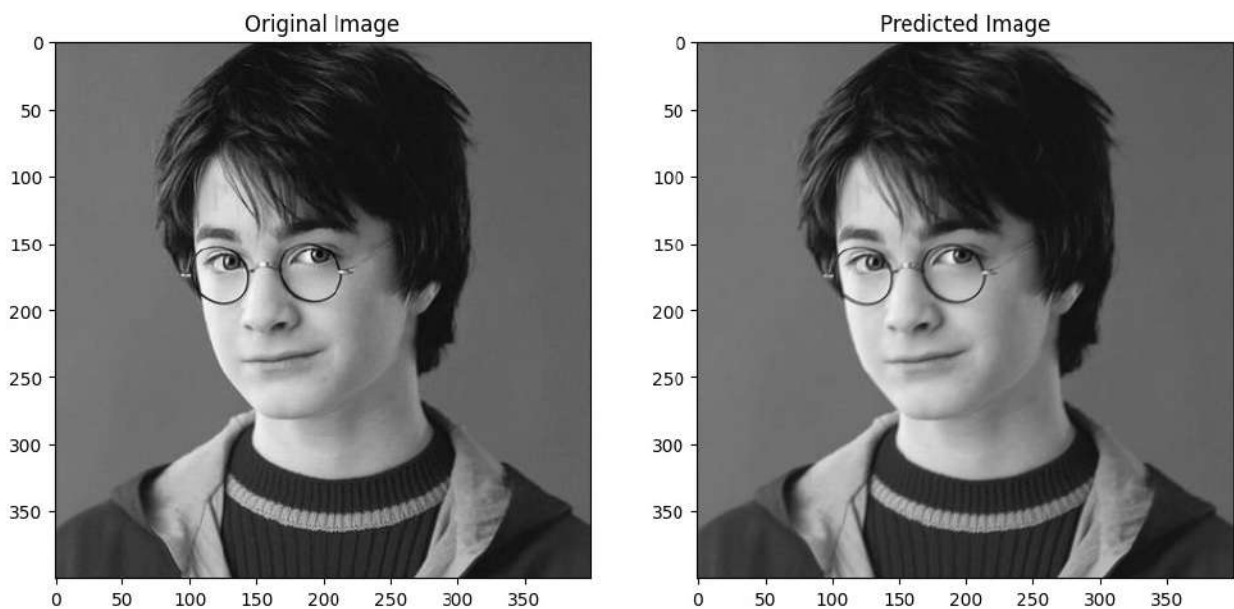
```

```
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))

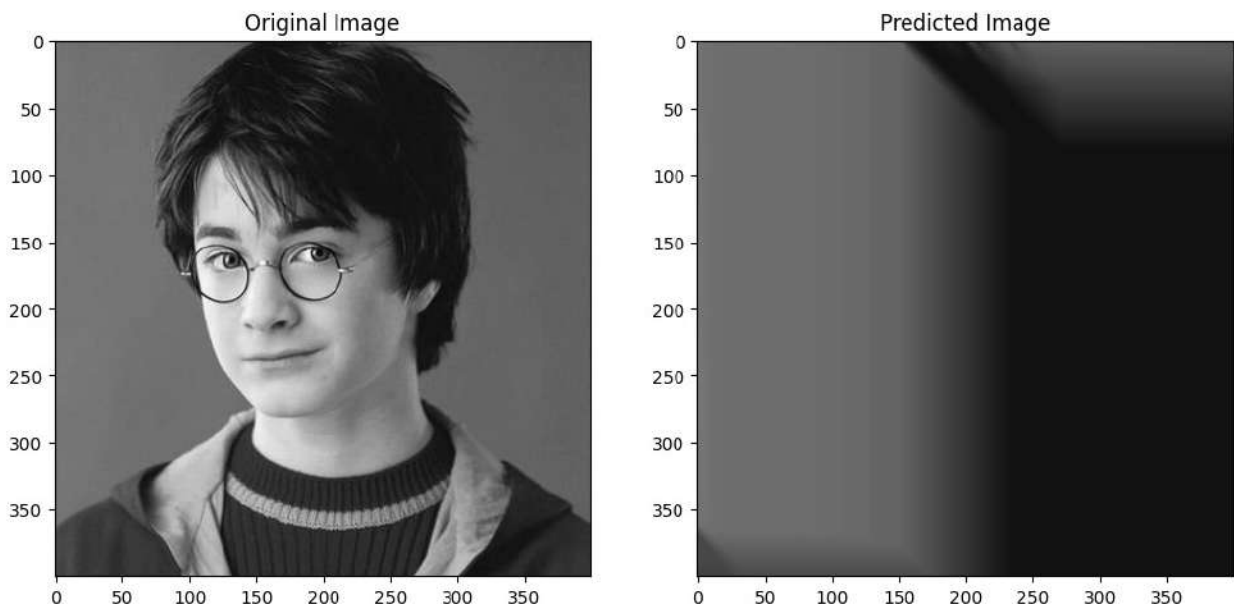
# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# 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

```

