



PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

**MAY 2020: IN SEMESTER ASSESSMENT (ISA) B.TECH. IV SEMESTER
UE18MA251- LINEAR ALGEBRA**

MINI PROJECT REPORT

ON

IMAGE COMPRESSION USING SINGULAR VALUE DECOMPOSITION (SVD)

Submitted by

- | | | |
|----|------------------------|--------------------|
| 1. | Name: Sreyans Bothra | SRN: PES1201802012 |
| 2. | Name: Sahith Kurapati | SRN: PES1201800032 |
| 3. | Name: Revanth Babu P N | SRN: PES1201800042 |

Branch & Section : CSE 4A

PROJECT EVALUATION

(For Official Use Only)

Sl.No.	Parameter	Max Marks	Marks Awarded
1	Background & Framing of the problem	4	
2	Approach and Solution	4	
3	References	4	
4	Clarity of the concepts & Creativity	4	
5	Choice of examples and understanding of the topic	4	
6	Presentation of the work	5	
	Total	25	

Name of the Course Instructor:

Signature of the Course Instructor :

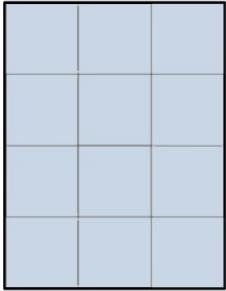
INTRODUCTION:

As the world is developing and digitalising, there is large amounts of data being transferred across the World Wide Web and Internet. A large amount of data to be transmitted is in the form of images, videos and other such visual mediums. This transmission is expensive because of the large storage capacity and bandwidth that it requires. Hence, the consequent need for compression of such data arises so that redundancy is reduced. This reduces the amount of space required to store them and therefore the bandwidth required for transmission is also reduced. Images are basically 2D (/3D) matrices with each element in the matrix representing grayscale values, one for each pixel and colour and videos are basically multiple images grouped together with a specified number of frames i.e. images appearing every second (fps).

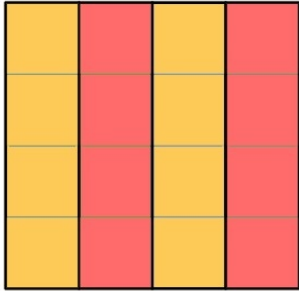
In this context, we have tried to implement a Singular Value Decomposition (SVD factorization) method to compress images and consequently videos.

Singular Value Decomposition (SVD factorization) is a method in which a general ' \mathbf{M} ' matrix is broken into three parts as:

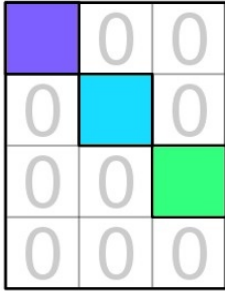
$$\mathbf{M}_{m \times n} = \mathbf{U}_{m \times m} * \mathbf{S}_{m \times n} * \mathbf{V}_{n \times n}^T$$


 \mathbf{M}
 $m \times n$
 (4) (3)

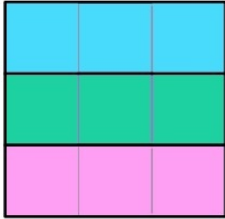
=


 \mathbf{U}
 $m \times m$
 (4) (4)

×

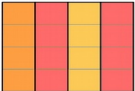

 Σ
 $m \times n$
 (4) (3)

×

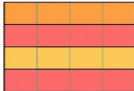

 \mathbf{V}^*
 $n \times n$
 (3) (3)

where:

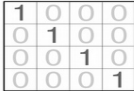
1. \mathbf{U} – It is a unitary matrix of order ' $\mathbf{m} \times \mathbf{m}$ ' consisting of all eigen vectors of $\mathbf{M}^* \mathbf{M}^T$. $\mathbf{M}^* \mathbf{M}^T$ is the product of the matrices ' \mathbf{M} ' and ' \mathbf{M}^T ' (\mathbf{M} transpose- such that if \mathbf{m}_{ij} is an element in \mathbf{A} in the i th row and j th column, it will become \mathbf{m}_{ji} i.e. element at j th row and i th column in \mathbf{M}^T). Eigen vectors of $\mathbf{M}^* \mathbf{M}^T$ are the vectors ' \mathbf{x} ' which satisfy $(\mathbf{M}^* \mathbf{M}^T - \lambda \mathbf{I}) \mathbf{x} = \mathbf{0}$, where ' λ ' is one of the eigen values of $\mathbf{M}^* \mathbf{M}^T$. Picture represents unitary behaviour of \mathbf{U} .


 \mathbf{U}

×


 \mathbf{U}^*

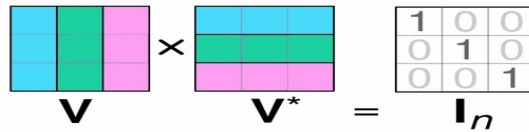
=


 \mathbf{I}_m

2. **S** – also represented by '**Σ**' is a rectangular diagonal matrix of order '**m×n**'. The elements of these matrix are the singular values of M^*M^T . If ' λ ' is an eigen value of M^*M^T then ' $\sigma = \sqrt{\lambda}$ (square root of λ) (only positive value) is called the singular value of M^*M^T . The singular values are stored in a descending order i.e. the highest singular value is stored in the first row first column, 2nd highest in 2nd row 2nd column and so on, i.e.

$$\forall_{(0 < i < n)} S_{ii} \geq S_{((i+1)(i+1))} \quad .$$

3. **V^T** – Is a unitary matrix of order '**n×n**' consisting of all eigen vectors of M^T*M . Picture represents unitary behaviour of V/V^T .



$$\mathbf{V} \times \mathbf{V}^* = \mathbf{I}_n$$

4. **U** and **V^T** are each orthonormal matrices.

