Image Compression with Singular Value Decomposition

Supervisor:
Dr. Tsui-Wei Weng

Posted:
Dec 6, 2022

Course:
DSC 210 FA'22 Numerical Linear Algebra

Team:

Yashi Shukla
PID: A59020018
Data Science

Sagarika Sardesai
PID: A59020049
Data Science

Shreya Reddy Pakala
PID: A59019863
Data Science

Sai Kaushik Soma
PID: A59020013
Data Science

1. Introduction

Image compression is the process of minimizing the image size in bytes without degrading image quality and is of 2 types - Lossy and Lossless.

In Lossy compression, non-critical and redundant data is permanently removed. Lossy is generally used where information loss is acceptable and algorithms like Discrete Cosine Transform and Singular Value Decomposition fall under this. The drawbacks of this method include image degradation after too much of compression and the inability to reconstruct the original image from the compressed one.

In Lossless compression, images are compressed without removing critical data or reducing image quality. The original image can be reconstructed from the compressed image. Lossless is applied where image quality is crucial and algorithms like Huffman Coding, Arithmetic encoding fall under this. A major drawback of this is that the compressed image size is still quite large.

1.1 History/Background:

The Singular Value Decomposition (SVD) was discovered and developed independently by a number of mathematicians. Eugenio Beltrami and Camille Jordan were the first to do so, in 1873 and 1874, respectively; they were followed by James Joseph Sylvester, Erhard Schmidt, and Hermann Weyl, among others. The first proof for rectangular and complex matrices was given by Carl Eckart and Gale J. Young in 1936, and methods for computing the SVD of a matrix continued to be refined throughout the mid-20th century, revolutionizing the field of numerical linear algebra

The Discrete Cosine Transform (DCT) was first conceived by Nasir Ahmed, T. Natarajan, and K. R. Rao while working at Kansas State University, and he proposed the concept to the National Science Foundation in 1972. He originally intended DCT for image compression

Ahmed developed a practical DCT algorithm with his Ph.D. students T. Raj Natarajan, Wills Dietrich, and Jeremy Fries, and his

friend Dr. K. R. Rao at the University of Texas at Arlington in 1973, and they found that it was the most efficient algorithm for image compression. They presented their results in a January 1974 paper, titled *Discrete Cosine Transform.*

1.2 Applications:

Image compression is used extensively in :

- Pattern recognition
- Computer/Machine Vision
- Video Processing

Some of the real-world applications that utilize image compression involve:

- Websites
- Digital Photography
- Image editing software
- Health industry
- Defense and Security
- and many more.

1.3 State-of-the-art:

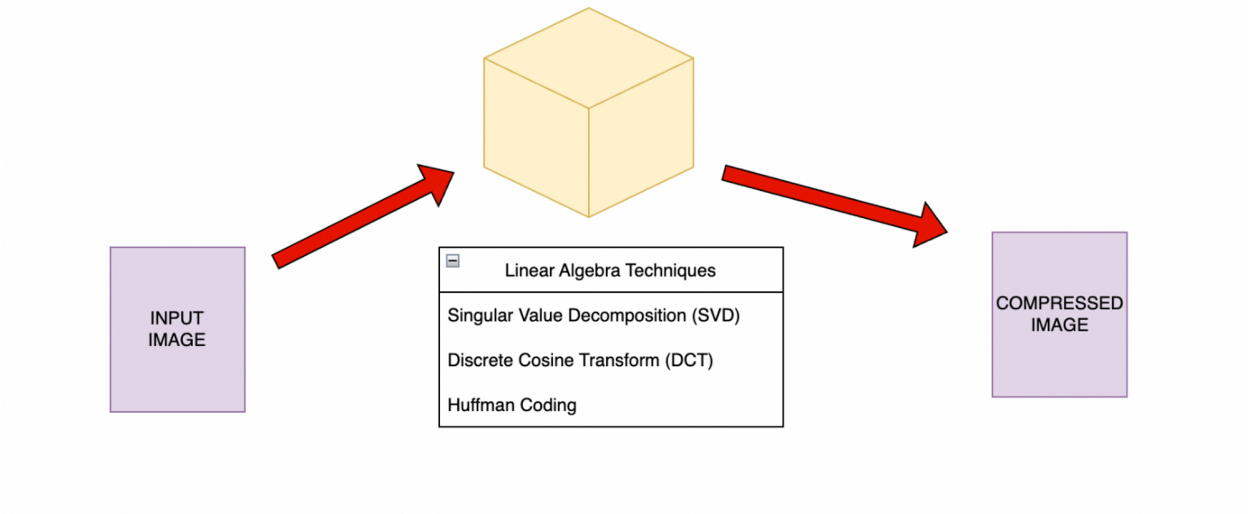
In this report, we study two approaches for Image Compression:

- Singular Value Decomposition (SVD)
- Discrete Cosine Transform (DCT)

In this report, we discuss the use of **Singular Value Decomposition (SVD)** as our first approach. However, we highlight the simplicity of SVD and its drawbacks and limited performance. Thus we discuss about one of the state of the art techniques **Discrete Cosine Transform (DCT)** which is one of the popular image compression algorithms even today.

2. Problem Formulation

2.1 Relation to numerical linear algebra:



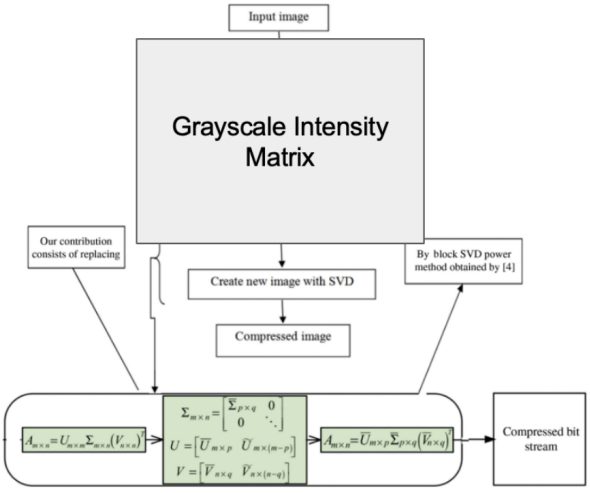
Every image consists of a matrix of pixels which is passed through one of the Linear Algebra Techniques and then we obtain a compressed image. Techniques like Singular Value Decomposition (SVD), Discrete Cosine Transform (DCT), and Huffman Coding are all built using Numerical Linear Algebra whose functionalities are utilized for image compression.

2.2 Approach description:

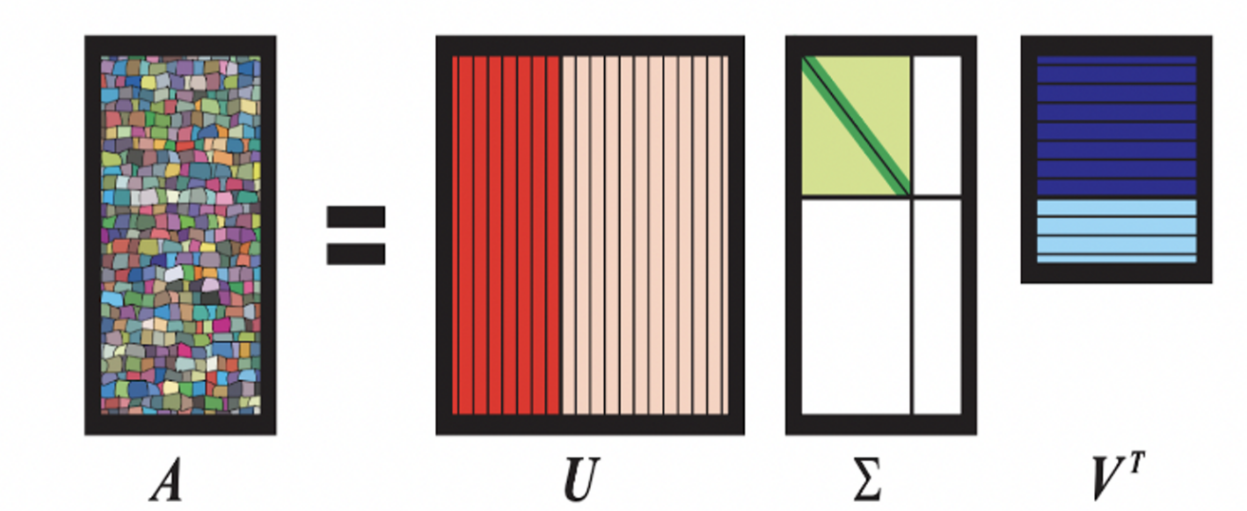
Singular value decomposition is a factorization of a real or complex matrix. SVD aims to approximate a dataset of a large number of dimensions using fewer dimensions. SVD considers highly variable, high dimensional data points and exposes the substructure of the original data by
reducing the higher dimensional data into lower dimensional data. Exposure of the substructure orders the data from the most variation to the least. It generalizes the eigen decomposition of a square normal matrix with an orthonormal eigen basis to any $m \times n$ matrix. It is represented by the equation:

$$\underset{(m\times n)}{A} = \underset{(m\times m)}{U} \quad \underset{(m\times n)}{\sum} \quad \underset{(n\times n)}{V}^{T}$$

