Image Compression and its implementation in real life

Shreyansh Tripathi, Vedant Bonde, Yatharth Rai

Roll No. 11741, 11743, 11745

Cluster Innovation Centre

University of Delhi

Delhi – 110007

Declaration by Candidates

We hereby declare that this project, which is titled "Image Compression and its implementation in real life" is our original work and has not been submitted anywhere else. We also certify that we did not copy any material from previously published or unpublished work without citing the appropriate sources.

Shreyansh Tripathi

Vedant Bonde

Yatharth Rai

B.Tech (1st Year)

CIC, University Of Delhi

Acknowledgement

We would like to acknowledge with thanks and appreciation the efforts of all the people who played a major part in the successful completion of the project.

We would like to express our gratitude to Prof. Shobha Bagai, who bestowed us with her precious guidance and support without which , successful completion of which would have not have been possible.

We would like to thank CIC for providing us with the necessary resources to complete this project.

Contents

Abstract

This report is based on Image Compression and analysis of its methods and techniques focusing especially on the JPEG compression. We analyzed the standard JPEG compression algorithm, its effectiveness, its limitations and its implementation in C++ and Python. We then made an attempt to provide some variations to the existing method of image compression to reduce the redundancy of the images stored on desktop using the python script and library files.

Image

An image is two dimensional array of pixels where each pixel is a fundamental unit which can represent color on its own. Each pixel has its own value which in case of RGB color space has values ranging from 0 to 255. An RGB image consists of 3 layers of Red, Green and Blue color components where each cell represents the intensity of the color component in the given layer.

Image Compression

Image Compression is a way to encode an image which results in the reduction of the size of the digital images without reducing the quality of the image to an unacceptable level which may result in a distorted image.

Types of Image Compression

There are two types of image compression, Lossy and Lossless.

- Lossy Compression: When the image is to be compressed at the cost of its clarity and quality, it is known as Lossy compression. The original image data which was worked upon is lost due to inexact approximations and partial data discarding. Well designed lossy compression like high quality JPEG compression and DjVu compression often reduce the image size significantly while keeping the quality of the image acceptable to the end-user. Lossy compression is often used to compress multimedia data, especially in applications such as streaming media.

- Lossless Compression : When image is compressed without the compromise in image data and the quality of the image is always retained , then the technique is known as Lossless compression. Lossless compression is used in cases when its important that the original image and the decompressed data be identical to one another. Typical examples are the programs PNG and GIF. Lossless compression is often used in applications like ZIP formats and the GNU tools gzip. Its also used in MP3 encoders.

The below image shows the distinction between Lossy and Lossless method of compression.

Fig. Comparison between lossless and lossy images

Observing the above image very carefully , we notice that there is loss of image clarity in the wings and the plant. To be more precise , if we observe the edges of the objects present in the image above, we may notice loss of clarity in the lossy compression.

Process of JPEG Image Compression

JPEG image compression essentially involves the following steps :

1. Color Space Transformation : Colors in an image is usually represented through RGB space. First step in the compression is to represent these colors through another color space YC_bC_r, where Y represents luminosity component which essentially represents brightness or intensity of image pixels. C_b and C_r represent chrominance (or color) of blue and green respectively. This conversion is based on a fact that human eye is more sensitive to luminosity rather than the color components. This transformation is done to achieve high compression. The mathematics we use for this transformation is

$$[Y\ \ C_b\ \ C_r] = [R\ \ G\ \ B]\begin{bmatrix} 0.299 & -0.169 & 0.499 \\ 0.587 & -0.331 & -0.418 \\ 0.114 & 0.500 & -0.081 \end{bmatrix}$$

2. Downsampling : Human eyes contain rods and cones. The former is not sensitive to color and can only detect brightness, on the other hand cones are sensitive to color. The density of cones is very less compared to that of rods so they detect brightness better than color. JPEG uses this fact and reduces the spatial resolution of C_b and C_r components. This is done by merging some pixels together and assigning the same color (chrominance values) to certain group of pixels in accordance with JPEG standards. This is usually done by a factor of 2 in both directions. Figure shows downsampling by factor of 4 (2 in each direction).

3. Block Splitting : After downsampling image is split into blocks of size 8 x 8 pixels. If after block splitting incomplete boxes remain, we fill it repeating the pixel data of edges.