The Singular Value Decomposition breaks/decomposes a general matrix into multiple simple matrices. Each such simple matrix is a column vector multiplied with a row vector. An '**m×n**' matrix has **m*n** entries. The product m*n is very large for images. But a column and a row only have m (elements in the Σ matrix)+n components, much lesser than $m*n^{[1]}$. These simple matrices are full size matrices that can be processed with extreme speed—they need only m+n elements.

The basis of using SVD factorization is that there are many pixels in the image which are not independent of each other. Therefore, SVD factorization works properly only if there is redundancy in the image and totally depends on the fact that nearby pixels have similar values. Similarly for videos, we only transmit the small differences between pixels of two consecutive frames/images and that is how videos are compressed

Taking the example of a black and white television- assume we have 24 fps videos being played. Since we are assuming just grayscale videos- we have $8*m*n$ bits to be transmitted across for a perfect copy of the frames. ('m' and 'n' represent the dimensions of the image to be transmitted). Assuming a High Definition (HD) video quality with m=1080 and n=1920 we have $(1080*1920*24*8)$ 39,81,31,200 bits being transmitted every second something that is very expensive to transmit and store and wouldn't be possible by the transmitters to cope up with. And if

we use colour images having 3 components, we multiply the last value by 3 which yields an extremely large value. Therefore the need for having compression arises.

This project represents the need to store and transmit data efficiently. The project aims to reduce redundancy in images and videos and hence make it more efficient to store and transmit them. The scope of the project lies in a wide range. According to recent reports many video streaming and social media companies like Netflix, YouTube, Instagram etc. generate a lot of content everyday which needs to be compressed so that they can be efficiently stored and transmitted and hence reduce the expenses.

LITERATURE SURVEY

In recent years, numerous image compression schemes and their applications in image processing have been proposed. In this section, a brief review of some important contributions from the existing literature is presented.

Awwal et al.^[2] presented a compression technique using SVD factorization and the Wavelet Difference Reduction (WDR). The WDR used for further reduction. This technique has been tested with other techniques such as WDR and JPEG 2000 and gives a better result than these techniques. Furthermore, using WDR with SVD factorization enhance the PSNR and compression ratio.

Adiwijaya et al.^[3] proposed a technique based on Wavelet-SVD factorization, which used a graph colouring technique in the quantization process. This technique worked well and enhanced the PSNR and compression ratio. The generated compression ratio by this work ranged between 50–60%, while the average PSNR ranged between 40–80db.

H. S. Prasanth et al.^[4] have implemented another approach for compression of images using singular value decomposition (SVD factorization). By applying singular value decomposition on the image matrix, compression of image is achieved. This method employs a data compression technique, wherein operations are performed on matrix after the image is decomposed into matrices. Further, it is observed that as rank increases in image matrix, the number of entries will also increase which leads to make picture quality better correspondingly. Due to this, more compression ratio has been achieved by smaller ranks. Following the same objective of image compression using SVD factorization, the most problem is which K rank to use for giving a better image compression. For this reason, the method presented in El Asnaoui et al.^[5], introduces two new approaches: The first one is an improvement of the Block Truncation Coding method that overcomes the disadvantages of the classical Block Truncation Coding, while the second one describes how to obtain a new rank of SVD factorization method, which gives a better image compression.

T. J. Peters et al.^[6] has implemented SVD factorization (singular value decomposition) to compress the microarray image. Huge amounts of DNA information for research purposes are stores as microarray images. These are of high-resolution images which highlights minute details of the image. Because of the high resolution, these images tend to be larger in size, which means storage on the hard disk requires lot of space. So, it is very important to reduce the size of the image without compromising the quality or compromising the amount of detail present in the image. This calls for comparatively complicated process, where in microarray images need to be clustered and classified before selecting the features. SVD factorization can be used here to divide the image into small sub -images and on each sub-image SVD factorization is performed. This method gives a better high peak signal to noise ratio in addition to increasing compression ratio.

REPORT ON THE PRESENT INVESTIGATION

1) Converting Images to their Equivalent Matrix form

The images to be compressed are converted to their respective equivalent pixel matrices using **Python's Scikit-Learn** library's function: '**img_as_float**' which converts the image into an array of floating-type data points. We then convert all these into Gray images by taking average of 'rgb' pixel values or keep them in the same form.

2) Computing Singular Value Decomposition (SVD factorization) of a given matrix

Singular Value Decomposition (SVD factorization) is a process by which we can compress images. SVD factorization is performed on the images whose matrix equivalent has been computed using **Python's numpy library's numpy.linalg.svd**. This implementation is based on **LAPACK (Linear Algebra PACKage)** routine **_gesdd** and **_dgesdd**.^[7] DGESDD is based on QR iteration^[8] to find Σ (or S) and computes U and V^T using a divide and conquer approach. The divide and conquer approach is based on **Golub-Kahan bidiagonalization**^[9] which uses placeholders to find the left and right vectors, i.e. U and V^T matrices.

3) Compression

As discussed, many images have redundancy which makes them occupy more space. This redundancy is in the form of non-independent pixel values. Non-independent pixel values are those which have very similar adjacent pixel values and hence very small differences in pixel values in adjacent pixels. So removing redundancy means that we try to minimise the number of such pixels in the image.