- Let A be an $m \times n$ matrix with rank r. Then there exists an $m \times n$ matrix $\sum$ for which the diagonal entries are the first r singular values of A where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > 0$, an $m \times m$ orthogonal matrix U and an $n \times n$ orthogonal matrix V.
- Calculation of the SVD consists of finding the eigenvalues and eigenvectors of AA^T and A^TA.
  The eigenvectors of A^TA make up the columns of V, the eigenvectors of AA^T make up the columns of U. Also, the singular values in $\sum$ are square roots of eigenvalues from AA^T or A^TA.
- The singular values are the diagonal entries of the $\sum$ matrix and are arranged in descending order ($\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r$). The singular values are always real numbers. If matrix A is a real matrix, then U and V are also real.



First, the input image is converted into a grayscale intensity matrix. On each of the input images, SVD is computed where the image undergoes SVD matrix factorization. This can be seen in the figure given below where SVD refactors the given image into matrices. Singular values are used to refactor the image and at the end of this process, the image is represented with a smaller set of values, hence reducing the storage space required by the image. The color gradient in each of the matrices as seen in the image signifies the singular values.



2.3 SOTA Approach description:

The state-of-the-art technology that we will be implementing and comparing the SVD approach with, is Discrete Cosine Transform (DCT). Like SVD, DCT is also a lossy compression and is generally used to compress JPEG Images.

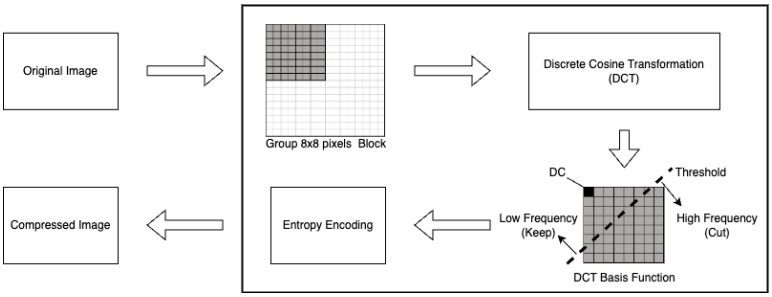
SOTA Approach description:

DCT in itself is a fast computing Fourier Transform that maps the signals to corresponding values in the frequency domain. It transforms them into a domain in which they can be more efficiently encoded.

This algorithm is widely used in image compression. For multichannel images, we apply DCT individually to every channel.

Steps for implementing DCT for image compression:

- DCT breaks the image into n x n blocks, usually the value of n=8 according to JPEG standards.
- DCT is then applied to each of the blocks serially; the values which fall under the frequency domain are kept and other values are cut down.
- Lastly, each block goes through entropy encoding and we get our compressed image.



2.4 Why SOTA?

For compression rates up to 30%, both SVD and DCT yield no visual difference in the compressed image. But DCT is a more acceptable and widely used technique in image compression compared to SVD for the following reasons:

- DCT spreads its loss throughout the image evenly unlike SVD where we can actually lose an entire vector of data at a time.

- As DCT is a block-based approach, it takes less computing memory while processing the images. But SVD works on the whole image at once and needs more memory.
- For higher compression ratios, DCT provides a less distorted image as compared to SVD.

2.4 Evaluation:

The performance metrics that we'll be using are described below:

- **Compression Ratio:** Compression Ratio is the ratio between the sizes of the original image and the compressed image. Higher the compression ratio, the smaller the size but the lower the quality of the compressed image.

$$Compression\ Ratio\ =\ \frac{Size\ of\ the\ Original\ Image}{Size\ of\ the\ Compressed\ Image}$$

```
def compressionratio(original_img_arr, compressed_img_arr):
# original image size
    cv2.imwrite(f"4kImages/original_temp.jpg", original_img_arr)
    original_size = round(os.stat("4kImages/original_temp.jpg").st_size)
# compressed image size
    cv2.imwrite(f"4kImages/compressed_temp.jpg", compressed_img_arr)
    compressed_size = round(os.stat("4kImages/compressed_temp.jpg").st_size)
# computing the performance metric compression ratio
    compression_ratio = original_size/compressed_size
    return round(compression_ratio,2),original_size,compressed_size
```

