Linear Algebra Project: Image Compression

International Institute of Information Technology, Hyderabad

Vinit Mehta, Swarang Joshi, Nijesh Raghava

1 Introduction

Image compression is a fundamental technique in the field of computer vision and image processing. It plays a crucial role in reducing the storage requirements and transmission bandwidth for images without significant loss in visual quality. Over the years, numerous image compression techniques have been developed, aiming to achieve higher compression ratios while maintaining good perceptual quality. In recent years advanced algorithms and approaches to push the boundaries of image compression performance have emerged. This thesis focuses on exploring the use of Singular Value Decomposition(SVD), Vector Quantization, Discrete Cosine Transform(DCT) and Wavelet Transform.

1.1 SVD

Image compression using Singular Value Decomposition (SVD) has been an active area of research, and several recent updates have focused on enhancing compression efficiency and image quality. Here are some notable advancements in the field of image compression using SVD:

- **Adaptive SVD-based Compression:** Traditional SVD-based compression methods use a fixed number of singular values to compress the image. Recent approaches have introduced adaptive techniques that dynamically select the number of singular values based on image characteristics. This adaptive approach allows for better image quality by allocating more singular values to preserve important image features.
- **Hybrid Compression Schemes:** Researchers have explored combining SVD with other compression methods, such as discrete cosine transform (DCT), wavelet transform, or predictive coding, to achieve even higher compression ratios with improved image quality. These hybrid approaches leverage the strengths of multiple compression techniques to achieve superior results.
- **Deep Learning-based Approaches:** Deep learning approaches, particularly using convolutional neural networks (CNNs), have shown promising results in image compression. These methods learn an optimized representation of the image and exploit redundancy for efficient compression. By training CNN-based models on large image datasets, researchers have achieved competitive compression ratios with improved image quality.
- **Progressive Compression:** Progressive compression techniques allow for gradual refinement of the image quality during decompression. This means that a low-quality version of the image can be quickly reconstructed from a subset of compressed data, and subsequent iterations gradually improve the image quality. Recent developments in progressive compression using SVD have focused on optimizing the bit allocation and refinement process to achieve better compression efficiency and visual quality.

1.2 DCT/Wavelet Transform

Haar wavelet transformation has been widely used in image and video compression techniques, such as JPEG2000 and MPEG-4. By exploiting the sparse representation of images in the wavelet domain, it enables efficient storage and transmission of visual data while maintaining good quality. Signal and data analysis: Haar wavelets are used in signal processing and data analysis to extract meaningful information from noisy or complex signals. They can detect and localize abrupt changes or edges in signals, making them useful in applications such as image denoising, speech recognition, and pattern recognition. Biomedical signal processing: Haar wavelets are employed in various biomedical applications, including electrocardiography (ECG) and electroencephalography (EEG) analysis. They can help detect and classify different patterns and abnormalities in these signals, aiding in diagnosing heart conditions, seizures, and other neurological disorders.Deep Learning and Haar Wavelet: Recent research has explored the integration of deep learning techniques with Haar wavelet transformation. This hybrid approach combines the strengths of deep learning in feature extraction and classification with the multiresolution analysis capabilities of the Haar wavelet, leading to improved performance in various tasks, such as image recognition and anomaly detection.

1.3 Vector Quantization

Vector quantization (VQ) is a powerful technique widely used in image and video compression, pattern recognition, and data analysis. It offers an efficient means of representing and compressing data by exploiting the inherent similarities and redundancies within the dataset. By grouping similar data vectors together and representing them with a smaller set of representative vectors, vector quantization achieves high compression ratios while minimizing the loss of important information.

In vector quantization, a dataset is divided into smaller subgroups or clusters based on the similarity of the data vectors. Each cluster is then represented by a single vector known as a codeword or codebook entry. The process of creating these representative vectors is known as the training or codebook generation phase. Once the codebook is established, the original dataset is encoded by replacing each data vector with the index of its closest codeword in the codebook.

Vector quantization offers several advantages over other compression techniques. First, it provides a flexible and adaptable approach, capable of handling a wide range of data types and structures. It is particularly effective in scenarios where the data exhibits spatial or temporal correlations, such as in images, videos, or speech signals. Second, vector quantization can achieve high compression ratios by effectively exploiting the redundancies and similarities within the dataset. By representing clusters of similar vectors with a single codeword, it reduces the storage and transmission requirements. Third, vector quantization can be tailored to specific application requirements by adjusting the size and structure of the codebook. This allows for a trade-off between compression ratio and the quality of the reconstructed data.

The process of vector quantization involves two main steps: encoding and decoding. During the encoding phase, each data vector is compared with the codewords in the codebook to find the closest match. The index of the closest codeword is then assigned to represent the original data vector. In the decoding phase, the encoded indices are used to reconstruct the data by replacing them with the corresponding codewords. Although there is some loss of information due to the quantization process, vector quantization techniques strive to minimize this loss and preserve the essential characteristics of the original data.

In this project, we will explore the principles and applications of vector quantization in image compression. We will investigate various aspects, such as codebook generation methods, quantization strategies, and evaluation metrics for assessing the performance of vector quantization algorithms. By understanding and implementing vector quantization, we aim to achieve efficient compression while maintaining satisfactory image quality. The insights gained from this project will provide a foundation for utilizing vector quantization in real-world scenarios and contribute to the broader field of data compression and analysis.

2 Key Objectives

2.1 SVD

1. Understand the concept of SVD: The primary objective is to grasp the fundamental concept of SVD, which involves decomposing a matrix into three separate matrices: U, Σ, and $V^\top$.
2. earn the mathematical formulation: Gain a comprehensive understanding of the mathematical formulation of SVD, including the relationship between the original matrix, U, Σ, and $V^\top$.
3. Comprehend the geometric interpretation: Explore the geometric interpretation of SVD, which involves interpreting the effect of each matrix (U, Σ, and $V^\top$) on the shape and orientation of the original matrix.
4. Apply SVD for dimensionality reduction: Utilize SVD as a tool for dimensionality reduction in data analysis and machine learning tasks. Understand how to select and retain the most important singular values to preserve relevant information.
5. Utilize SVD in data compression: Explore how SVD can be employed in data compression techniques, such as image compression, by leveraging the property of SVD to capture the most significant information using a smaller number of singular values.
6. Understand the relationship with eigenvalues and eigenvectors: Establish a clear understanding of the relationship between SVD and eigenvalue decomposition. Recognize that the singular values of a matrix are the square roots of the eigenvalues of the corresponding symmetric positive semidefinite matrix.
7. Apply SVD in matrix approximation and reconstruction: Learn how to use SVD to approximate a matrix by retaining only a subset of singular values, allowing for efficient storage and reconstruction of the original matrix.