Through our observations we observed that the ' Σ (or S)' matrix has 'm' non-zero (diagonal) values.

Out of these ‘m’ values (which are arranged in descending order), we found that only a few of these singular values are responsible for constructing a similar quality image. These number of singular values is represented by a variable ‘k’. This variable ‘k’ represents the number of vectors in each of the three matrices U, Σ (or S) and V^T that will be chosen. It is basically the number of singular values used in the Σ (or S) matrix.

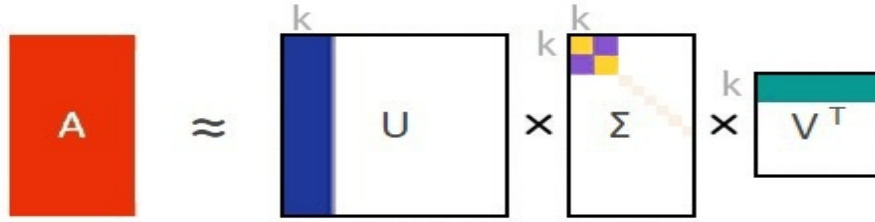
From the ‘U’ matrix calculated from the previous step, we choose all the rows but only the first ‘k’ **columns** from each row.

From the ‘ Σ (or S)’ matrix calculated from the previous step, we choose the first ‘kxk’ matrix from the ‘mxn’ matrix.

From the ‘ V^T ’ matrix calculated from the previous step, we choose the first ‘k’ **rows** of the matrix.

4)Generation of New Image

The matrices generated from the previous steps are multiplied with each other making a new image.



$$A_{m \times n} \approx U_{m \times k} \cdot \Sigma_{k \times k} \cdot V_{k \times n}^T$$

Depending on the value of ‘k’ the quality of the images changes. As ‘k’ increases, the quality of the compressed image increases because more pixels of the original image are used and hence more data points are used.

5)Compression Ratio and Space Saving Ratio

From the previous step, we see that the new ‘U’ matrix has ‘mxk’ elements. The new ‘ V^T ’ matrix has ‘kxn’ matrix. The resized ‘ Σ (or S)’ matrix has ‘kxk’ elements but it is also a diagonal matrix, therefore, we need to store only ‘k’ (along principal diagonal) elements because other ‘k²-k’ elements are 0. Therefore, we only need to store and transmit **k*m+k+k*n** elements.

Compression Ratio, which is defined as the ratio of the size of the Uncompressed Image to the size of the Compressed Image, is therefore:

$$\text{Compression Ratio} = \frac{\text{Size of uncompressed image}}{\text{Size of compressed Image}}$$

$$\text{Compression Ratio} = \frac{m * n}{k * (m + n + 1)}$$

where,

m, n – represent the dimensions of the image

k – represents the number of vectors used.

Space Saving Ratio is the ratio of the difference between the space taken by the uncompressed image and the compressed image over the space taken by the uncompressed image.

$$\text{Space Saving Ratio} = 1 - \frac{\text{Space taken by compressed image}}{\text{Space taken by uncompressed image}}$$

$$\text{Space Saving Ratio} = 1 - \frac{1}{\text{Compression Ratio}}$$

$$\text{Space Saving Ratio} = \frac{(m*n) - k*(m+n+1)}{m*n}$$

6) Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR)

The Mean Squared Error (MSE) and Peak Signal to Noise Ratio (PSNR) are used to represent the difference in the quality of the two – uncompressed and compressed images.

The MSE is the mean of the sum of the squared error between the compressed image and the uncompressed image.

$$MSE = \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n (C_{ij} - U_{ij})^2$$

where,

C -Compressed Image matrix

U -Uncompressed Image matrix

PSNR represents a measure of the peak error. It is a log based value just like noise and is also measured in decibels(db).

$$PSNR = 10 \log_{10} \left(\frac{MAX_i^2}{MSE} \right)$$

where,

MAX_i – represents the maximum possible pixel value i.e. 2^B-1 where B is the number of bits for each pixel. For grayscale: B=8 and MAX_i = 255

RESULTS

The Singular Value Decomposition method helps compress the image by reducing redundancy in the image.

Meaning of Values of Certain Terms:

The **Compression Ratio** should be as **high** as possible because it then shows that the uncompressed image is occupying more space compared to the compressed image.

The **Space-Saving Ratio** should be **positive** and **as close to 1 as possible**. Being close to 1 represents that the compressed image occupies very less space compared to the uncompressed image. But, if the ratio is negative it means that the compressed image occupies more space than the uncompressed image.

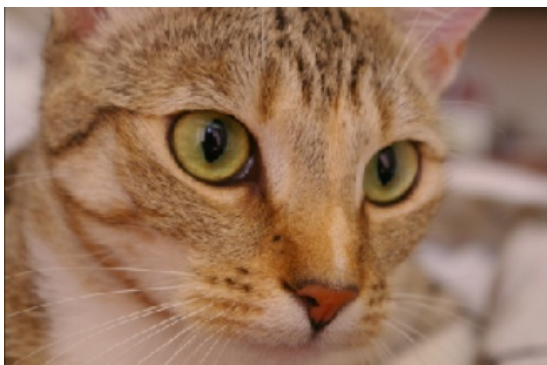
The **Mean Squared Error (MSE)** should be **low** thus indicating that the visual degradation is less in the compressed image.

The **Peak Signal to Mean Ratio (PSNR)** is inversely proportional to the MSE, therefore, has to be **high** thereby indicating less visual degradation in the compressed image. If it decreases, it represents that the compressed image is noisy and is degraded compared to the original uncompressed image.