- **Mean Squared Error (MSE):** Mean Squared Error is the average of squared errors where the error is the difference between each corresponding pixel in the actual and compressed image. The lower the MSE, the closer the compressed image is to the original image.

$$MSE\ between\ two\ images\ I(x,y)\ and\ I'(x,y):$$
$$MSE\ =\ \frac{1}{M\times N}\sum_{y=1}^{M}\sum_{x=1}^{N}[I(x,y)-I'(x,y)]^2$$

```
def get_mse(original_img_arr, decoded_img_arr):
# computing the performance metric Mean Squared Error (MSE)
    mse = np.sum((original_img_arr.astype("float") - decoded_img_arr.astype("float")) ** 2)
    mse /= float(original_img_arr.shape[0] * decoded_img_arr.shape[1])
    return mse
```

- **Peak Signal to Noise Ratio (PSNR):** PSNR measures the ratio between maximum pixel intensity and the power of distorting noise affecting the quality of its representation. The noise is the error produced by compression. Higher the PSNR value, the better the quality of the compressed image.

$$PSNR=\frac{10\log_{10}(peakval)^2}{MSE}$$
$$where\ peakval=maximum\ intensity\ value\ of\ the\ image$$
$$and\ MSE=Mean\ squared\ error$$

```
def get_psnr(original, compressed):
#  computing the performance metric Peak Signal to Noise Ratio (PSNR)
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal ==> PSNR has no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * log10(max_pixel / sqrt(mse))
    return psnr
```

- **Structural Similarity Index Measure (SSIM):** In this method, the structural similarity index calculation is based on three aspects namely: image luminance, contrast, and structure. The index measures the similarity between two images x and y,

which can be calculated using the given formula:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_x\sigma_y + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$
$$where\ \mu_x, \mu_y\ are\ the\ local\ means;$$
$$\sigma_x, \sigma_y\ are\ the\ standard\ deviations;$$
$$C_1, C_2\ are\ the\ constants\ used\ to\ ensure\ stability\ when\ denominator\ is\ close\ to\ 0$$

```
def get_ssim(img1, img2):
#  computing the performance metric Structural Similarity Index Measure (SSIM)
    if not img1.shape == img2.shape:
        raise ValueError('Input images must have the same dimensions.')
    if img1.ndim == 2:
        return ssim(img1, img2)
    elif img1.ndim == 3:
        if img1.shape[2] == 3:
            ssims = []
            for i in range(3):
                ssims.append(ssim(img1, img2))
            return np.array(ssims).mean()
        elif img1.shape[2] == 1:
            return ssim(np.squeeze(img1), np.squeeze(img2))
    else:
        raise ValueError('Wrong input image dimensions.')
```

---

3. Experiments

3.1 Setup and logistics:

Libraries and Tools:

We will implement our project in **Python**, using the following libraries:

- **Matplotlib:** Matplotlib is a multi-platform, graphical plotting library for Python used to create interactive visualizations.
- **Numpy:** Numpy is a Python library used for working with arrays. It provides support for large, multi-dimensional arrays and matrices as well as computations including statistical functions, linear algebra, Fourier transform, and other mathematical functions to utilize on these arrays.
- **OpenCV:** OpenCV is an open-source computer vision library mainly used for image processing and machine learning applications. It provides functions to read, save and manipulate image properties among several others.
- **SciPy**: SciPy is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

```
import os
import cv2
import warnings
import numpy as np
import pandas as pd
import scipy.fftpack
import ipywidgets as widgets
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
from matplotlib import rc
from numpy.linalg import svd
from scipy import signal,misc
from math import pi, cos, sqrt, log10
from cv2 import imread, imwrite, imshow
from numpy import pi,sin,r_,zeros, array, clip, trunc
from skimage.metrics import structural_similarity as ssim
from ipywidgets import interact,interactive,interact_manual

warnings.filterwarnings('ignore')
%matplotlib inline
```

Project Logistics:

- **GitHub:** Open source platform used for version control and collaboration.
- **Notion:** To create the project proposal document.
- **Google Colab:** Jupyter notebook environment for collaborating on notebooks.