4. Discrete Cosine Transformation : **Discrete Cosine Transform** or **DCT** is used in lossy image compression because it has very strong energy compaction, i.e., its large amount of information is stored in very low frequency component of a signal

and rest other frequency having very small data which can be stored by using very less number of bits (usually, at most 2 or 3 bit).
To perform DCT Transformation on an image, first we have to fetch image file information (pixel value in term of integer having range 0 – 255) which we divide in block of 8 X 8 matrix and then we apply discrete cosine transform on that block of data.
After applying discrete cosine transform, we will see that more than 90% data will be in lower frequency component.

Algorithm

The discrete cosine transform (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. The source code is given below in the C++ 14 language.

Source Code

```
// CPP program to perform discrete cosine transform
#include <bits/stdc++.h>
using namespace std;
#define pi 3.142857
const int m = 8, n = 8;

// Function to find discrete cosine transform and print it
int dctTransform(int matrix[][n])
{
   int i, j, k, l;

   // dct will store the discrete cosine transform
   float dct[m][n];

   float ci, cj, dct1, sum;

   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {

         // ci and cj depends on frequency as well as
         // number of row and columns of specified matrix
         if (i == 0)
            ci = 1 / sqrt(m);
```

```
            else
                ci = sqrt(2) / sqrt(m);
            if (j == 0)
                cj = 1 / sqrt(n);
            else
                cj = sqrt(2) / sqrt(n);

            // sum will temporarily store the sum of
            // cosine signals
            sum = 0;
            for (k = 0; k < m; k++) {
                for (l = 0; l < n; l++) {
                    dct1 = matrix[k][l] *
                           cos((2 * k + 1) * i * pi / (2 * m)) *
                           cos((2 * l + 1) * j * pi / (2 * n));
                    sum = sum + dct1;
                }
            }
            dct[i][j] = ci * cj * sum;
        }
    }

    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%f\t", dct[i][j]);
        }
        printf("\n");
    }
}

// Driver code
int main()
{
    int matrix[m][n] = { { 253, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
                         { 255, 255, 255, 255, 255, 255, 255, 255 },
```

```
                    { 255, 255, 255, 255, 255, 255, 255, 255 } };
   dctTransform(matrix);
   return 0;
}
```

//Time Complexity of the algorithm is O(n*m) due to the two loops running in the DCT function.

The above code was prepared in C++ 14 with complexity $O(n^2)$.

5. Quantization : Human eye has the capability to distinguish small difference in brightness but is not so good at distinguishing high frequency brightness variations. Quantization is a process of reducing number of bits needed to store data of high frequency by reducing precision of its values.

The matrix that we get after DCT step contains all important data at the top left corner and all A.C. signals at lower right corner. JPEG provides us with a quantization matrix. In JPEG standards, to get quantized matrix every element in DCT matrix is divided by corresponding element in quantization matrix. The quantization matrix is designed so as to contain higher value at lower right side so as to give lower values when element of DCT matrix is divided by its corresponding element. Quantization matrix has following structure.

$$\begin{bmatrix} 1 & 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 1 & 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 2 & 2 & 2 & 4 & 8 & 16 & 32 & 64 \\ 4 & 4 & 4 & 4 & 8 & 16 & 32 & 64 \\ 8 & 8 & 8 & 8 & 8 & 16 & 32 & 64 \\ 16 & 16 & 16 & 16 & 16 & 16 & 32 & 64 \\ 32 & 32 & 32 & 32 & 32 & 32 & 32 & 64 \\ 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \end{bmatrix}$$

$$B(i,j) = \frac{M(i,j)}{Q(i,j)}$$

where B(i,j) denotes element of new quantized matrix, M(i,j) denotes element of DCT matrix and Q(i,j) denotes element of quantization matrix

After dividing elements of DCT matrix by corresponding element of quantization matrix, we round off the value to its nearest integer. Thus rounding off most values to 0.
This is the actual lossy step of the whole JPEG compression process.

6. Entropy Encoding : Entropy encoding is a lossless form of storing data where redundant data is stored in fewer bits and non-redundant data is represented in many bits. Since there is redundancy of data (0s), rather than storing the same data many times, we store symbol and a logic with it so as to reduce the amount of data actually stored.

