Image Compression

Phase #2

Sohrab Namazinia

97522085

Introduction

I have covered the theoretical parts in the first phase, now I have implemented image compression.

I have also uploaded my code with this documentation in LMS.

Dependencies

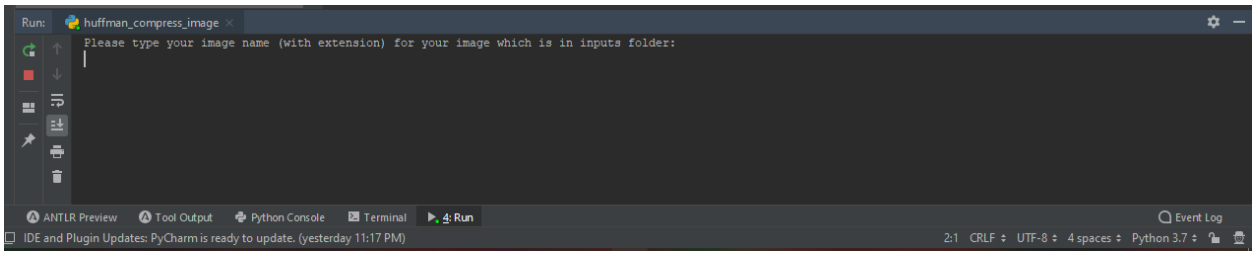
You should have pillow and NumPy installed:

pip3 install pillow

Pip3 install numpy

Run it

Compress_image.py is the python file that I have coded to run to get the result. After you run this file, it prints you this message:



So you must first put your desired photo in the inputs folder. But there are already some photos for testing in this folder.

Then, it runs the codes, uses my methods, creates the compressed image with the name comressed_[image_name] in the outputs folder.

Directories

There are 3 folders in my code:

“Code” which contains my python file to run. also “Inputs” for input images and “outputs” for output images. I have already generated outputs for all my input photos:

K-means

In simple words, the K-means algorithm is for dividing data into k groups so that each group of data is similar to each others. so k = number of our clusters.

So we initially pick k random training examples, which each one has n number of features. Then I do this for a certain number of iterations:

1) For each example, C(i) = closest centroid.
2) For each centroid, we set the location to be the average of its assigning examples.

Finally, centroids will move to the center of the clusters and the overall distance of the examples to the clusters gets smaller.

K-means in Image Compression

Each pixel of an image has 3 metrics R, G & B. so pixels can be seen as the points on the grid in 3D space. So we need to reduce the number of colors by taking the average k colors that look closest to the original image. Fewer colors in the image, less space taken by that which is good for us.

Here are what my functions do for k-means:

```
def init_K_centroids(X, K):
    m = len(X)
    return X[np.random.choice(m, K, replace=False), :]
```

It chooses k starting points randomly.

```
def find_closest_centroids(X, centroids):
    m = len(X)
    c = np.zeros(m)
    for i in range(m):
        distances = np.linalg.norm(X[i] - centroids, axis=1)
        c[i] = np.argmin(distances)
    return c
```

It finds the closest centroid for each of the training examples.

```
def find_means(X, idx, K):
    _, n = X.shape
    centroids = np.zeros((K, n))
    for k in range(K):
        examples = X[np.where(idx == k)]
        mean = [np.mean(column) for column in examples.T]
        centroids[k] = mean
    return centroids
```

It computes the distance of each example to its centroid, then takes an average of distance for each centroid

```
def find_k_means(X, K):
    centroids = init_K_centroids(X, K)
    previous_centroids = centroids
    # i have set max iterations = 10
    for _ in range(10):
        idx = find_closest_centroids(X, centroids)
        centroids = find_means(X, idx, K)
        if (centroids == previous_centroids).all():
            # The centroids aren't moving anymore.
            return centroids
        else:
            previous_centroids = centroids
    return centroids, idx
```

Finally, it does the k-means algorithm, if centroids get converged, the algorithm can finish and it is mostly optimized. So I have returned the result.

Load image

My code consists of these parts in this section:

```
def load_image(path):
    image = Image.open(path)
    return np.asarray(image) / 255
```

It loads an image from the input path, returns the corresponding normalized NumPy array

```
try:
    image_base_path_input = "../inputs/"
    image_base_path_output = "../outputs/compressed_"
    file_name = input("Please type your image name (with extension) for your image which is in inputs folder:\n")
    image_path = image_base_path_input + file_name
    assert os.path.isfile(image_path)
except (IndexError, AssertionError):
    print('File not found!')
```

Then, I have defined some helping path-related variables, take the input file name from the user in the command line, also check if exists, if not, I have thrown an exception.

```
image = load_image(image_path)
w, h, d = image.shape
print('Image dimensions:\n\thas Width = {}, Height = {}, Depth = {}'.format(w, h, d))
```

Then I have loaded the image and just printed its dimensions.

```
X = image.reshape((w * h, d))
K = 20
colors, _ = find_k_means(X, K)
```

Since pixels have the same meaning (all color), so no need to represent it as a grid, that is why I have reshaped it. Also ran K-means to chose the colors.

```
idx = find_closest_centroids(X, colors)
```

But the resulting indices are 1 iteration behind the colors, so I have computed the indices for the current colors, each pixel having a value in 0, K that represents its color.

```
idx = np.array(idx, dtype=np.uint8)
X_reconstructed = np.array(colors[idx, :] * 255, dtype=np.uint8).reshape((w, h, d))
```

Now just we need to replace the color index with the color and then reshape the image to its first dimensions that we printed.

```
result = Image.fromarray(raw_image)
```

Now I just converted raw numbers back to image by Pillow.Image.fromarray

```
result.save(image_base_path_output + file_name)
print("Done\nImage is save in outputs folder")
```

So, finally, we have just saved the compressed image in the “outputs” folder.

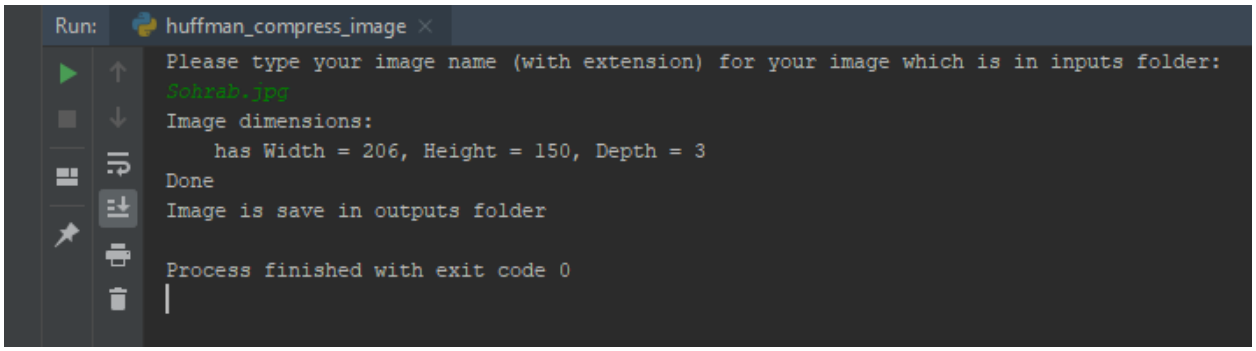
Now everything is done!

you will see the result of image compression in before-after format for each of my test photos.

In the next action, you will see the results of the tests.

Demo

Here is my console result sample format:



Now, you will see the result of image compression in the before-after format for each of my test photos. (left one = before, right one = after compression)