3.2 Dataset

The Images 4k dataset - performed the compression experiments on 100 jpg images taken from Images 4k

The images stored in the repository folder 4kImages can be loaded into jupyter notebook using the following code snippet.

```
def load_images_from_folder(folder,n):
    data_images = {}
    count = 0
    for filename in os.listdir(folder):
        if count==n:
            return data_images
        img = cv2.imread(os.path.join(folder,filename),cv2.IMREAD_GRAYSCALE)
        if img is not None:
            count += 1
            data_images["Image"+str(count)] = img
    return data_images

data_images = load_images_from_folder("4kImages",100)
```

3.3 Implementation:

First, install and import all the relevant libraries needed.

Next we define functions that will perform the different functionalities such as compressing an image using SVD. from numpy.linalg. The below function performs SVD decomposition on a given image to get the orthogonal matrix U, V, and an array of singular values s. It then creates a truncated reconstruction (using k singular values/vectors) which returns a reconstructed matrix reconst_matrix and an array of singular values s.

```
def compress_svd(image,k):
    U,s,V = svd(image,full_matrices=False)
    reconst_matrix = np.dot(U[:,:k],np.dot(np.diag(s[:k]),V[:k,:]))
    return reconst_matrix
```

3.4 Results:

Singular Values :

- A singular value and its singular vectors give the direction of maximum action among all directions orthogonal to the singular vectors of any larger singular value.
- The hyperparameter involved during performing SVD is k which represents a number of singular values.
- The lower the value of k, the lower the quality of the compressed image will be and the smaller the size of the image will be. Conversely, the higher the value of k as it approaches the rank of the image matrix, the higher the quality of the image will be, and the larger the size will be

Hyperparameter: K = rank of Σ matrix with singular values as diagonal entries

We define and call the below function to get the original and compressed image for a given image, along with the performance metrics.

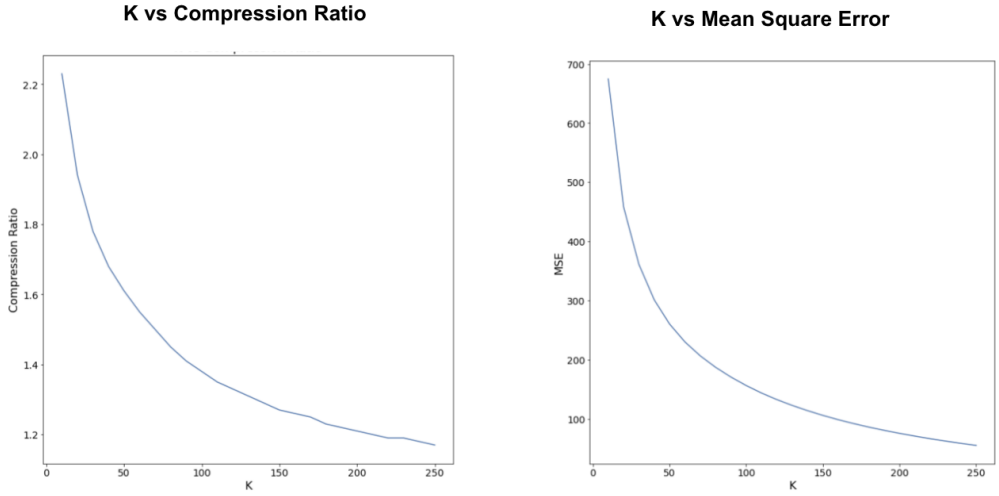
```
pylab.rcParams['figure.figsize'] = (20.0, 7.0)

def compress_svd_images(Image,K):
    image=data_images[Image]
    svd_img = compress_svd(image,K)
```

