**IMPLEMENTATION OF UNIFORM QUANTISATION IN GRAYSCALE IMAGE**

**Objective: To implement uniform quantization for a grayscale image using Python, and visualize the results including the original image, quantized image, and quantization error. A detailed explanation of each step and line of code is provided.**

**1. Introduction to Quantization:** Quantization in image processing is the process of reducing the number of distinct gray levels in an image. It is essential for compression and reducing complexity. Uniform quantization divides the grayscale range (0-255) into equal intervals and maps each pixel to the center of its interval.

**2. Libraries Used:**

**import cv2**

**import numpy as np**

**import matplotlib.pyplot as plt**

* **cv2: Part of OpenCV, used to load and process images.**
* **numpy: Used for numerical operations like array manipulation.**
* **matplotlib.pyplot: Used for displaying images and plots.**

**3. Loading the Image:**

**image\_path = 'dog1.jpg'**

**image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)**

* **image\_path: A string with the file path of the image to be processed.**
* **cv2.imread(..., cv2.IMREAD\_GRAYSCALE): Loads the image in grayscale format.**

**Image Existence Check:**

**if image is None:**

**raise FileNotFoundError(f"Image not found at path: {image\_path}")**

* **Prevents the program from crashing if the image isn’t found or the path is incorrect.**

**4. Defining the Quantization Function:**

**def uniform\_quantize(image, num\_levels):**

**step = 256 // num\_levels**

**quantized\_image = (image // step) \* step + step // 2**

**quantized\_image = np.clip(quantized\_image, 0, 255).astype(np.uint8)**

**return quantized\_image**

* **step: Determines the width of each quantization interval.**
* **image // step: Finds which interval each pixel falls into.**
* **\* step + step // 2: Maps pixel to the center of its interval.**
* **np.clip(...): Ensures pixel values remain in the range [0, 255].**
* **.astype(np.uint8): Converts the image to standard 8-bit image format.**

**5. Applying Quantization:**

**num\_levels = 4**

**quantized = uniform\_quantize(image, num\_levels)**

* **Chooses to quantize the image to 4 levels (2-bit image).**

**6. Calculating Quantization Error:**

**error = np.abs(image.astype(np.int16) - quantized.astype(np.int16))**

* **astype(np.int16): Ensures that subtraction doesn’t overflow or underflow.**
* **np.abs(...): Computes the absolute error between original and quantized images.**

**7. Displaying the Images:**

**plt.figure(figsize=(15, 5))**

* **Creates a large figure to show all plots side-by-side.**

**Original Image:**

**plt.subplot(1, 3, 1)**

**plt.imshow(image, cmap='gray')**

**plt.title('Original Image')**

**plt.axis('off')**

* **Displays the original grayscale image.**

**Quantized Image:**

**plt.subplot(1, 3, 2)**

**plt.imshow(quantized, cmap='gray')**

**plt.title(f'Quantized Image ({num\_levels} levels)')**

**plt.axis('off')**

* **Displays the quantized image with reduced gray levels.**

**Quantization Error Map:**

**plt.subplot(1, 3, 3)**

**plt.imshow(error, cmap='hot')**

**plt.title('Quantization Error')**

**plt.colorbar()**

**plt.axis('off')**

* **Shows a heatmap of quantization error. Bright colors indicate larger errors.**