EXPERIMENTATION:

1.) We started our experimentation by **converting the colour image into a gray-scale** (black and white) image and then compressing it.

Using a Cat's^[10] image:

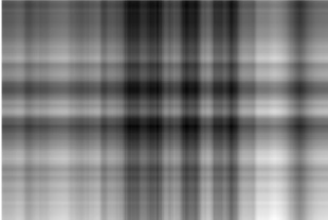







Original Image



Original Image Converted to Grayscale

Tabulated Results:

Sl.No	Compressed Image	'k'	Compression Ratio	Space Saving Ratio	Mean Squared Error (MSE)	Peak Signal to Noise Ratio (PSNR)
1		1	179.920	0.994	1.30×10^{-2}	66.975
2		10	17.992	0.944	2.76×10^{-3}	73.714
3		25	7.197	0.861	1.00×10^{-3}	78.110
4		50	3.598	0.722	4.06×10^{-4}	82.042
5		100	1.799	0.444	1.07×10^{-4}	87.808

6		200	0.899	-0.111	$7.32 \cdot 10^{-6}$	99.488
---	---	-----	-------	--------	----------------------	--------

CODE:

```

from skimage import data
from skimage.color import rgb2gray
from numpy.linalg import svd
from skimage import img_as_ubyte, img_as_float
gray_images = {
    "cat": rgb2gray(img_as_float(data.chelsea())),
    "astro": rgb2gray(img_as_float(data.astronaut())),
}

def compress_svd(image, k):
    """
    Perform svd decomposition and truncated (using k singular values/vectors) reconstruction
    returns
    -----
    reconstructed matrix reconst_matrix, array of singular values s
    """
    plt.imshow(image, cmap='gray')
    U, s, V = svd(image, full_matrices=False)
    #U[:, :k] all rows but first k columns
    #s[:k] all k*k matrices
    #V[:, :k] all columns for first k rows
    reconst_matrix = np.dot(U[:, :k], np.dot(np.diag(s[:k]), V[:, :k]))
    return reconst_matrix, s

def mean_squared_error(im1, im2):
    sq_diff = (im1 - im2) ** 2
    s = np.sum(sq_diff)
    mse = s / (np.size(im1, 0) * np.size(im1, 1))
    return mse

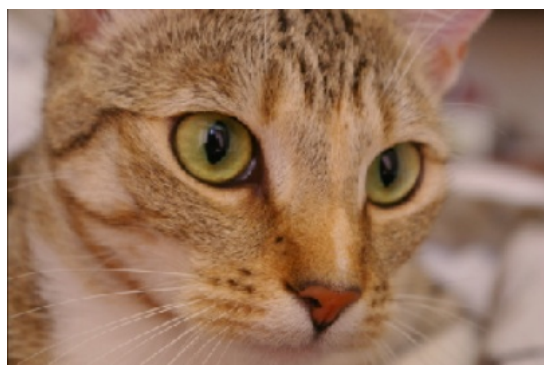
```

```
def compress_show_gray_images(img_name,k):
    """
    compresses gray scale images and display the reconstructed image.
    Also displays a plot of singular values
    """
    image=gray_images[img_name]
    original_shape = image.shape
    print(original_shape)
    reconst_img,s = compress_svd(image,k)
    fig,axes = plt.subplots(1,2,figsize=(8,5))
    axes[0].plot(s)
    #print(s)
    ko=(original_shape[0]*original_shape[1])/(k*(original_shape[0] + original_shape[1])+k)
    compression_ratio =ko
    space_saving=1-1/compression_ratio
    mse=mean_squared_error(image,reconst_img)
    psnr=10*log10(255*255/mse)
    s="compression ratio= "+str(compression_ratio)+"\nspace_saving= "+str(space_saving)
    +"\nmse"+str(mse)+"\npsnr"+str(psnr)
    axes[1].set_title(s)
    axes[1].imshow(reconst_img,cmap='gray')
    axes[1].axis('off')
    fig.tight_layout()

    interact(compress_show_gray_images,img_name=list(gray_images.keys()),k=(1,300));
```


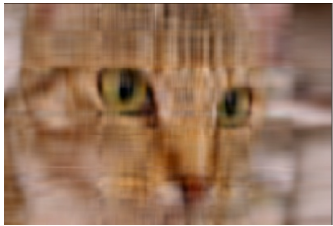
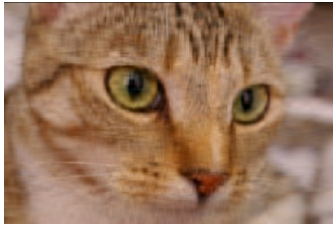
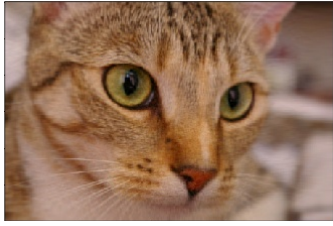

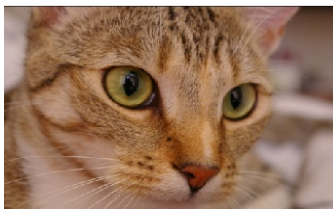
2.) Compressing a Colour Image using Reshape method – This method involves averaging the third dimension and involves flattening the third dimension to make it a two dimension array.