Example : Suppose we have to store 1110001111 in memory. We observe that there is a repetition of data. Now suppose we make a rule to store 1 represented as the number of times of its repetition and three 0 through 0^3. Then the pattern will look like 30^34. This logic will take less space when applied on large amounts of data.
The entropy encoding that we usually use are:

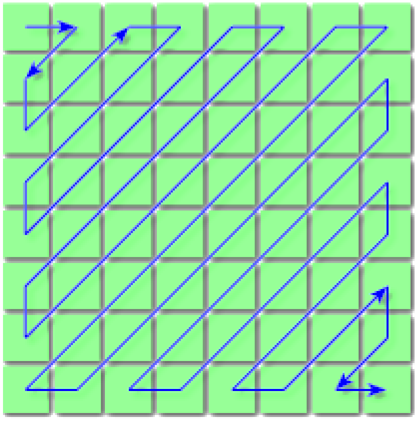
- Run Length Encoding: Suppose we have a long sequence of character

  AAAAAAAAAAAABBBBBBBBBBBBCCCCCCCCCDDDDDDDDDDD

  This data is redundant and would take large space to store. But logically we can also store this by simply writing the symbol along with the repetitions that it has i.e. like,
  A12B12C9D11
  This type of representation would consume less space when stored. This is called Run Length Encoding.
  The actual run length coding of JPEG is applied zig zag on a matrix and is of format,

  (Run-length, Size) (Amplitude)

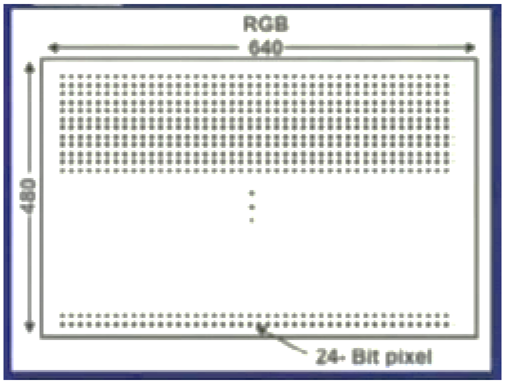
where run-length= the no of zeros before non-zero element, x
size= no of bits required to represent non-zero element, x
amplitude = bit representation of x



Section change :

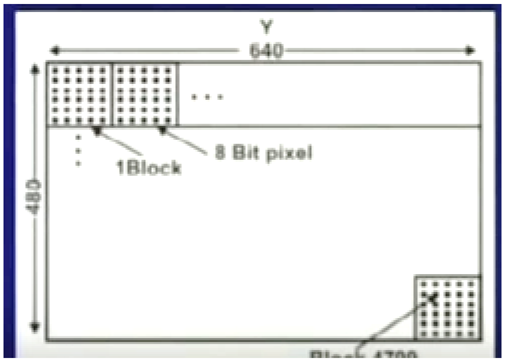
This section gives examples to explain every process of jpeg compression

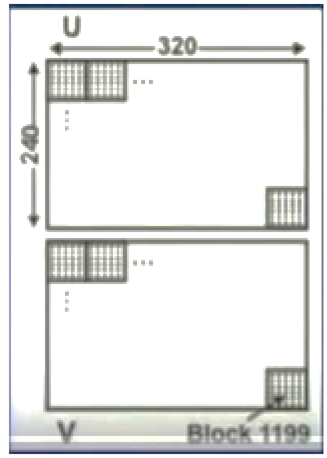
All the steps are explained through the following example :



Suppose we have a 640 X 480 image. It then would have 307,200 pixels in total with 640 pixels on one and 480 pixels on the other edge. Each pixel would have its own RGB values with each of R, G & B having 8 bits so a pixel in total would have 24 bits of data in it.

i. The first step is that we convert the image from RGB color space to YC_bC_r color space. This will split the image into luminance and chrominance components. Next we downsample the image by a factor of 4 (2 in each direction)
ii. In the next step the divide the components of image into blocks of 8X8





Pictures showing block splitting of all three and

downsampling of U= C_b and V= C_r components.

iii. The next step applies discrete cosine transformation on each element of the matrix.

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Suppose above is a matrix of luminance of a certain block. Values in the block represents the grayscale intensity or brightness values. Before computing the DCT of the 8×8 block, its values are shifted from a positive range to one centered on zero i.e. we change the range of values from [0, 255] to [-128, 127]. This can be done by subtracting 128 from each value.

$$\begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix}$$

Next, we apply DCT

$$M = \begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix}$$

The resulting matrix has high values for DC components and AC components show variation from DC component.

iv. The next step is Quantization, for 50% quantization generally we use a different quantization matrix. Each element of DCT matrix is divided by corresponding element in quantization matrix and then rounded off to nearest integer.