2.2 DCT/Wavelet Transform

The key objectives in wavelet transform include:

Compression and Denoising: Wavelet transform is widely used for data compression and denoising applications. By discarding or quantizing wavelet coefficients with low magnitudes, redundant or noise components can be removed, leading to efficient compression or denoising of signals or images while preserving important features.
Signal Processing and Filtering: Wavelet transform provides a flexible tool for signal processing and filtering. It allows for the analysis and manipulation of signals in both the time and frequency domains simultaneously. Wavelet-based filters can be designed to enhance or suppress certain frequency components or time-localized features in a signal.
Image and Video Processing: Wavelet transform is widely used in image and video processing tasks such as image compression, denoising, enhancement, and object detection. Its ability to capture localized features and its multiresolution analysis properties make it suitable for analyzing and processing images and videos efficiently.

2.3 Vector Quantization

3 Problem Statement/ Formulation

The problem statement of our project is to investigate and compare different image compression techniques that utilize linear algebra, specifically focusing on vector quantization, singular value decomposition, discrete cosine transform, and wavelet transform. We aim to understand the underlying mathematical principles, advantages, and limitations of each technique. Furthermore, we plan to implement these techniques and evaluate their performance on a common example.

4 Use Case of Linear Algebra in the Project

Linear algebra plays a crucial role in image compression techniques. It provides the mathematical foundation for representing and manipulating images as matrices and vectors. Specifically, vector quantization relies on linear algebra to quantize image vectors into codebooks, while principal component analysis utilizes linear algebra to find orthogonal basis vectors that capture the most significant variations in the image data. Similarly, discrete cosine transform and wavelet transform employ linear algebra to transform the image into a compressed representation by exploiting certain properties of the image signals.

5 Tentative Timeline Plan

Our tentative timeline plan for the remainder of the project is as follows:

- **Day 1 and 2:** Conduct literature review on image compression techniques, focusing on vector quantization, singular value decomposition, discrete cosine transform, and wavelet transform. Study and analyze the selected research papers, understanding the mathematical principles and algorithms behind each technique.
- **Day 3:** Implement vector quantization, principal component analysis, discrete cosine transform, and wavelet transform algorithms.
- **Day 4:** Prepare a common example image and compress it using all three techniques.
- **Day 5 and 6:** Analyze the results and draw conclusions on the effectiveness and suitability of each technique. Begin finalizing the project report and presentation.

6 Individual Work Contribution

Each team member have done the following works:

- Vinit Mehta: Prepared Interim Report in Latex. Created GitHub repository for collaborating. Conduct the literature review, analysed research papers, and contributed to the implementation of the image compression techniques. Explored the method of Singular Value Decomposition and it's application of Imgae compression using Singular Value Decomposition algorithm. Prepared the ppt demostrating the key highlights of the pdf document. Merged all the latex files and written all common headings.
- Swarang Joshi: Will study and analyse the mathematical principles and algorithms of the selected techniques, implement the vector quantization algorithm, and assist in the evaluation and analysis of the results.
- Nijesh Raghava: Will explore the application of discrete cosine transform, and wavelet transform, and assist in the evaluation and analysis of the results. Implement the wavelet transform algorithm and assist in the evaluation and analysis of the results.

7 Methodology

Note we will be using various python libraries like Pandas, Numpy, MatPlotLib etc. for manipulating data and plotting graphs and images. All the code related to this discussion here can be found in this Git Repository.

7.1 SVD

There are various steps involved in performing SVD, and they are as follows.

Prepare the data by ensuring it is in a suitable format for SVD. If necessary, perform any required data cleaning, normalization, or scaling. There are two methods for image compression we are talking about here, first one uses the decomposition of the image data matrix (with some rank > 1) into the sum of various rank

1 matrices (this can be achieved using Singular Value Decomposition(SVD)).

Singular Value Decomposition

The images need to be converted into a data matrix A where each entry A_{ij} represents the pixel color value at that point (for simplicity we will take an example of grey scale image that consists of only two colors black(represented by 1) and white(represented by 0)). For example consider the following image of a heart and it's corresponding data matrix

0	1	1	0	1	1	0
1	1	1	1	1	1	1
1	1	1	1	1	1	1
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	1	0	0	0

Fig: Example image

```python
# The data matrix can be represented as an numpy 2d-array like this
A = numpy.array([[0,1,1,0,1,1,0],
                 [1,1,1,1,1,1,1],
                 [1,1,1,1,1,1,1],
                 [0,1,1,1,1,1,0],
                 [0,0,1,1,1,0,0],
                 [0,0,0,1,0,0,0]])
```

This data matrix can be decomposed into a composition of 3 matrices by using the method of Singular Value Decomposition(SVD)

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times m}V^{\top}_{n\times n}$$

Here U and V are orthogonal matrices and Σ is a diagonal matrix whose diagonal entries represent the relative amount of data each rank 1 matrix that is generated by outer product of corresponding columns of U and rows of $V^{\top}$ contains.

```python
# We can use the .svd() method from the linear algebra class of Numpy Library
U, S, V = numpy.linalg.svd(A)
```

The above function generates the 3 matrices as follows:

$$U = \begin{bmatrix} -0.36 & 0 & -0.73 & -0.05 & 0.56 & 0.13 \\ -0.54 & 0.35 & 0.27 & -0.08 & -0.16 & 0.69 \\ -0.54 & 0.35 & 0.27 & -0.08 & 0.16 & -0.69 \\ -0.45 & -0.35 & -0.27 & 0.52 & -0.56 & -0.13 \\ -0.28 & -0.71 & 0.18 & -0.62 & 0 & 0 \\ -0.08 & -0.35 & 0.46 & 0.57 & 0.56 & 0.13 \end{bmatrix}_{6\times 6}$$

$$S = \begin{bmatrix} 4.74 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.41 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.41 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.73 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{6\times 6}$$

$$V = \begin{bmatrix} -0.23 & -0.4 & -0.46 & -0.4 & -0.46 & -0.4 & -0.23 \\ 0.5 & 0.25 & -0.25 & -0.5 & -0.25 & 0.25 & 0.5 \\ 0.39 & -0.32 & -0.19 & 0.65 & -0.19 & -0.32 & 0.39 \\ -0.22 & 0.42 & -0.44 & 0.42 & -0.44 & 0.42 & -0.22 \\ 0.56 & -0.43 & 0.03 & 0 & -0.03 & 0.43 & -0.56 \\ -0.42 & -0.55 & -0.16 & 0 & 0.16 & 0.55 & 0.42 \\ -0.12 & -0.11 & 0.69 & 0 & -0.69 & 0.11 & 0.12 \end{bmatrix}_{7\times 7}$$

Now before moving forward let us analyse what the SVD equation represents and how it expresses the original matrix as a sum of rank 1 matrices.