Using the same **Cat's** image:



Original Image

Tabulated Results:

Sl.No	Compressed Image	'k'	Compression Ratio	Space Saving Ratio	Mean Squared Error (MSE)	Peak Signal to Noise Ratio (PSNR)
1		1	179.920	0.994	4.22×10^{-2}	158.237
2		10	17.992	0.944	8.88×10^{-3}	165.008
3		25	7.197	0.861	3.24×10^{-3}	169.385
4		50	3.598	0.722	1.37×10^{-3}	173.129
5		100	1.799	0.444	4.07×10^{-4}	178.396
6		200	0.899	-0.111	4.26×10^{-5}	188.200

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from math import log10
get_ipython().run_line_magic('matplotlib', 'inline')

from ipywidgets import interact, interactive, interact_manual
from skimage import data
from skimage.color import rgb2gray
from numpy.linalg import svd
from skimage import img_as_ubyte, img_as_float
color_images={
    "cat":img_as_float(data.chelsea()),
    "astro":img_as_float(data.astronaut()),
    "coffee":img_as_float(data.coffee())
}

def compress_svd(image,k):
    """
    Perform svd decomposition and truncated (using k singular values/vectors) reconstruction
    returns
    -----
    reconstructed matrix reconst_matrix, array of singular values s
    """

    plt.imshow(image,cmap='gray')
    U,s,V = svd(image,full_matrices=False)
    #U[:,k] all rows but first k columns
    #s[k] all k*k matrices
    #V[k,:] all coulmsns for first k rows
    reconst_matrix = np.dot(U[:,k],np.dot(np.diag(s[k]),V[k,:]))
    return reconst_matrix,s

def compress_col_images(img_name,k):
```

```

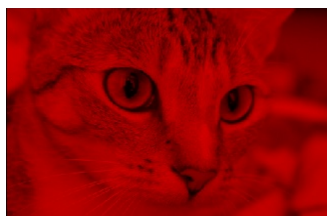
image=color_images[img_name]
original_shape=image.shape
print(original_shape)
image_reshaped=image.reshape((original_shape[0],original_shape[1]*3))
image_reconst,s=compress_svd(image_reshaped,k)
image_reconst=image_reconst.reshape(original_shape)
ko=(original_shape[0]*original_shape[1])/(k*(original_shape[0] + original_shape[1]+1))
compression_ratio =ko
mse=mean_squared_error(image,image_reconst)
value=log10(16777215*16777215)
psnr=10*(value-log10(mse))
space_saving=1-1/compression_ratio
print(compression_ratio,space_saving,mse,psnr)
plt.title("space_saving "+str(space_saving))
plt.imshow(image_reconst)
interact(compress_col_images,img_name=list(color_images.keys()),k=(1,300))

```

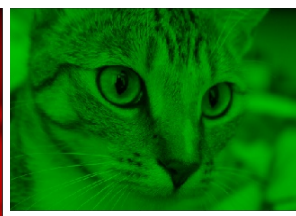
3.) Compressing a Colour Image using Layers Method – The given **cat's** image is converted into three component parts – Red(R), Green(G) and Blue(B) parts/layers.



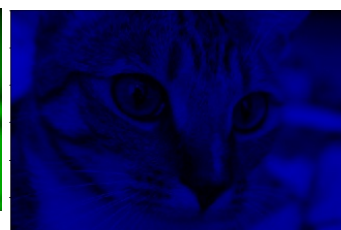
Original Image



The red equivalent



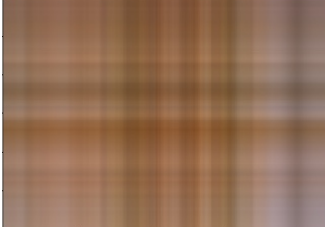





The green equivalent



The blue equivalent

Then we perform SVD factorization for each of these images separately and then combine the computed layers (using Python's numpy's library for matrix manipulation) together to create the required color image.

Tabulated Results:

Sl.No	Compressed Image	'k'	Compression Ratio	Space Saving Ratio	Mean Squared Error (MSE)	Peak Signal to Noise Ratio (PSNR)
1		1	179.920	0.994	294.36	23.442
2		10	17.992	0.944	193.56	25.262
3		25	7.197	0.861	117.87	27.417
4		50	3.598	0.722	66.86	29.879
5		100	1.799	0.444	21.72	34.761
6		200	0.899	-0.111	1.79	45.600

CODE:

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import skimage
from skimage import img_as_ubyte
from skimage import data
import warnings; warnings.simplefilter('ignore')
def mean_squared_error(im1, im2):
    sq_diff = (im1 - im2) ** 2
    s = np.sum(sq_diff)
    mse = s / (np.size(im1,0) * np.size(im1,1))
    return mse
im = skimage.img_as_float(data.chelsea());
R = im[:, :, 0]
G = im[:, :, 1]
B = im[:, :, 2]
Ur, Sr, Vr = np.linalg.svd(R, full_matrices=True)
Ug, Sg, Vg = np.linalg.svd(G, full_matrices=True)
Ub, Sb, Vb = np.linalg.svd(B, full_matrices=True)
plt.imshow(im)
k = 2
Ur_k = Ur[:, 0:k]
Vr_k = Vr[0:k, :]
Ug_k = Ug[:, 0:k]
Vg_k = Vg[0:k, :]
Ub_k = Ub[:, 0:k]
Vb_k = Vb[0:k, :]
Sr_k = Sr[0:k]
Sg_k = Sg[0:k]
Sb_k = Sb[0:k]
im_r = np.dot(Ur_k, np.dot(np.diag(Sr_k), Vr_k))
im_g = np.dot(Ug_k, np.dot(np.diag(Sg_k), Vg_k))
im_b = np.dot(Ub_k, np.dot(np.diag(Sb_k), Vb_k))
```



```

im_reconstr = np.zeros((300,451,3))
im_reconstr[:, :, 0] = im_r
im_reconstr[:, :, 1] = im_g
im_reconstr[:, :, 2] = im_b
im_reconstr[im_reconstr < 0] = 0
im_reconstr[im_reconstr > 1] = 1
plt.imshow(im_reconstr)
original_bytes = im.nbytes
mse = []
psnr = []
k_values = []
compression_ratio = []
for i in range(2,21,2):
    k = i
    Ur_k = Ur[:,0:k]
    Vr_k = Vr[0:k,:]
    Ug_k = Ug[:,0:k]
    Vg_k = Vg[0:k,:]
    Ub_k = Ub[:,0:k]
    Vb_k = Vb[0:k,:]
    Sr_k = Sr[0:k]
    Sg_k = Sg[0:k]
    Sb_k = Sb[0:k]
    im_r = np.dot(Ur_k,np.dot(np.diag(Sr_k),Vr_k))
    im_g = np.dot(Ug_k,np.dot(np.diag(Sg_k),Vg_k))
    im_b = np.dot(Ub_k,np.dot(np.diag(Sb_k),Vb_k))
    im_reconstr = np.zeros((300,451,3))
    im_reconstr[:, :, 0] = im_r
    im_reconstr[:, :, 1] = im_g
    im_reconstr[:, :, 2] = im_b
    im_reconstr[im_reconstr < 0] = 0
    im_reconstr[im_reconstr > 1] = 1
    compressed_bytes = sum([matrix.nbytes for matrix in
[Ur_k,Sr_k,Vr_k,Ug_k,Sg_k,Vg_k,Ub_k,Sb_k,Vb_k]])