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$B(i,j) = \frac{M(i,j)}{Q(i,j)}$$

where B(i,j) denotes element of new quantized matrix, M(i,j) denotes element of DCT matrix and Q(i,j) denotes element of quantization matrix.

Next we round off elements of quantized matrix to get final matrix.

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

v. Now we apply Run Length Encoding to further compress the data to be stored in any storage system.

After Run Length encoding we get,

(0, 2)(−3); (1, 2)(−3); (0, 2)(−2); (0, 3)(−6); (0, 2)(2); (0, 3)(−4); (0, 1)(1); (0, 2)(−3); (0, 1)(1); (0, 1)(1); (0, 3)(5); (0, 1)(1); (0, 2)(2); (0, 1)(−1); (0, 1)(1); (0, 1)(−1); (0, 2)(2); (5, 1)(−1); (0, 1)(−1); (0, 0).

as the data to store.

Average Hashing

The average hash algorithm is a very simple version of a perceptual hash. It’s a good algorithm to use if you’re wanting to find images that are very similar and haven’t had localized corrections performed (like a border or a watermark).

Methodology

1.) Reduce The Image Size : The fastest way to get rid of high frequencies and detail is to reduce the size of the image. The image is shrunk to an 8*8 format so as to contain 64 pixels.

2.) Reduce Color : The reduced image is further converted into grayscale format so as to have a total of 64 colors.

3.) Compute Mean : Compute the mean value of 64 colors.

4.) Compute the Bits : Each bit is simply set based on whether the color value is above or below the mean.

5.) Construct the hash : Set the resultant 64 bits into a 64-bit integer.

The code was implemented in python and after the definition of phash and use of the libraries numpy, scipy and PIL . The resulting definition was used in the imagehash library.

Source Code

```
def
average_hash(image,
hash_size=8):
        image = image.convert("L").resize((hash_size,
hash_size), Image.ANTIALIAS)
        pixels =
numpy.array(image.getdata()).reshape((hash_size,
hash_size))
        avg = pixels.mean()
        diff = pixels > avg
        # make a hash
        return ImageHash(diff)
```

Perceptual Hashing

Perceptual hashing is the use of algorithms, specifically DCT that produce a snippet or fingerprint of various forms of multimedia. The use of Perceptual Hashing is attributed to whether or not two images are similar or not, if yes, then to what extent are they similar. Perceptual hashes must be robust enough to take into account transformations or "attacks" on a given input and yet be flexible enough to distinguish between dissimilar files. Perceptual Hashing is similar to Average Hashing in some regards but the main difference comes in the implementation point of view. Images can be scaled larger or smaller, have different aspect ratios, and even minor coloring differences (contrast, brightness, etc.) and they will still match similar images when applied the Perceptual Hashing algorithm.

Methodology

1.) **Reduce Size** : Like Average Hash, pHash starts with a small image. However, the image is larger than 8x8; 32x32 is a good size. This is really done to simplify the DCT computation and not because it is needed to reduce the high frequencies.
2.) **Reduce Color :** Then the image is reduced to a grayscale format to just further simplify the number of computations.
3.) **Compute DCT :** The DCT separates the image into a collection of frequencies and scalars.
4.) **Reduce the DCT :** The DCT matrix is currently of size 32x32. To simplify things , we just keep the top left 8x8 matrix.
5.) **Compute the average value :** Like the Average Hash, compute the mean DCT value (using only the 8x8 DCT low-frequency values and excluding the first term since the DC coefficient can be significantly different from the other values and will throw off the average).

6.) **Further reduce the DCT :** Set the 64 hash bits to 0 or 1 depending on whether each of the 64 DCT values is above or below the average value. The result doesn't tell us the actual low frequencies; it just tells us the very-rough relative scale of the frequencies to the mean.
7.) **Construct the hash :** Set the 64 bits into a 64-bit integer. The order does not matter, as long as you remember the order. This is the hash which acts as a unique fingerprint to your image.

The source code for the phash algorithm is given below. The code was implemented in python and after the definition of phash and use of the libraries numpy, scipy and PIL . The resulting definition was used in the imagehash library.

Source Code

```
def
phash(image,
hash_size=32):
        image = image.convert("L").resize((hash_size,
    hash_size), Image.ANTIALIAS)
        pixels = numpy.array(image.getdata(),
    dtype=numpy.float).reshape((hash_size, hash_size))
        dct = scipy.fftpack.dct(pixels)
        dctlowfreq = dct[:8, 1:9]
        avg = dctlowfreq.mean()
        diff = dctlowfreq > avg
        return ImageHash(diff)
```

The difference where perceptual hashing works can be evident after working on some of the examples which are modified from the initial image.