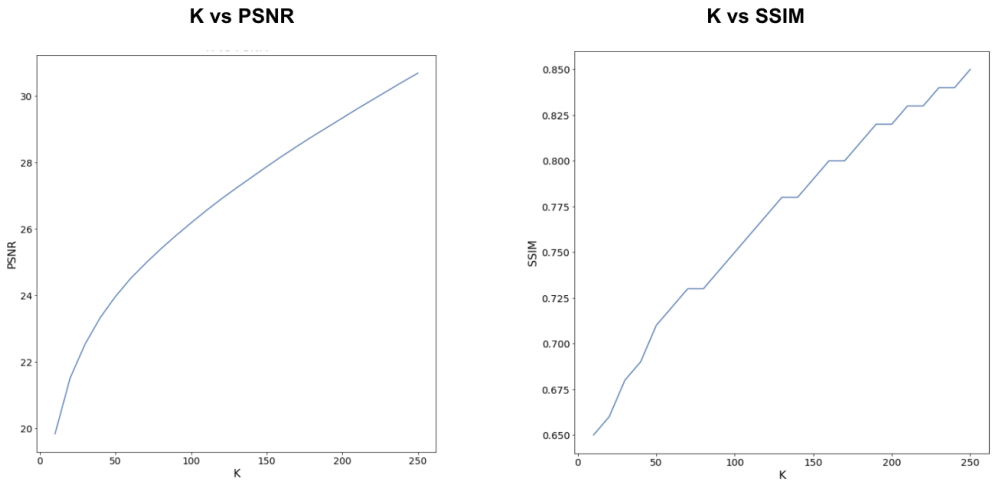
```
    cr_svd,original_size,compressed_size = compressionratio(image,svd_img)
    mse_svd = round(get_mse(image,svd_img),2)
    psnr_svd = round(get_psnr(image,svd_img),2)
    ssim_svd = round(get_ssim(image,svd_img),2)

    f, axarr = plt.subplots(1,2)
    axarr[0].imshow(image,cmap='gray')
    axarr[1].imshow(svd_img,cmap='gray')
    axarr[0].axis('off')
    axarr[1].axis('off')
    axarr[0].title.set_text(r"$\bf{Original\:Image}$"+"\nSize = {} KB\n\n\n\n".format(round(original_size/1024)))
    axarr[1].title.set_text(r"$\bf{SVD\:Image}$"+"\nSize = {} KB\nCompression Ratio = {}\nMean Squared Error = {}\nPeak Signal to Noise Rat
```

K value chosen here to perform SVD = 30

```
compress_svd_images("Image3",30)
```

We also visualized the performance of SVD for various values of K for the above image





3.5 SOTA Implementation:

Similar to SVD we further define functions dct2 and idct2 to compute the DCT and Inverse DCT of an arbitrary type sequence.

```
def dct2(a):
    return scipy.fftpack.dct( scipy.fftpack.dct( a, axis=0, norm='ortho' ), axis=1, norm='ortho' )

def idct2(a):
    return scipy.fftpack.idct( scipy.fftpack.idct( a, axis=0 , norm='ortho'), axis=1 , norm='ortho')
```

We define another function to compress the given image using DCT for a certain threshold value.

```
def compress_dct(im,thresh):
    imsize = im.shape
    dct = np.zeros(imsize)

    # Do 8x8 DCT on image (in-place)
    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            dct[i:(i+8),j:(j+8)] = dct2( im[i:(i+8),j:(j+8)] )

    # Threshold
    dct_thresh = dct * (abs(dct) > (thresh*np.max(dct)))

    percent_nonzeros = np.sum( dct_thresh != 0.0 ) / (imsize[0]*imsize[1]*1.0)

    im_dct = np.zeros(imsize)
    # Do 8x8 Inverse DCT
    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            im_dct[i:(i+8),j:(j+8)] = idct2( dct_thresh[i:(i+8),j:(j+8)] )

    return im_dct
```

3.6 SOTA Results:

Thresholding in a sparse representation reduces the variance of the estimate without increasing the bias, therefore resulting in a small MSE and better denoising estimate.

Hyperparameter: Threshold = discarding those values of the DCT matrix that fall below this threshold value

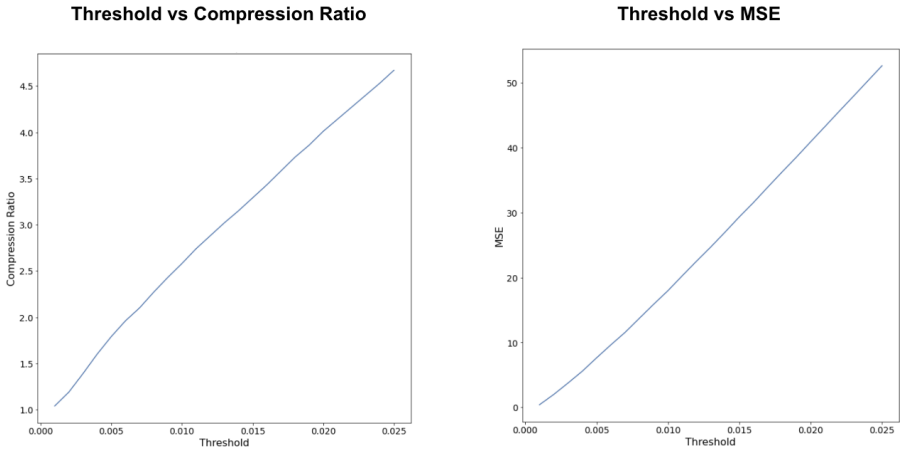
We define and call the below function to get the original and compressed image for a given image, along with the performance metrics.