```

```

k_values.append(k)
err_mse = mean_squared_error(img_as_ubyte(im),img_as_ubyte(im_reconstr))
err_psnr = 10 * np.log10((255 ** 2) /err_mse)
mse.append(err_mse)
psnr.append(err_psnr)
compression_ratio.append(original_bytes / compressed_bytes)
for i in range(25,151,25):
    k = i
    Ur_k = Ur[:,0:k]
    Vr_k = Vr[0:k,:]
    Ug_k = Ug[:,0:k]
    Vg_k = Vg[0:k,:]
    Ub_k = Ub[:,0:k]
    Vb_k = Vb[0:k,:]
    Sr_k = Sr[0:k]
    Sg_k = Sg[0:k]
    Sb_k = Sb[0:k]
    im_r = np.dot(Ur_k,np.dot(np.diag(Sr_k),Vr_k))
    im_g = np.dot(Ug_k,np.dot(np.diag(Sg_k),Vg_k))
    im_b = np.dot(Ub_k,np.dot(np.diag(Sb_k),Vb_k))
    im_reconstr = np.zeros((300,451,3))
    im_reconstr[:,:,0] = im_r
    im_reconstr[:,:,1] = im_g
    im_reconstr[:,:,2] = im_b
    im_reconstr[im_reconstr < 0] = 0
    im_reconstr[im_reconstr > 1] = 1
    compressed_bytes = sum([matrix.nbytes for matrix in
[Ur_k,Sr_k,Vr_k,Ug_k,Sg_k,Vg_k,Ub_k,Sb_k,Vb_k]])
    k_values.append(k)
    err_mse = mean_squared_error(img_as_ubyte(im),img_as_ubyte(im_reconstr))
    err_psnr = 10 * np.log10((255 ** 2) /err_mse)
    mse.append(err_mse)
    psnr.append(err_psnr)
    compression_ratio.append(original_bytes / compressed_bytes)

```

```

import matplotlib.image as img
img.imsave('ori.png',im)
img.imsave('comp.png',im_reconstr)
plt.subplots(figsize=(14,10))
plt.plot(k_values,mse)
plt.xlabel('Number of singular values used',fontsize = 12)
plt.xticks(np.arange(0,151,step = 10))
# plt.yticks(np.arange(0,141,step = 10))
plt.ylabel('Mean Squared error between original image and compresseds image',fontsize =
12)
plt.grid()
plt.subplots(figsize=(12,8))
plt.plot(k_values,psnr)
plt.xlabel('Number of singular values used',fontsize = 12)
plt.ylabel('PSNR',fontsize = 12)
plt.grid()
plt.subplots(figsize=(12,8))
plt.plot(k_values[1:],compression_ratio[1:])
plt.xlabel('Number of singular values used',fontsize = 12)
plt.ylabel('Compression Ratio',fontsize = 12)
plt.grid()