Rank of a Matrix:
Rank of a matrix is defined as the number of linearly independent column/row vectors (whichever is less in number) or in other words it is the number of zero rows in the row-reduced echelon form of the matrix.
For our purpose we would just need to consider the first definition of the matrix and we won't dwell into the latter definition
Consider the matrix given below

$$X = \begin{bmatrix} 6 & 3 & 9 & 12 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 6 & 8 \\ 8 & 4 & 12 & 16 \end{bmatrix}_{4\times 4}$$

Do you notice some pattern?
On first sight it seems that since the matrix has 4 rows and 4 columns than it's rank should be 4, but it turns out that it is not always the case. Here we observe an interesting pattern in the rows (as well as columns) of the matrix. Notice that

$$R_1 = 3 \times R_2 \tag{1}$$
$$R_3 = 2 \times R_2 \tag{2}$$
$$R_4 = 4 \times R_2 \tag{3}$$

So after this analysis we observe that actually the rank of the given matrix is just 1 and not 4.

Vector Outer Product:
You might be familiar with the vector inner product which is just the dot product of two vectors, the result of dot/inner product of two vectors is a single scalar quantity which is the summation of products of respective elements of the vectors (note that both the vectors must have same number of elements in them for inner product to exist)

$$\vec{x} \cdot \vec{y} = \textstyle\sum_{i=1}^{n} x_i y_i, \ \ where \ ||\vec{x}|| = ||\vec{y}|| = n$$

On the other hand outer product of two vectors produces an $m \times n$ matrix where $||\vec{x}|| = m$ and $||\vec{y}|| = n$. The outer product of $\vec{x}$ and $\vec{y}$ is the same as the product $\vec{x}\vec{y}^{\top}$ where $\vec{x}$ and $\vec{y}$ are both column vectors.

$$\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^{\top} = \mathbf{A} = \begin{bmatrix} x_1y_1 & x_1y_2 & \dots & x_1y_n \\ x_2y_1 & x_2y_2 & \dots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \dots & x_my_n \end{bmatrix}_{m\times n}$$

Let us take the previous example and try to visualise what is actually happening.
The matrix A can be written as the outer product of two vectors

$$\mathbf{a} = \begin{bmatrix} 3 \\ 1 \\ 2 \\ 4 \end{bmatrix}_{4\times 1} \quad \textit{and } \mathbf{b}^\top = \begin{bmatrix} 2 & 1 & 3 & 4 \end{bmatrix}_{1\times 4}$$

So basically outer product is nothing but the scaling of vector $b^\top$ by different factors given by $\left[\vec{a}\right]_i$ to get the row_i of matrix A. So if we repeat the process for all the rows we get out matrix A.

$$\mathbf{a} \otimes \mathbf{b} = \mathbf{a}\mathbf{b}^\top = \mathbf{A} = \begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 & a_1b_4 \\ a_2b_1 & a_2b_2 & a_2b_3 & a_2b_4 \\ a_3b_1 & a_3b_2 & a_3b_3 & a_3b_4 \\ a_4b_1 & a_4b_2 & a_4b_3 & a_4b_4 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 9 & 12 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 6 & 8 \\ 8 & 4 & 12 & 16 \end{bmatrix}$$

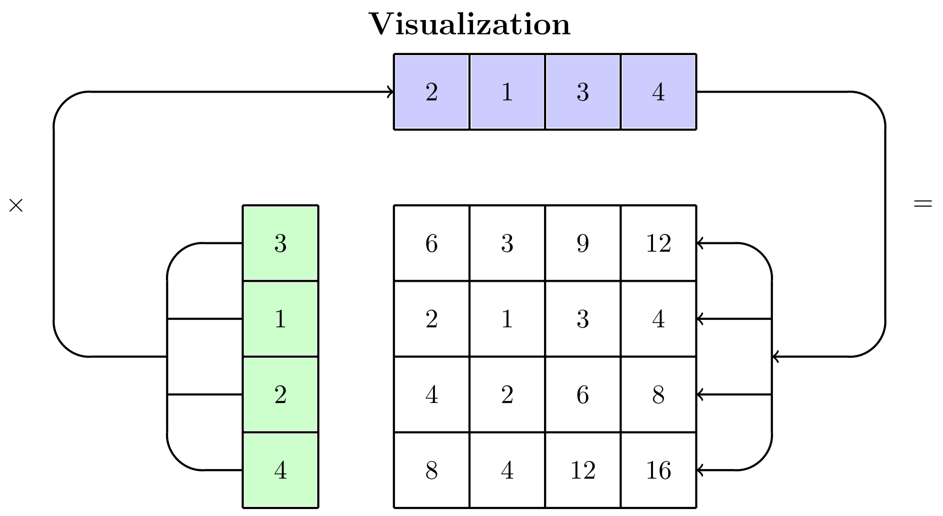


Fig: How outer product of two vectors is calculated

Having the knowledge of Rank of a matrix and outer product of two vectors we can go ahead in our journey. So how does the SVD decomposition represents the original matrix as the sum of rank 1 matrices? We can answer this question now with our knowledge of rank of matrices and outer product of vectors. Note that S (Σ) is a diagonal matrix so when U is multiplied with S it is just as scaling each corresponding column vector of U with the corresponding diagonal entry, that is, if

$$U_{m\times m} = \begin{bmatrix} \vec{u}_1 & \vec{u}_2 & \dots & \vec{u}_m \end{bmatrix}$$

and

$$\Sigma_{m\times m} = diag(\sigma_1 \quad \sigma_2 \quad \dots \quad \sigma_m)$$

than

$$U\Sigma = \begin{bmatrix} \vec{u}_1\sigma_1 & \vec{u}_2\sigma_2 & \dots & \vec{u}_m\sigma_m \end{bmatrix} = \begin{bmatrix} \sigma_1\vec{u}_1 & \sigma_2\vec{u}_2 & \dots & \sigma_m\vec{u}_m \end{bmatrix}$$

