

Image compression using K-means clustering algorithm

1. Importing necessary libraries


First, we need to import the libraries required for our analysis: `numpy` for numerical operations, `matplotlib` for plotting images, and `cv2` (OpenCV) for image processing.

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

2. Uploading the image

Next, we upload an image file from our local machine to the Google Colab environment.

```
from google.colab import files
uploaded = files.upload()
```

 Choose Files sample.jpg

- **sample.jpg**(image/jpeg) - 7160799 bytes, last modified: 12/9/2024 - 100% done

3. Preprocessing the image

We define a function, `read_image`, to perform the following tasks: read the uploaded image using OpenCV, convert it from BGR format (used by OpenCV) to RGB format, and scale the pixel values to the range [0, 1] for better numerical stability during processing.

```
def read_image():
    img = cv2.imread('sample.jpg')

    # Converting the image from BGR to RGB
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Scaling the image so that the values are in the range 0-1
    img = img/255.0

    return img
```

4. Initialising cluster means

The `initialise_means` function is responsible for flattening the image into a 2D array (where each row represents a pixel's RGB values), creating an array for the initial cluster means (centroids), and randomly selecting pixels to initialise the cluster means by averaging the RGB values of randomly chosen pixels.

```
def initialise_means(img, clusters):
    # `clusters` is the number of clusters or the number of colours that we choose

    # Reshaping or flattening the image into a 2D matrix
    points = img.reshape((-1, img.shape[2]))
    m, n = points.shape

    means = np.zeros((clusters, n))    # array of assumed means or centroids

    # Random initialisation of means
    for i in range(clusters):
        rand_indices = np.random.choice(m, size=10, replace=False)
        means[i] = np.mean(points[rand_indices], axis=0)

    return points, means
```

5. Calculating distance

We also define a function to compute the Euclidean distance between two points in a 2D space. It is used to find the closest centroid to each pixel during the clustering process.

```
def distance(x1, y1, x2, y2):
    dist = np.square(x1-x2) + np.square(y1-y2)
    dist = np.sqrt(dist)
    return dist
```

6. Implementing the K-means clustering algorithm

The `k_means` function implements the K-means clustering algorithm. It iterates a fixed number of times (10 in this case); for each pixel, it finds the nearest centroid and assigns the pixel to that cluster, and after assigning all pixels, it recalculates the centroid of each cluster based on the assigned pixels.

```
def k_means(points, means, clusters):
    iterations = 10
    m, n = points.shape

    index = np.zeros(m)    # index values that correspond to the cluster to which each pixel belongs

    while iterations > 0:
        for j in range(m):
            min_dist = float('inf')
            temp = None

            for k in range(clusters):
                x1, y1 = points[j,0], points[j,1]
                x2, y2 = points[k,0], points[k,1]

                if distance(x1, y1, x2, y2) <= min_dist:
                    min_dist = distance(x1, y1, x2, y2)
                    temp = k
                    index[j] = k

            for k in range(clusters):
                cluster_points = points[index == k]
                if len(cluster_points) > 0:
                    means[k] = np.mean(cluster_points, axis=0)

        iterations -= 1

    return means, index
```

7. Compressing the image

The `compress_image` function reconstructs the compressed image by replacing each pixel's color with its corresponding centroid. It then reshapes the array back into the original image dimensions, displays the compressed image, and saves it to a file.

```
def compress_image(means, index, img):
    # Recovering the compressed image by assigning each pixel to its corresponding centroid
    centroid = np.array(means)
    recovered = centroid[index.astype(int), : ]

    # Getting back the 3D matrix (row, col, rgb(3))
    recovered = recovered.reshape(img.shape)

    # Plotting the compressed image
    plt.imshow(recovered)
    plt.show()

    # Saving the compressed image
    cv2.imwrite('compressed_' + str(clusters) + '_colors.jpg', recovered)
```

8. Displaying the original image

We now display the original image before processing, providing a visual comparison with the compressed output.

```
plt.imshow(img)
plt.title("Original Image")
plt.show()
```



Original Image



9. Displaying the compressed image

Finally, we define the main execution point of the program.

```
if __name__ == '__main__':  
    img = read_image()  
  
    clusters = 16  
    clusters = int(input("Enter the number of colours in the compressed image (default: 16): "))  
  
    points, means = initialise_means(img, clusters)  
    means, index = k_means(points, means, clusters)  
    compress_image(means, index, img)
```