Examples and Observations

The initial image was chosen to be a nearly square image of the size 600x574 with reference to nature. The initial image is given below.

Fig. image.jpg

The original image is then altered by using various image editing software and then analysed using average_hash and phash function.

Grayscale Imaging

The image is converted to grayscale image and then is compared with the initial image for similarity.

Fig. imagebw.jpg

The difference between the hashes of the respective images are compared which gives the result that there is no difference between the hashes of the respective images. This is to be expected as the process of average_hash and phash require the initial color image to be already converted to grayscale imaging.

```
Python 2.7.14 Shell
File Edit Shell Debug Options Window Help
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> from PIL import Image
>>> import imagehash
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imagebw.jpg')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
0
```

Fig. Hash Difference Code for Grayscale

Removing colors from Image

The image is converted again by removing the color components from the initial image. The image below has the red and green components removed.

Fig. imageblue.jpg

When the hash difference was observed between the initial and the final image, it was observed that there was no difference between the hashes of the original and the final images. Here is the attached image.

```
Python 2.7.14 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> from PIL import Image
>>> import imagehash
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imageblue.gif')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
0
```

Fig. Hash Difference Code for imageblue.jpg

Similar differences were observed when applied the rest of the filters. But , one noticeable difference was that there was a hash difference of 4 bits when the Sepia filter was applied. Both the hashing functions phash and average_hash produced the same result.

Harsh Image Conditions

Harsh image conditions are the conditions in which the lighting conditions are too bright and the image looks distinctively brighter than the original image. This type of condition is often suited for the photographers. This type of condition produced a special distinction in the image recognition. Below is the attached image under harsh lighting conditions.

Fig. imageharsh.jpg

Here it is interesting to notice that when phash function is applied to the image to give a hash , the hash difference between the new and the old hash comes out to be equal to 10 bits which can very well mean that the images are different ! But in reality only the lighting conditions are different. This perspective is observed better by the average_hash which gives a hash difference of only 2 bits ! The phash difference is illustrated by hash1-hash2 in the below and the average_hash is represented by ahash1-ahash2 in the image below.

Python 2.7.14 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> from PIL import Image
>>> import imagehash
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imageharsh.jpg')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
10
>>> hash1 = imagehash.average_hash(image1)
>>> hash2 = imagehash.average_hash(image2)
>>> hash1-hash2
2
>>> |
```

Fig. Hash Difference Code for imageharsh.jpg

This difference indicates that in the cases of harsh lighting conditions , average_hash seems to work better than the phash. The difference between the two functions was observed as enormous and average_hash produces a better result every time we experimented.

High Contrast Blur Image

The original image was initially blurred and then adjusted to high contrast. The resultant image again gave favourable results in favour of the average_hash. The phash almost dismissed the image as not being of similar characteristics while average_hash gave a favorable score to the comparison. Below is the high contrast blurred image.

Fig. imageblurish.jpg

The code below demonstrates the difference between the conventional average_hash and the phash method. The hash difference is quite evident and distinct.

```
Python 2.7.14 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> import imagehash
>>> from PIL import Image
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imageblurish.jpg')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
14
>>> ahash1 = imagehash.average_hash(image1)
>>> ahash2 = imagehash.average_hash(image2)
>>> ahash1-ahash2
4
```

Fig. Hash Difference Code for imageblurish.jpg

Image Cropping

Image cropping is used almost everywhere from graphic industry to well, this own project of ours. Below is the image which was cropped from the original image.

Fig. imagecrop.jpg

This interestingly produces the result that average_hash function is again better than the phash function! The phash dismisses the cropped image as a completely different image by giving a hash difference of 34, while average_hash gives the bit difference of only 8 bits. The code below elaborates more.

Python 2.7.14 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> import imagehash
>>> from PIL import Image
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imagecrop.jpg')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
34
>>> ahash1 = imagehash.average_hash(image1)
>>> ahash2 = imagehash.average_hash(image2)
>>> ahash1-ahash2
8
>>>
```

Fig. Hash Difference Code for imagecrop.jpg

Rotated Images

There is a big possibility that the image taken by the camera was taken in two different modes , i.e with the landscape mode. Normally both the functions wouldn’t work to identify the image, but there is a possibility of identifying theimage if rotate image function is used. Here is an example of an image rotated by 90 degrees.