Now right multiplying both sides by $V^\top$ we get

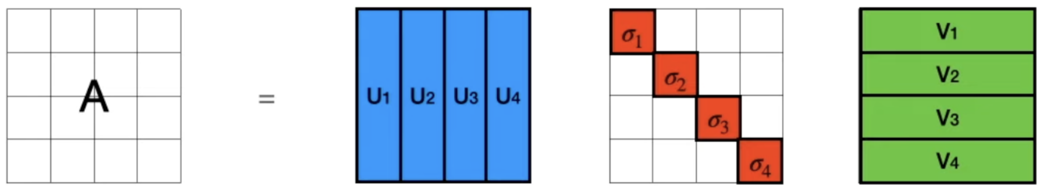
$$U\Sigma V^\top = \begin{bmatrix}\sigma_1\vec{u}_1 & \sigma_2\vec{u}_2 & \dots & \sigma_m\vec{u}_m\end{bmatrix}\begin{bmatrix}\vec{v}_1^\top \\ \vec{v}_2^\top \\ \vdots \\ \vec{v}_n^\top\end{bmatrix}$$

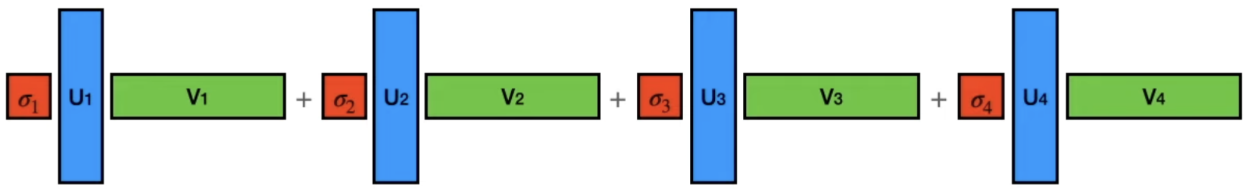
$$\therefore [U\Sigma V^\top]_i = \sigma_i(\vec{u}_i \otimes \vec{v}_i^\top)$$

Note that here $\vec{v}_i$ are row vectors and $\vec{u}_i$ are column vectors so here we will be doing vector outer product (also you can notice that the dimensions of the two matrices are not multiplication compatible. We can calculae the outer product using the following python code:

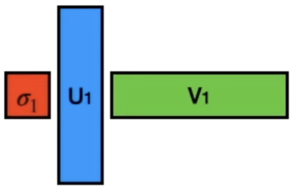
```python
# code for calculating the outer product of two vectors in python
# All the layers in python can be generated by this code (taking outer product)
# array to store all the layers
array_of_layers = []
for layer in range(len(U[0])): # number of layers = number of columns of U
    img_layer = [] # each layer would be a 2D matrix
    colU = []
    rowV = V[layer]
    for row in range(len(U)):
        colU.append(U[row][layer])
    sigma = S[layer]
    for row in range(len(colU)):
        r = []
        for i in range(len(rowV)):
            r.append(colU[row] * rowV[i] * sigma)
        img_layer.append(r)
    array_of_layers.append(img_layer)
```

Visualization





Here in the figure each of the summation term is a rank 1 matrix as all the rows of a matrix in particular term is the linear combination of the vectors $\vec{v}_i$ according to $\vec{u}_i$s. So now we have decomposed our original matrix as a sum of rank 1 matrices. The amount of relative information contained in each rank 1 matrix about the original matrix is given by the associated σ_i, so now we want to approximate our original matrix in terms of the sum of rank 1 matrices we can drop those matrices in this expansion whose σ coefficient is very small. The first term in the matrix example that we are working with will be



$$= (4.74) \begin{bmatrix} -0.36 \\ -0.54 \\ -0.54 \\ -0.45 \\ -0.28 \\ -0.08 \end{bmatrix} \begin{bmatrix} -0.23 & 0.5 & 0.39 & -0.22 & 0.56 & -0.42 & -0.12 \end{bmatrix}$$

$$= (4.74) \begin{bmatrix} 0.08 & -0.18 & -0.14 & 0.07 & -0.20 & 0.15 & 0.04 \\ 0.12 & -0.27 & -0.21 & 0.11 & -0.30 & 0.23 & 0.06 \\ 0.12 & -0.27 & -0.21 & 0.12 & -0.30 & 0.23 & 0.06 \\ 0.10 & -0.22 & -0.17 & 0.09 & -0.25 & 0.18 & 0.05 \\ 0.06 & -0.14 & -0.10 & 0.06 & -0.16 & 0.12 & 0.03 \\ 0.01 & -0.04 & -0.03 & 0.02 & -0.04 & 0.03 & 0.01 \end{bmatrix}$$

Similarly we calculate the data matrix for each term in the summation. After multiplying the σ into the matrix and plotting the image from the matrix treating each entry as the value of the pixel we get the following. The value written at the top of each image indicates the respective rank 1 matrix that it represents.

```python
# All the layers in python can be generated by this code (taking outer product)
# array to store all the layers
array_of_layers = []
for layer in range(len(U[0])): # number of layers = number of columns of U
    img_layer = [] # each layer would be a 2D matrix
    colU = []
    rowV = V[layer]
    for row in range(len(U)):
        colU.append(U[row][layer])
    sigma = S[layer]
    for row in range(len(colU)):
        r = []
        for i in range(len(rowV)):
            r.append(colU[row] * rowV[i] * sigma)
        img_layer.append(r)
    array_of_layers.append(img_layer)
```

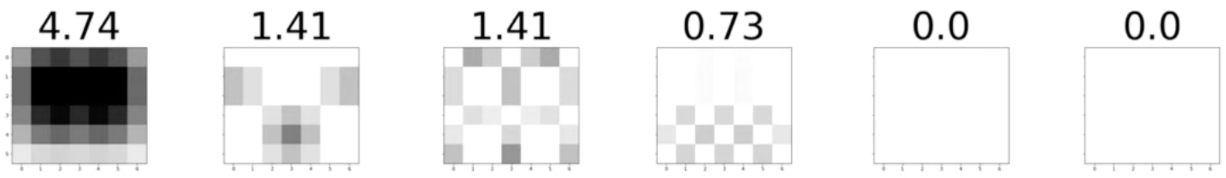


Fig: Layers (Representation of each rank 1 matrix)