```

CONCLUSIONS

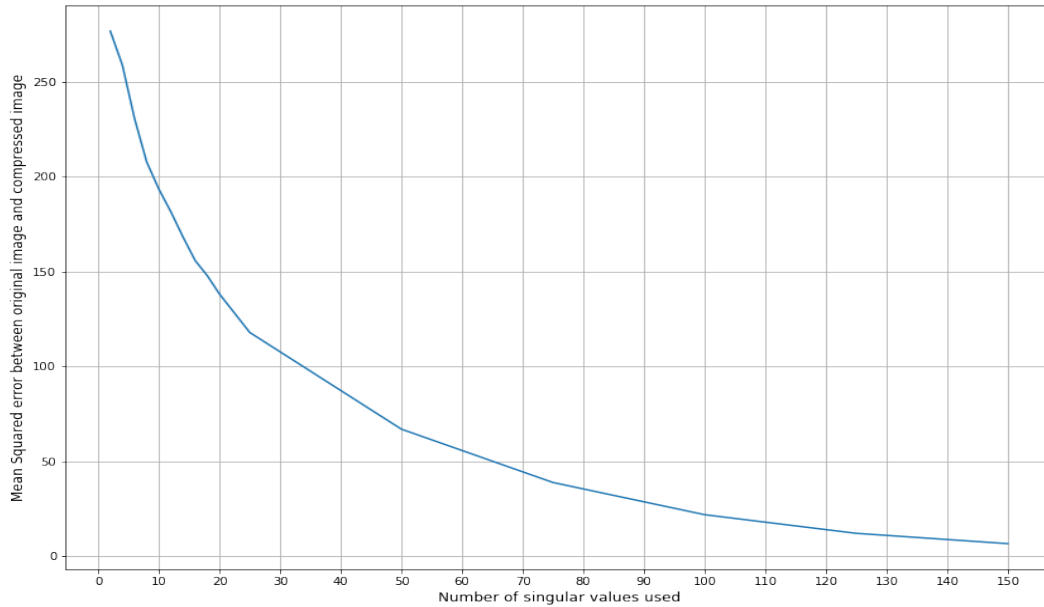
1. Finding Optimal Value of ‘k’ (number of singular values used):

El Asnaoui et al.^[5] had suggested that the value of $k = \frac{m*n}{m+n+1}$. We found that though the Mean Squared Error (MSE) is low and Peak Noise to Signal Ratio (PSNR) is high at such ‘k’, the Compression Ratio at such ‘k’ is not quite high and hence the compressed image occupies more space.

We found through our research that even at half of the ‘k’ value i.e. $k = \frac{m*n}{2*(m+n+1)}$, the image generated had a good quality (low MSE) and also occupied very less space (higher compression ratio). Therefore, we suggest the use of such ‘k’ for image compression. For videos, as well this concept works thereby reducing the size of the video and hence better storage and transmission at less bandwidth.

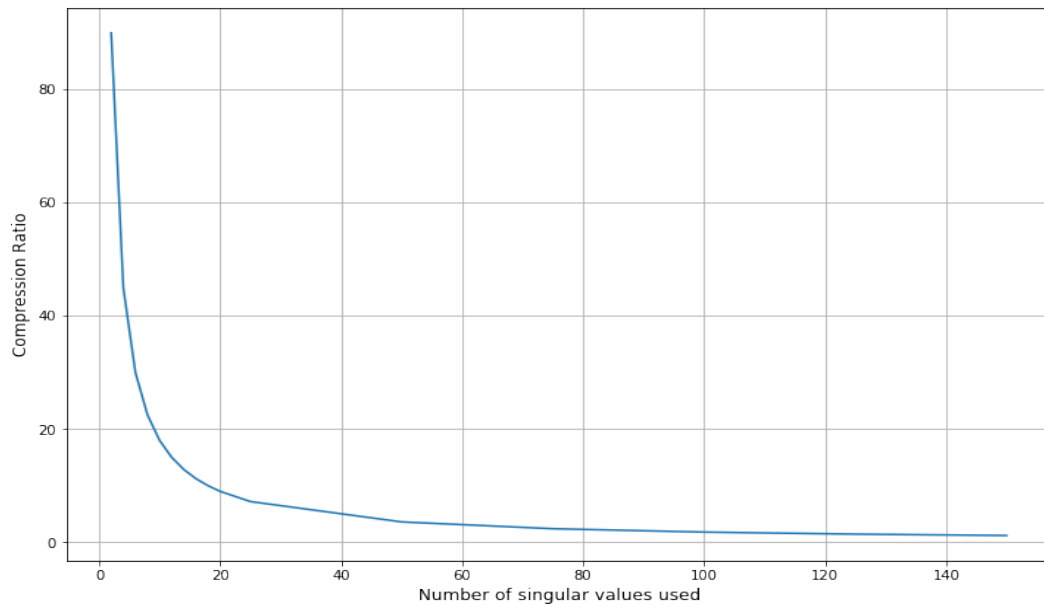
2. Verifications of Results explained in the Results part:

i. Variation of Mean Square Error(MSE) against 'k'



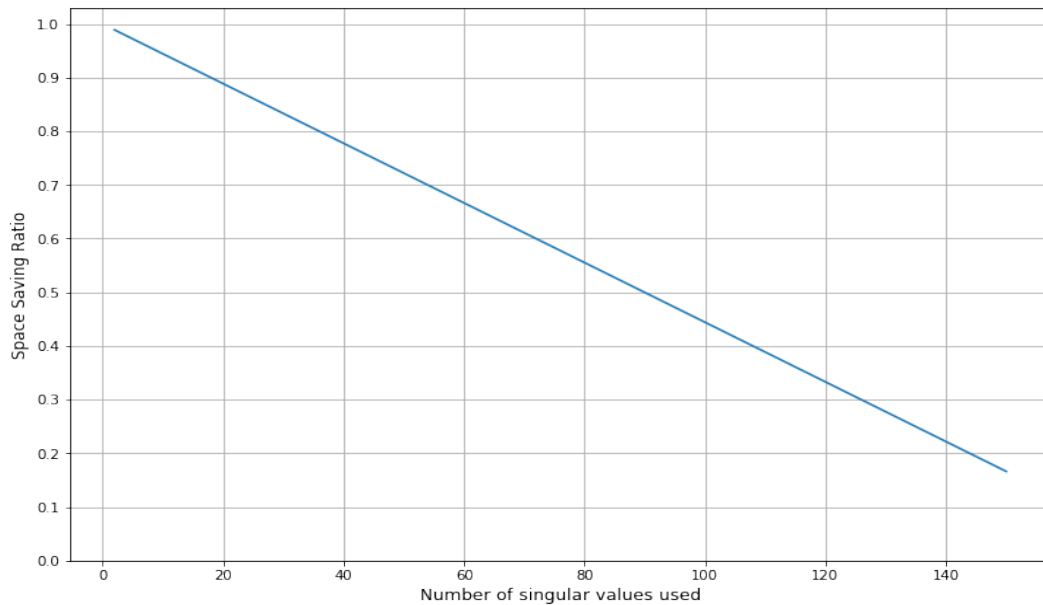
As seen from the above graph, as 'k' increases the Mean Squared Error (MSE) decreases, which represents the fact that greater the number of singular values used, greater is the quality (as the MSE is less).