```

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

# Add  $x_1 + x_2 + x_3 + x_4 = 1$  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))

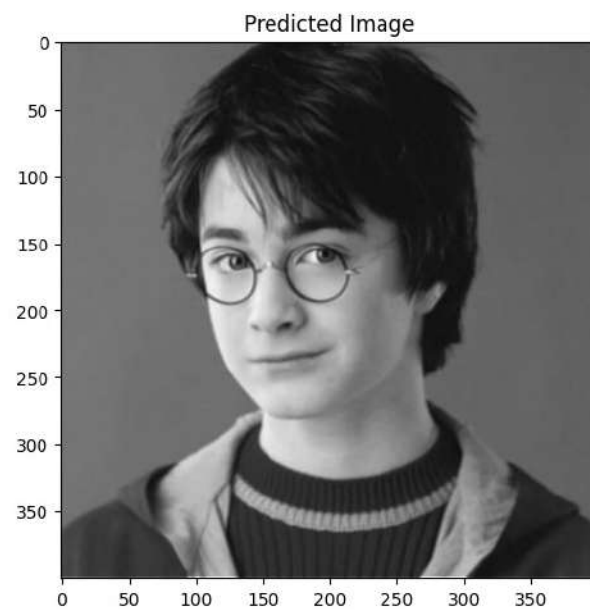
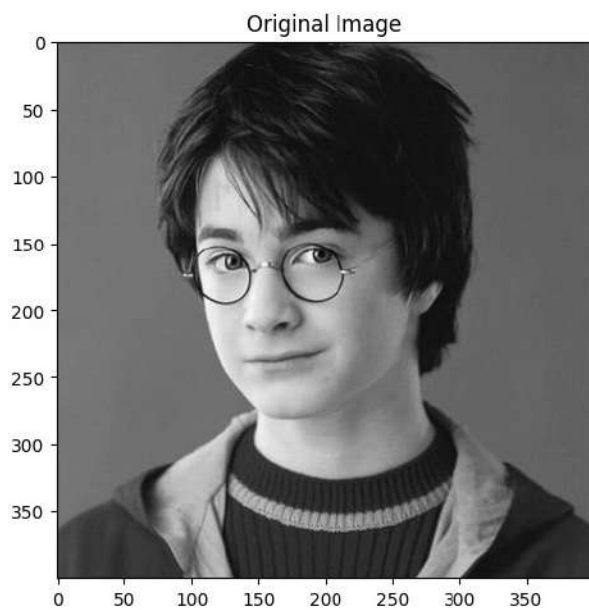
# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_matrix, cmap='gray')
plt.title("Original Image")

# 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

```



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() # flattening 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]: 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)

```

```
# 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')

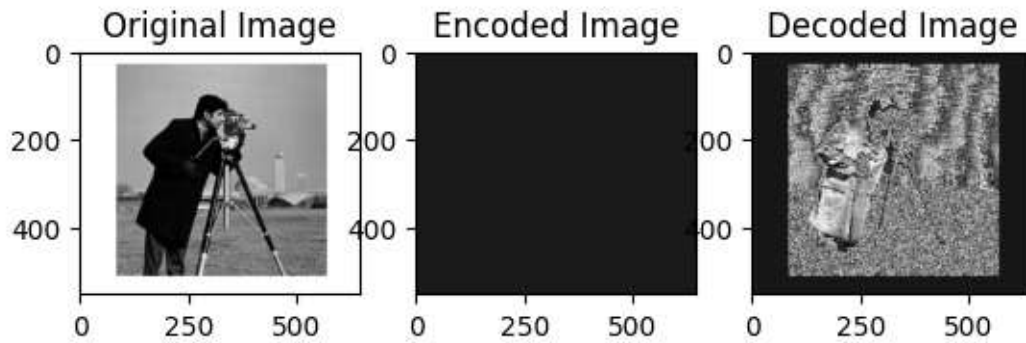
decoded_dct_image = idct(idct(quantized_dct_image.T, norm='ortho').T,
↪ norm='ortho').astype(np.uint8)

# 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
```

```
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')
```

```
[ ]:
```

```

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
    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);

```