The above images can be represented as different layers of the original image so as we go on putting one layer on top of the other our original image starts to form. As we keep adding more and more terms we get colser and closer to our original image. On adding these images on top of one another we get the following:

```python
# The masked images can be generated by the following code
array_of_masked_images = []
for layer in array_of_layers:
    if len(array_of_masked_images) == 0:
        array_of_masked_images.append(layer)
        pass
    else:
        prev_img = array_of_masked_images[-1]
        curr_layer = layer
        # generates a zero array of the dimensions of the image
        new_image = [[0 for _ in range(len(prev_img[0]))] for _ in range(len(prev_img))]
        for i in range(len(curr_layer)):
            for j in range(len(curr_layer[0])):
                new_image[i][j] += prev_img[i][j] + curr_layer[i][j]
        array_of_masked_images.append(new_image)
```

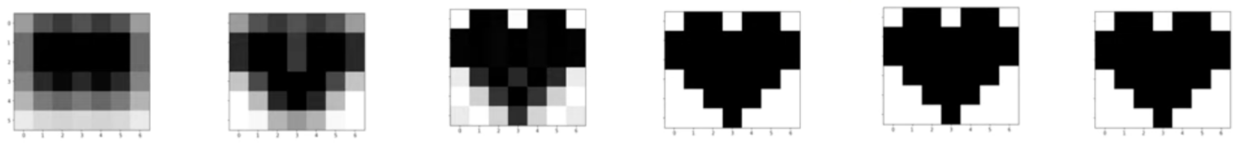


Fig: Overlapping of each layer progressively

This shows that even if we just store the first three layers we get a pretty good approximation of our original image without loosing much information (Note that here the number of layers is quiet low so it does not make sense to store just a few parts as the image is already very low resolution but in case of high resolution images this methods makes a significant difference). If we just store the first three layers than the total number of values we will need to store is the first three columns of U and first three columns of V (or first three rows of $V^\top$, that is $\implies$ $(3 \times 6) + (3 \times 7) = 18 + 21 = 39$ values (Since we can easily generate the image by taking the outer products of the respective vectors and adding each 1 rank matrix)(Also note that we need not explicitely store the σ values as it can be first multiplied into the respective columns of U and then those columns can be directly stored). Which is less than the $7 \times 6 = 42$ values that we would need to store for the original image(Although it is not much in this case but for higher resolution images this is a significant number, for example for a image containing 4000 x 4000 resolution if we even store just the first 100 columns(with retaining more than 95% of quality) of respective U and V matrices we would just need to store 8,00,000 values as compared to 1,60,00,000 values we would otherwise need to store for the original image).

7.2 DCT/Wavelet Transform

Haar wavelet compression is an efficient way to perform both lossless and lossy image compression. It relies on averaging and differentiating values in an image matrix to produce Wavelet Transform.

Averaging and Differentiating

To understand how the technique works, let us start with an example. As the computer system deals the image as a matrix or an array of discrete values known as “pixels” or “picture elements,” these discrete values range from 0 (black) to some positive number values 255 (white). For computing the Haar wavelet of an image, we must convert an image to a discrete matrix as Haar wavelet transform cannot deal with continuous data, and discrete matrix of an image can be achieved using MATLAB (a programming tool). In this paper, we considered only an 8x8 image block for processing but note that the same technique is applied to the whole image to calculate the HWT of the image.

$$P = \begin{bmatrix} 576 & 704 & 1152 & 1280 & 1344 & 1472 & 1536 & 1536 \\ 704 & 640 & 1156 & 1088 & 1344 & 1408 & 1536 & 1600 \\ 768 & 832 & 1216 & 1472 & 1472 & 1536 & 1600 & 1600 \\ 832 & 832 & 960 & 1344 & 1536 & 1536 & 1600 & 1536 \\ 832 & 832 & 960 & 1216 & 1536 & 1600 & 1536 & 1536 \\ 960 & 896 & 896 & 1088 & 1600 & 1600 & 1600 & 1536 \\ 768 & 768 & 832 & 832 & 1280 & 1472 & 1600 & 1600 \\ 448 & 768 & 704 & 640 & 280 & 1408 & 1600 & 1600 \end{bmatrix}_{8\times 8}$$

To convert matrix "P" to wavelet transform, first we will find the average and difference of each row and column of the matrix, as these are required for computing Haar wavelet transform of an image. The following steps were applied to obtain a transformed matrix:

- Deal each vector (row) of the matrix as a string.
- Now, group all of the columns in pairs
- We replace the first 4 columns of row with the average of these pairs and replace the last 4 columns with $\frac{1}{2}$ of the difference of these pairs.
- We repeat the process until desired level of compression or decomposition is reached and is repeated on all the rows of the matrix

Initially the first row of P

$$P_1 = \begin{bmatrix} 576 & 704 & 1152 & 1280 & 1344 & 1472 & 1536 & 1536 \end{bmatrix}$$

Averaging and differentiating the first row of P results in :

$$P_1 = \begin{bmatrix} 640 & 1216 & 1408 & 1536 & -64 & -64 & -64 & 0 \end{bmatrix}$$

Averaging and differentiating the first row of P again results in :

$$P_1 = \begin{bmatrix} 928 & 1472 & -288 & -64 & -64 & -64 & -64 & 0 \end{bmatrix}$$

and averaging and differentiating the first row of P for the third time we get :

$$P_1 = \begin{bmatrix} 1200 & -272 & -288 & -64 & -64 & -64 & -64 & 0 \end{bmatrix}$$

As the averaging and differentiating is a reversible process, we can work back from any row to the previous row in the table and hence to the first row using appropriate addition and subtraction techniques. Here, in the whole transformation process, we lost nothing. After completing the transformation process on rows, we will repeat the same transformation process on the columns to achieve high compression. This can be achieved by applying transpose to the row transformed matrix. This gives another matrix "T" with an 8x8 dimension, called Haar wavelet transforms of the table "P" as shown.

$$T = \begin{bmatrix} 1212 & -306 & -146 & -54 & -24 & -68 & -40 & 4 \\ 30 & 36 & -90 & -2 & 8 & -20 & 8 & -4 \\ -50 & -10 & -20 & -24 & 0 & 72 & -16 & -16 \\ 82 & 38 & -24 & 68 & 48 & -64 & 32 & 8 \\ 8 & 8 & -32 & 16 & -48 & -48 & -16 & 16 \\ 20 & 20 & -56 & -16 & -16 & 32 & -16 & -16 \\ -8 & 8 & -48 & 0 & -16 & -16 & -16 & -16 \\ 44 & 36 & 0 & 8 & 80 & -16 & -16 & 0 \end{bmatrix}$$

The matrix T has an overall average value at the top and 63 detail elements. These detail elements are used to capture the high frequency components or details of the original data.The matrix T is called the wavelet-transformed matrix of the matrix P. A matrix with a high proportion of zeros is called a sparse matrix. Corresponding wavelet-transformed matrices of many images are sparser than the original image matrices and that is why it is easier and efficient to transmit and store a matrix that is more sparse than an ordinary matrix of the same size.