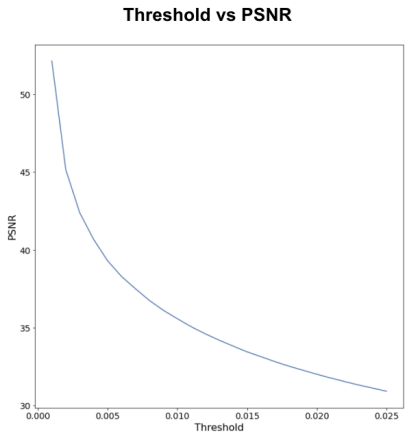
```
pylab.rcParams['figure.figsize'] = (20.0, 7.0)
def compress_dct_images(Image,Threshold):
    image = data_images[Image]
    dct_img = compress_dct(image,Threshold/10000)

    cr_dct,original_size,compressed_size = compressionratio(image,dct_img)
    mse_dct = round(get_mse(image,dct_img),2)
    psnr_dct = round(get_psnr(image,dct_img),2)
    ssim_dct = round(get_ssim(image,dct_img),2)

    f, axarr = plt.subplots(1,2)
    axarr[0].imshow(image,cmap='gray')
    axarr[1].imshow(dct_img,cmap='gray')
    axarr[0].axis('off')
    axarr[1].axis('off')
    axarr[0].title.set_text(r"$\bf{Original\:Image}$"+"\nSize = {} KB\n\n\n\n".format(round(original_size/1024)))
    axarr[1].title.set_text(r"$\bf{DCT\:Image}$"+"\nSize = {} KB\nCompression Ratio = {}\nMean Squared Error = {}\nPeak Signal to Noise R
ation = {}\nStructural Similarity Index Measure = {}".format(round(compressed_size/1024),cr_dct,mse_dct,psnr_dct,ssim_dct))
```

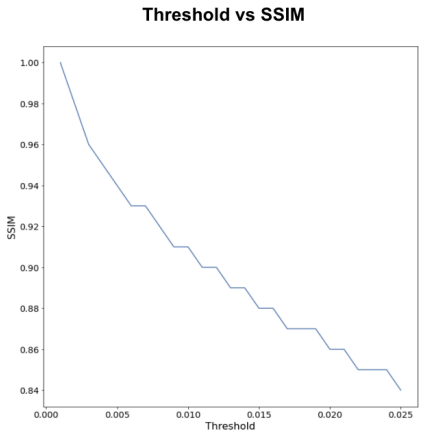
The threshold value is chosen here to perform DCT = 0.0050

```
compress_dct_images("Image3",50)
```

We also visualized the performance of DCT for various values of Threshold for above image







3.7 Compare and contrast:

Singular Value Decomposition

Advantages:

1. Very efficient.
2. The basis is hierarchical, ordered by relevance.
3. It tends to perform quite well for most data sets

Disadvantages:

1. If the data is strongly non-linear it may not work so well. Results are not always the best for visualization
2. It is difficult to interpret
3. Strongly focused on variance, sometimes there's not a direct relationship between variance and predictive power so it can discard useful information

Discrete Cosine Transform

Advantages:

1. The transformation is an orthogonal fast algorithm that can be used for computation, and the output for (near) constant matrices generally consists of a large number of (near) zero values.
2. The DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain

Disadvantages:

1. It does not localize the frequency components in space.
2. While the input from preprocessed 8 x 8 blocks is integer-valued, the output values are typically real-valued. Thus we need a quantization step to make some decisions about the values in each DCT block and produce output that is integer-valued.