Fig. rotateimage.jpg

The functions rotate(img) gave us an estimate to rotate the image by running a for loop. The following image explains the use of the rotate function and its effect on phash.

```
Python 2.7.14 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> from PIL import Image
>>> import imagehash
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\imagerotate.jpg')
>>> image3 = image2.rotate(90)
>>> image4 = image2.rotate(-90)
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash3 = imagehash.phash(image3)
>>> hash4 = imagehash.phash(image4)
>>> hash1-hash2
36
>>> hash1-hash3
2
>>> hash1-hash4
32
```

Fig. Hash Difference Code for rotateimage.jpg

The difference between hash1 and hash3 in the above picture which is only 2 explains a lot that there might be a lot of similarities between them. Both the functions phash and average_hash were observed to be equally effective here.

Gamma Correction

Gamma correction function is a function that maps luminance levels to compensate the non-linear luminance effect of display devices (or sync it to human perceptive bias on brightness). This is often used to correct the abnormalities associated with the image. Below is a new image which was used for gamma correction later.

Fig. gamma.jpg

The image was applied gamma correction from a third party image editor and the resulting image was displayed below.

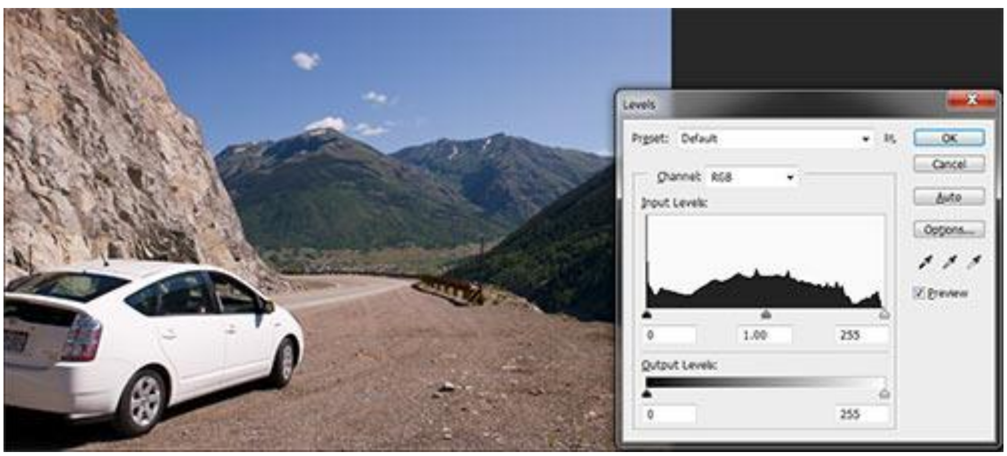


Fig. gammaafter.jpg

This time , after applying both the hashing methods gave us for a fact that phash was actually better at identifying the similarity between the images rather than the average_hash method. Phash method gave us a hash difference of the bits as 4 bits whereas the average_hash method gave us a difference of 8 bits for the after image.

The code is displayed below.

Python 2.7.14 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> import PIL
>>> from PIL import Image
>>> import imagehash
>>> image1 = PIL.Image.open('C:\Users\Administrator\image.jpg')
>>> image1 = PIL.Image.open('C:\Users\Administrator\gamma.jpg')
>>> image2 = PIL.Image.open('C:\Users\Administrator\gammaafter.jpg')
>>> hash1 = imagehash.phash(image1)
>>> hash2 = imagehash.phash(image2)
>>> hash1-hash2
4
>>> ahash1 = imagehash.average_hash(image1)
>>> ahash2 = imagehash.average_hash(image2)
>>> ahash1-ahash2
8
```

Fig. Hash Difference Code for gammaafter.jpg

This gives the result that the phash difference which is hash1-hash2 is 4 bits whereas average_hash difference ahash1-ahash2 is 8 bits.

Conclusion

This project firstly dealt with the intricacies and the methodology used in JPEG compression. Then we applied the processes involved in JPEG compression to identify the duplication of images using various methods of hashing and further analyzed the effects of image editing using the hashing methods phash and average_hash. The source codes of the hashing methods were written with care in python. The efficiencies of both the methods were compared with each other using a plethora of images.

References

1.) Python Official Documentation - PyPi Image Hashing : https://pypi.python.org/pypi/ImageHash

2.) Tyler Genter – "Using JPEG to Compress Still Pictures" (2010): https://mse.redwoods.edu/darnold/math45/laproj/fall2010/tgenter/paperPDFScreen.pdf