So, if we transmit the wavelet-transformation matrix, which is essentially the compressed version of the image, we can easily reconstruct the original image by applying some addition and subtraction techniques. We can also decrease the size or details of the image to be reconstructed to a significant level by using a process called thresholding.

Thresholding involves setting a non-negative threshold value that helps us reduce the amount of information in the detail elements, thereby achieving compression. After obtaining the detail elements, any absolute value of the detail value that falls below this threshold is considered insignificant or negligible and can be set to zero.

Let us define the threshold value of as 20, then a matrix D is obtained after thresholding T

$$D = \begin{bmatrix} 1212 & -306 & -146 & -54 & -24 & -68 & -40 & 0 \\ 30 & 36 & -90 & 0 & 0 & 0 & 0 & 0 \\ -50 & 0 & 0 & -24 & 0 & 72 & 0 & 0 \\ 82 & 38 & -24 & 68 & 48 & -64 & 32 & 0 \\ 0 & 0 & -32 & 0 & -48 & -48 & 0 & 0 \\ 0 & 0 & -56 & 0 & 0 & 32 & 0 & 0 \\ 0 & 0 & -48 & 0 & 0 & 0 & 0 & 0 \\ 44 & 36 & 0 & 8 & 80 & 0 & 0 & 0 \end{bmatrix}$$

After applying the inverse transformation on the matrix D we get the another matrix that has nearly similar values as the matrix P(of course this depends on selecting the threshold value) but when the image is reconstructed from the new matrix its size going to be significantly less than the original image at the cost of quality of the image. The quality of the new image will be quite less than the quality of the original image because we have ignored the detail values that are less than 20. Here is the new matrix that is obtained after reconstructing.

$$R = \begin{bmatrix} 582 & 726 & 1146 & 1234 & 1344 & 1424 & 1540 & 1540 \\ 742 & 694 & 1178 & 1074 & 1344 & 1424 & 1540 & 1540 \\ 706 & 754 & 1206 & 1422 & 1492 & 1572 & 1592 & 1592 \\ 818 & 866 & 1030 & 1374 & 1492 & 1572 & 1592 & 1592 \\ 856 & 808 & 956 & 1220 & 1574 & 1590 & 1554 & 1554 \\ 952 & 904 & 860 & 1124 & 1574 & 1590 & 1554 & 1554 \\ 776 & 760 & 826 & 836 & 1294 & 1438 & 1610 & 1610 \\ 456 & 760 & 668 & 676 & 1278 & 1422 & 1594 & 1594 \end{bmatrix}$$

Note that the threshold value determines the type of compression that we are dealing with. When the threshold value is set to zero that means no detail value is ignored thus, when the image is reconstructed back, you get the same image with the same quality and same size this is called as lossless compression of image and when the threshold value is not equal to zero, the reconstructed image now has lower quality than the original image, this is called as lossy compression. Thus Haar Wavelet Transformation is used to compute both Lossless and Lossy compression of images.

Compression Ratio

The Compression Ratio is defined as the Ratio of number of Non-Zero elements in the original image to the number of non zero elements in the Compressed Image.If we choose our threshold to be positive (i.e. greater than zero), then some entries of the transformed matrix will be reset to zero and there-fore some detail will be lost when the image is decompressed. The key issue is then to choose wisely so that the compression is done effectively with a minimum damage to the picture.

Linear Algebra in Haar Wavelet Transformation:

Change of Basis

Let us say P be the vector or pixel form of the some grayscale image, the vector

$$P_i = \begin{bmatrix} p_1 & p_2 & p_3 & \dots & p_n \end{bmatrix}$$

be a row in the matrix P. If the image has a colour block or nearly the same colour at a spot then the grayscale value of the values near that spot will be the same and if the vector is represented in the standard basis form, we will get that

$$P_i = p_1 \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix} + p_2 \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \end{bmatrix} + \dots + p_n \begin{bmatrix} 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Now since there will be a small block of colour in every image thus leading to some of the values of pi being nearly the same. The standard basis which gives the value of every pixel makes no use of the fact that we're getting a whole lot of pixels whose grey levels are nearly the same. When we try to transmit the image, if we take advantage of the fact that we have many redundancies in the pixel values, the data can be transmitted much more quickly.

So an ideal basis in this case would be the one which takes these redundancies into consideration. This is where the Haar Basis comes into picture. The Haar basis vectors are a set of orthogonal functions that form a basis for representing signals or images. In the case of one-dimensional signals, the Haar basis consists of square-shaped waveforms with alternating positive and negative values.The Haar wavelet transformation is a specific type of wavelet transformation that utilises the Haar basis vectors.

$$\begin{bmatrix} 1\\1\\1\\1\\1\\1\\1\\1 \end{bmatrix}, \begin{bmatrix} 1\\1\\1\\1\\-1\\-1\\-1\\-1 \end{bmatrix}, \begin{bmatrix} 1\\1\\-1\\-1\\0\\0\\0\\0 \end{bmatrix}, \begin{bmatrix} 0\\0\\0\\0\\1\\1\\-1\\-1 \end{bmatrix}, \begin{bmatrix} 1\\-1\\0\\0\\0\\0\\0\\0 \end{bmatrix}, \begin{bmatrix} 0\\0\\1\\-1\\0\\0\\0\\0 \end{bmatrix}, \begin{bmatrix} 0\\0\\0\\0\\1\\-1\\0\\0 \end{bmatrix}, \begin{bmatrix} 0\\0\\0\\0\\0\\0\\1\\-1 \end{bmatrix}$$

Haar Basis for an 8*8 dimension

Considering the 8*8 dimensions, let the Vector Pi in the basis be represented as follows,

$$\mathbf{P}_i = c_1 \begin{bmatrix} 1\\1\\1\\1\\-1\\-1\\-1\\-1 \end{bmatrix} + c_2 \begin{bmatrix} 1\\1\\-1\\-1\\0\\0\\0\\0 \end{bmatrix} + \dots + c_3 \begin{bmatrix} 0\\0\\0\\0\\0\\0\\1\\-1 \end{bmatrix}$$

Let a matrix H be such that its columns are the Haar basis vectors then the row vector P_i can be represented as product of the matrix H and another vector C such that

$$P_i = H \times C$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \; and \; C = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{bmatrix}$$

Note the values in c_1,c_2 ... ,c_8 represents the coefficients of the pixel(Pi) in the new basis, the values of c_i can be computed by inverting the Matrix H.