We define and call the below function to compare the SVD and DCT compressed images with respect to the original image.

```
pylab.rcParams['figure.figsize'] = (20.0, 10.0)
def compare_dct_svd_images(Image):
    '''
    function to compare the original image, svd compressed image, dct compressed image, as well as the performance metrics
    for svd and dct compression for the given image.
    '''
    image = data_images[Image]
    svd_img = compress_svd(image,30)
    dct_img = compress_dct(image,32/10000)

    cr_svd,original_size,size_svd = compressionratio(image,svd_img)
    mse_svd = round(get_mse(image,svd_img),2)
    psnr_svd = round(get_psnr(image,svd_img),2)
    ssim_svd = round(get_ssim(image,svd_img),2)

    cr_dct,original_size,size_dct = compressionratio(image,dct_img)
    mse_dct = round(get_mse(image,dct_img),2)
    psnr_dct = round(get_psnr(image,dct_img),2)
    ssim_dct = round(get_ssim(image,dct_img),2)

    f, axarr = plt.subplots(1,3)
    axarr[0].imshow(image,cmap='gray')
```

```
    axarr[1].imshow(svd_img,cmap='gray')
    axarr[2].imshow(dct_img,cmap='gray')
    axarr[0].axis('off')
    axarr[1].axis('off')
    axarr[2].axis('off')
    axarr[0].title.set_text(r"$\bf{Original\:Image}$"+"\nSize = {} KB\n\n\n\n".format(round(original_size/1024)))
    axarr[1].title.set_text(r"$\bf{SVD\:Image}$"+"\nSize = {} KB\nCompression Ratio = {}\nMean Squared Error = {}\nPeak Signal to Noise Rat
    axarr[2].title.set_text(r"$\bf{DCT\:Image}$"+"\nSize = {} KB\nCompression Ratio = {}\nMean Squared Error = {}\nPeak Signal to Noise Rat
    plt.axis('off')
```

```
compare_dct_svd_images("Image3")
```

Overall Performance of SVD and DCT

The SVD and DCT algorithms have been implemented on all **100 images** in our data set and performance metrics were calculated for every image. We have taken the average value of each metric and summarized the results as shown below:

Average image size: 2238KB Average compressed image size: 1224KB	**SVD K = 30**	**DCT Threshold = 0.0050**
CR	1.86	1.83
MSE	359.79	4.01
PSNR	24.75	42.65
SSIM	0.65	0.96

Latest Advances

The Image Compression technique being widely used these days is Huffman Coding. Huffman coding is a lossless data compression technique. Huffman coding is based on the frequency of occurrence of a data item i.e. pixel in images. The technique is to use a lower number of bits to encode the data into binary codes that occur more frequently. It is used in JPEG files.

---

4. Conclusion

- In this project, we implemented image compression using SVD and DCT techniques.
- We observed reduced image sizes post-compression for both of these methods.
- For SVD, large values of the hyperparameter give good results in terms of performance metrics and for DCT, low values of the hyperparameter give good results.
- For a given compression ratio, DCT resulted in a less distorted image as compared to SVD.
- Finally, we saw the challenges/limitations of the implementation and how to overcome them.

---

5. Acknowledgements

We are really grateful to Prof. Tsui-Wei Weng, Halıcıoğlu Data Science Institute, San Diego for her continuous support, encouragement, and willingness to help us throughout this project.

6. References

- Sara, U. et al. (2019) *Journal of Computer and Communications*, **7**, 8-18 Image Quality Assessment through FSIM, SSIM, MSE and PSNR—A Comparative Study.
- Asnaoui, K. (2020)*Journal of Intelligent Systems*, 29(1), 1345-1359 Image Compression Based on Block SVD Power Method.
- H R Swathi *et al* 2017 *IOP Conf. Ser.: Mater. Sci. Eng* **263** 042082 Image compression using singular value decomposition
- Medium. 2020. *Compressing Images Using Linear Algebra.* [online] Available at: https://medium.com/analytics-vidhya/compressing-images-using-linear-algebra-bdac64c5e7ef
- Rebaza J (2012). *A First Course in Applied Mathematics.* Image Compression. Wiley & Sons Ltd. http://people.missouristate.edu/jrebaza/assets/10compression.pdf
- Singular Value Decomposition - Applications in Image Processing. [online] Available at: https://www2.karlin.mff.cuni.cz/~tuma/Aplikace15/prezentace_Hnetynkova.pdf
- Singular Value Decomposition Tutorial. [online] Available at: https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm
- http://www.math.utah.edu/~gustafso/s2019/2270/projects-2016/kuhnConnor/kuhnConnor-svd-vs-dct-image-compression.pdf
- https://www.ipol.im/pub/art/2011/ys-dct/article.pdf
- https://www.dam.brown.edu/drp/proposals/SamChowning.pdf
- Discrete Cosine Transform