ii. Variation of Compression Ratio against 'k'(number of singular values used)



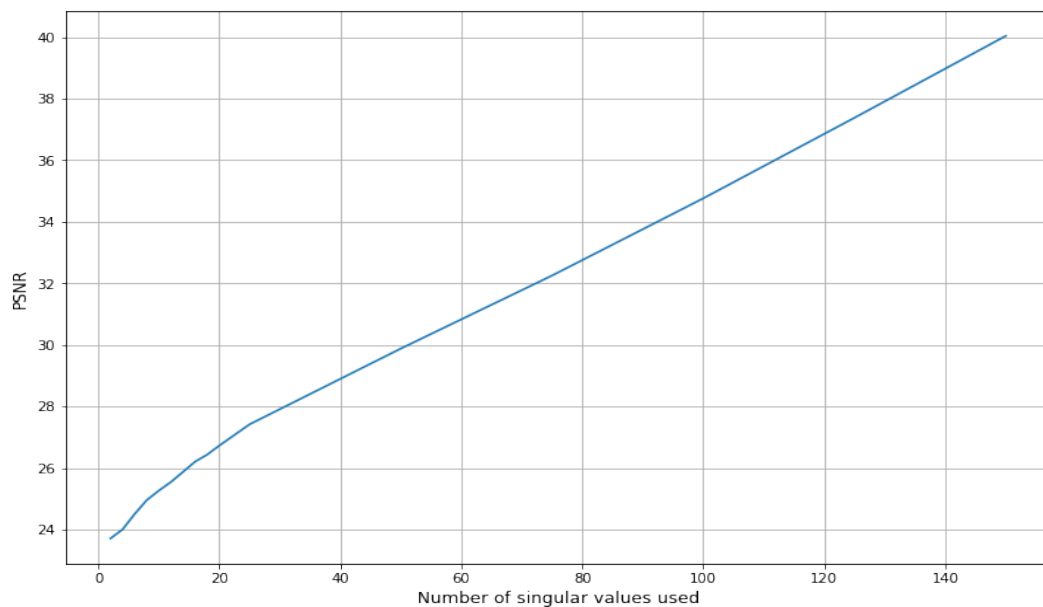
As seen from the above graph, as 'k' increases the compression ratio decreases, which represents the fact that greater the number of singular values used (i.e. greater quality), lesser can the compression can be.

iii. Variation of Space Saving Ratio against 'k'



As seen from the above graph, as 'k' increases the space-saving ratio decreases, which represents the fact that greater the number of singular values used (i.e. greater quality), lesser can the compression can be.

iv. Variation of Peak Signal to Noise Ratio (PSNR) against 'k'



As seen from the above graph, as 'k' increases the PSNR increases, which represents the fact that greater the number of singular values used, greater is the quality (as the visual degradation will be low).

BIBLIOGRAPHY

- [1] Gilbert Strang: Linear Algebra and its Applications, 5th edition, chapter 7, 364 – 369.
- [2] Awwal M.R, Anbarjafari G, Demirel H: Lossy image compression using singular value decomposition and wavelet difference reduction. Digit. Signal Process. 24, 117–123 (2014).
- [3] Adiwijaya, M. Maharani, B. K. Dewi, F. A. Yulianto, B. Purnama: Digital image. compression using graph coloring quantization based on wavelet SVD. J. Phys. Conf. Ser. 423 (2013).
- [4] Prasantha H S and B M K N Image compression using SVD 143–145 (2007).
- [5] El Asnaoui, K., Chawki, Y.: Two new methods for image compression. Int. J. Imaging Robot.15(4), 1–11 (2015).
- [6] Internationale C, T J Peters, Smolíková-wachowiak R and Wachowiak M P, Microarray Image Compression Using a Variation of Singular Value Decomposition 1176–1179 (2007).
- [7] The link for the LAPACK library's _gesdd and _dgesdd routine is present in the link:
http://www.netlib.org/lapack/explore-html/d1/d7e/group__double_g_esing_gad8e0f1c83a78d3d4858eaaa88a1c5ab1.html
- [8] David S Watkins, Understanding the QR Algorithm(Part 1 and Part 2) (1982).
- [9] Lloyd N Trefethen, David Bau, Numerical Linear Algebra, SIAM, Lecture 31, 236-237 (1997)
- [10] The cat image present in Python's scipy library present as data.chelsea() having dimensions (300, 451, 3) for the colour image and (300,451) for the grayscale image.