Since the Haar basis vectors are orthonormal to each other the values c_1,c_2,c_3,....c_8 can be easily computed by just multiplying both sides with the transpose of the Matrix H.

Observe that the Matrix H can be represented as the product of the three matrices as shown below.

$$\frac{1}{8} \times H = H_1 \times H_2 \times H_3, \; where, H_1 = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{-1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{-1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{-1}{2} \end{bmatrix}$$

$$H_2 = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{-1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{-1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} H_3 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Recall that in Averaging and Differentiating, we take a row vector and create pairs. The first four values are obtained by calculating the average of each pair, while the last four values are determined by calculating the difference of each pair. In the second level, we retain the last four values and repeat the same process. Specifically, we create pairs from the first four values, taking the first two as the average and the next two as the difference. In the third level, we repeat the same process by retaining the last six values and replacing the left-out two values with the average and difference.

Notice anything similar with the three product matrices of the Matrix H and the operations we are doing in Level 1, Level 2, and Level 3? Yes, the Level 1, Level 2, and Level 3 compression or decomposition of the matrix can also be computed by multiplying H_i with the original matrix. Note that the operations are to be done on the rows, and also the columns, to obtain the Level 1, Level 2, and Level 3 compression or decomposition of the matrix.

To achieve Level 1 compression or decomposition, we post-multiply the original matrix by H_1 and then pre-multiply it by the transpose of H_1, i.e., multiplying it with H_1 on the right side and multiplying it with the transpose of H_1 on the left side.

Similarly, to obtain the Level 2 compression or decomposition, we multiply the matrix by $H_1 \times H_2$ and multiply it at the front by the transpose of $H_1 \times H_2$.

To obtain the complete Haar Wavelet Transformation, the matrix should be multiplied by the product of $H_1 \times H_2 \times H_3$ and multiplied at the front by the transpose of the product of $H_1 \times H_2 \times H_3$.

Since, the Haar Wavelet Transformation of a matrix (assumed that the dimension is 8x8) is obtained by computing the three levels of compression or decomposition, and the Matrix $\frac{1}{8} \times H$ is the product of the three matrices, it can be said that the Haar Wavelet Transformation of an 8x8 matrix can be obtained from post-multiplying with the matrix $\frac{1}{8} \times H$ and pre-multiplying the matrix with its transpose (to apply the same operation to the columns) and that's why the matrix $\frac{1}{8} \times H$ is called the Haar Transformation Matrix.

The leading $\frac{1}{8}$ in the Haar Transformation Matrix is just for normalisation and we wont go much into that. Let A be a the corresponding array representation of the image and let B be another matrix such that,

$$B = H^T \times A \times H$$

Then the matrix B is the Haar Transformed Matrix of A and is the result of applying the Haar Transformation to A. Since the matrix B is the Haar Transformed Matrix of A, the matrix B is much more sparse than A and its transmission is much faster. The matrix A can be reconstructed from the Matrix B by multiplying with the inverse of H^T and H which are H and H^T respectively since the columns of the H matrix are Orthonormal.

$$NewA = (H^T)^{-1} \times B \times (H)^{-1}$$

The reconstructed matrix can be made into an image, If we had chosen lossless compression then the image quality would remain the same and if we chose the lossy compression that is, setting some threshold value and equating all the values in the Transformation matrix which are less than the threshold value, we get a reconstructed image which has less quality than the image.

Working Example:

The image below shows the initial image that we are going to work upon.

```
    # The Code below takes in an image and converts into gray scaled if its coloured
    # Resizes it to 512*512
    # And at last it reads the image as an array and saves it
    image = Image.open("grayscale.jpeg").convert("L")
    initial_size = image.size  # Store the initial size of the image
    desired_size = (512, 512)
    image = image.resize(desired_size, Image.ANTIALIAS)
    image_array = np.array(image)
```

The Updated Image below is resized to 512*512

As we discussed, we are going to restrict ourselves to the 8x8 dimension. Therefore, we will work with the 512x512 image by splitting it into blocks of 8x8 images and working on them individually. Now,We need to compute the Wavelet Transformtion of each 8x8 block Since Wavelet Transformation can just be computed by multiplying with the Haar Transformation Matrix, we store the required matrices i.e., the Haar Transformation Matrix, The Transpose of Haar Transformation matrix and the their respective inverses

The code below indexes upon the 512x512 array and makes blocks of 8x8 and we compute the Haar Wavelet Matrix and as per the given input of threshold value we make changes in the Haar Wavlet Matrix and update them in the main image matrix

```
# The code below indexes upon the 512x512 array and makes blocks of 8x8
# We compute the Haar Wavelet Matrix
# It is easily computed using the stored values of Haar Transformation Matrices
# As per the given input of threshold value we make changes in the Wavlet Matrix
# The Main image matrix is also updated at last
H = np.array([[0.125, 0.125, 0.25, 0, 0.5, 0, 0, 0],
        [0.125, 0.125, 0.25, 0, -0.5, 0, 0, 0],
        [0.125, 0.125, -0.25, 0, 0, 0.5, 0, 0],
        [0.125, 0.125, -0.25, 0, 0, -0.5, 0, 0],
        [0.125, -0.125, 0, 0.25, 0, 0, 0.5, 0],
        [0.125, -0.125, 0, 0.25, 0, 0, -0.5, 0],
        [0.125, -0.125, 0, 0.25, 0.5, 0, 0, 0.5],
        [0.125, -0.125, 0, -0.25, 0, 0, 0, -0.5]])
H_transpose = H.T
H_transpose_inverse = np.linalg.inv(H_transpose)
H_inverse = np.linalg.inv(H)
Threshold_value = float(input("Enter the Threshold Value: "))
block_size = 8

# Compression
compressed_image_array = np.copy(image_array)
for i in range(0, image_array.shape[0], block_size):
    for j in range(0, image_array.shape[1], block_size):
        A = image_array[i:i+block_size, j:j+block_size]
        B_Matrix = np.dot(H_transpose, A)
        B = np.dot(B_Matrix, H)
        for k in range(8):
            for l in range(8):
                if np.abs(B[k][l]) <= Threshold_value:
                    B[k][l] = 0
        final = np.dot(H_transpose_inverse, B)
        FinalA = np.dot(final, H_inverse)
        compressed_image_array[i:i+block_size, j:j+block_size] = FinalA
#Atlast we save the compressed image
compressed_image = Image.fromarray(compressed_image_array)
reconstructed_image = compressed_image.resize(initial_size, Image.ANTIALIAS)
# Reverting to original size
reconstructed_image.save("RECONSTRUCTED.jpeg")
```

The following images show the variation of their quality with respect to the input threshold values.