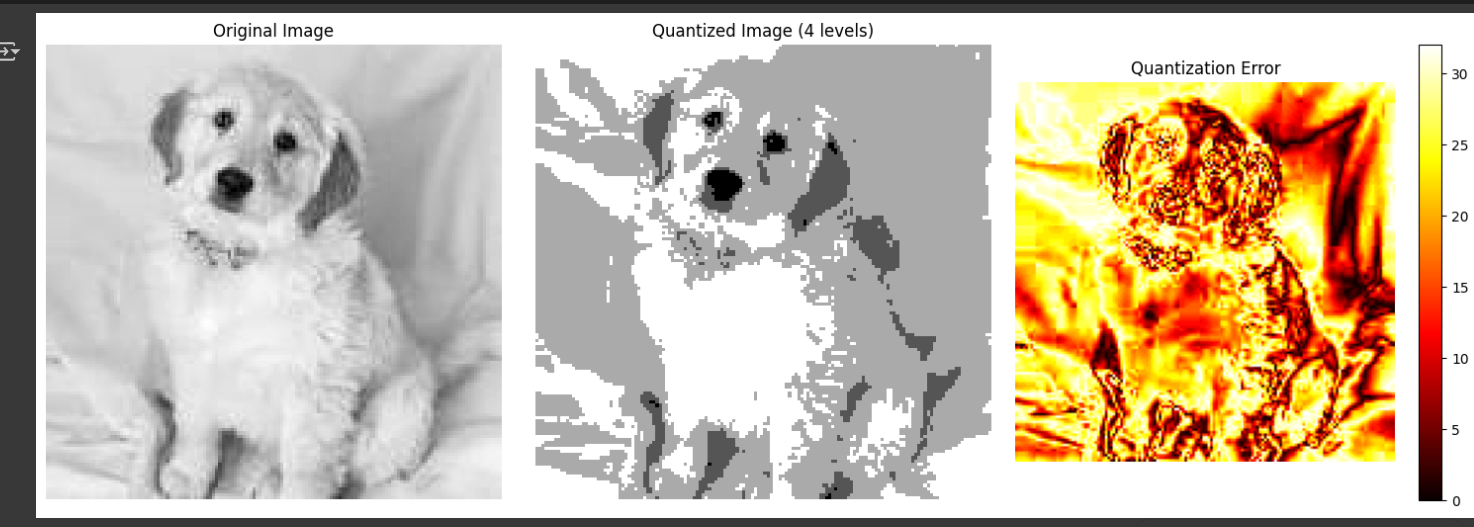
**Final Layout:**

**plt.tight\_layout()**

**plt.show()**

* **Adjusts spacing and renders all images in one window.**

**8. Conclusion: This experiment demonstrates how uniform quantization can simplify image representation by reducing the number of gray levels. While it helps reduce complexity, it introduces some quantization error, visible as a loss in detail. By visualizing the original, quantized, and error images, we gain a clearer understanding of the trade-offs involved.**

****

**IMAGE COMPRESSION**

**Objective:** To implement uniform quantization for a grayscale image and extend the work to JPEG-like image compression using Discrete Cosine Transform (DCT) and quantization. Both tasks include visualization and detailed explanation of each line of code.

**1. Introduction to Quantization and Compression:** Quantization is the process of reducing the number of distinct values in data. In image processing, this typically involves reducing grayscale or color levels. Compression further utilizes quantization to reduce storage space, especially through frequency-domain transformations like the Discrete Cosine Transform (DCT), as done in JPEG.

**2. Libraries Used:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.fftpack import dct, idct

* cv2: Loads and processes images.
* numpy: Handles numerical and matrix operations.
* matplotlib.pyplot: Displays images and graphs.
* scipy.fftpack.dct, idct: Apply and reverse the DCT transform.

**3. Loading the Image:**

image = cv2.imread('dog1.jpg', cv2.IMREAD\_GRAYSCALE)

if image is None:

raise FileNotFoundError("Image not found!")

* Loads a grayscale image.
* If loading fails, raises an error.

**4. Define Block Size and Quantization Matrix:**

block\_size = 8

Q = np.array([...]) # Standard JPEG quantization matrix

* JPEG processes images in 8x8 blocks.
* Q controls the compression ratio. Larger values result in more compression but more loss.

**5. Padding the Image:**

h, w = image.shape

h\_pad = (block\_size - h % block\_size) % block\_size

w\_pad = (block\_size - w % block\_size) % block\_size

padded\_image = np.pad(image, ((0, h\_pad), (0, w\_pad)), 'constant', constant\_values=0)

* Ensures image dimensions are divisible by 8.
* Pads with zeros if necessary.

**6. Compression Function (DCT + Quantization):**

def compress(img, Q):

h, w = img.shape

compressed = np.zeros((h, w), dtype=np.int32)

for i in range(0, h, block\_size):

for j in range(0, w, block\_size):

block = img[i:i+block\_size, j:j+block\_size] - 128

dct\_block = dct(dct(block.T, norm='ortho').T, norm='ortho')

quantized = np.round(dct\_block / Q)

compressed[i:i+block\_size, j:j+block\_size] = quantized

return compressed

* block = ... - 128: Centers pixel values for DCT.
* dct(...): Applies 2D DCT.
* quantized = ...: Divides DCT coefficients by Q and rounds.

**7. Decompression Function (Inverse DCT):**

def decompress(compressed, Q):

h, w = compressed.shape

decompressed = np.zeros((h, w), dtype=np.float32)

for i in range(0, h, block\_size):

for j in range(0, w, block\_size):

quantized = compressed[i:i+block\_size, j:j+block\_size]

dequantized = quantized \* Q

idct\_block = idct(idct(dequantized.T, norm='ortho').T, norm='ortho') + 128

decompressed[i:i+block\_size, j:j+block\_size] = idct\_block

return np.clip(decompressed, 0, 255).astype(np.uint8)

* Reverses quantization and DCT.
* +128 undoes the earlier centering.
* clip(...).astype(...): Ensures valid 8-bit image format.

**8. Run Compression and Decompression:**

compressed\_image = compress(padded\_image, Q)

reconstructed\_image = decompress(compressed\_image, Q)

reconstructed\_image = reconstructed\_image[:h, :w]

* Applies compression to padded image.
* Crops back to original dimensions.

**9. Display Original vs Compressed Image:**

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.imshow(image, cmap='gray')

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(reconstructed\_image, cmap='gray')

plt.title('Compressed Image (JPEG-like)')

plt.axis('off')

plt.tight\_layout()

plt.show()

* Shows original and reconstructed images side by side.
* Helps visualize compression loss.

**10. Conclusion:** This compression algorithm simulates JPEG by applying DCT to 8x8 blocks, followed by quantization. The reconstructed image may lose some detail but gains in compression efficiency. The balance between quality and compression depends on the quantization matrix.