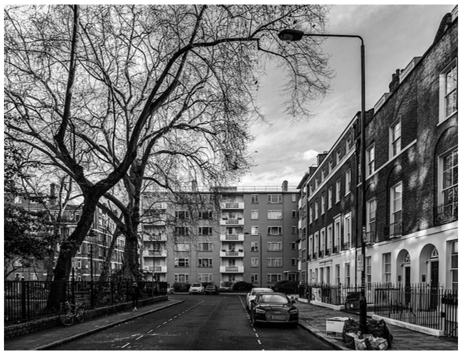
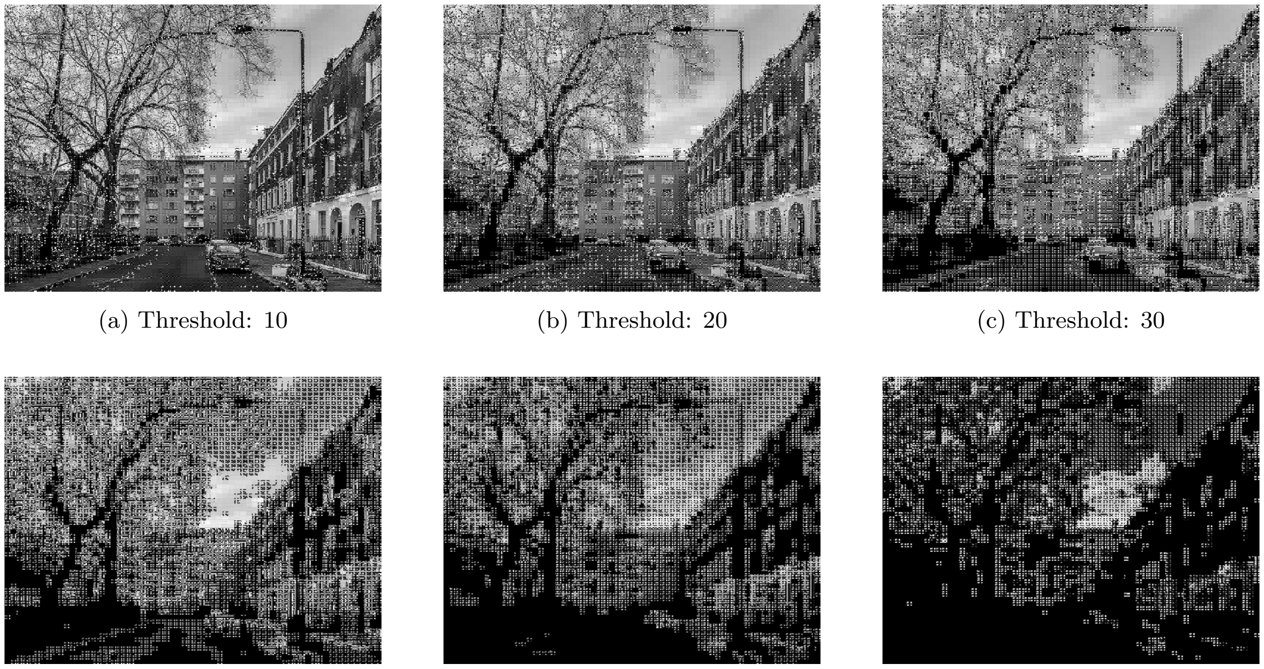


Figure 1: Threshold: 0

The above image is exactly the same as the original image. This is because the threshold value is set to zero, perfectly depicting lossless compression.


(a) Threshold: 10 (b) Threshold: 20 (c) Threshold: 30

(d) Threshold: 50 (e) Threshold: 70 (f) Threshold: 100

Figure 2: Quality of Images decreases with increasing threshold value

A common question arises regarding the presence of white dots or artifacts in the compressed image after applying thresholding. These white dots are a consequence of the lossy nature of the compression process.

Hard thresholding is the technique that we have used to set coefficients below a certain threshold to zero. It is a simple and direct approach to reduce the size of the transformed coefficients and achieve compression. However, thresholding, particularly with hard thresholding, can lead to abrupt changes and introduce artifacts in the reconstructed image.

The opposite approach to hard thresholding is soft thresholding, where we decrease the value of transformed coefficients if they are found to be less than the threshold. This results in a smoother compression process and eliminates the presence of white dots.

7.3 Direct Cosine Transformation

DCT stands for Discrete Cosine Transform, which is a mathematical technique used in signal and image processing to convert a signal or image from the spatial domain to the frequency domain. The DCT is closely related to the Discrete Wavelet Transform (DWT), as both are commonly employed in various applications such as data compression and feature extraction. Both DCT and DWT involve changing the basis representation, removing high-frequency components, and reconstructing the image, the specific techniques and properties of the two transforms differ, leading to different compression characteristics and applications.

The DCT is primarily used in compression standards such as JPEG. It transforms an image from the spatial domain to the frequency domain. The main steps involved in DCT-based compression are as follows:

- Divide the image into small, non-overlapping blocks.
- Apply the DCT to each block, which results in a frequency representation.
- Quantize the transformed coefficients by discarding less significant information.
- During reconstruction, reverse the steps to reconstruct the image, applying an inverse DCT.

You can notice that the base algorithm of DCT is same as DWT they work in a similar way but, they differ greatly in the basic functions they use.The DCT uses only cosine functions as its basis functions, while the DWT employs a set of wavelet functions as its basis functions. The cosine basis functions used in DCT are symmetric and oscillate at a single frequency, whereas the wavelet basis functions used in DWT have different scales and can capture localized features at different resolutions.

Below are the basic functions that are used in DCT for one dimension:

$$X(k) = \sqrt{\frac{2}{N}} \cdot C(k) \cdot \sum_{n=0}^{N-1} x(n) \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot k\right),$$
$$\text{for } k = 0 \text{ to } N-1,$$

and for 2-dimensional DCT is applied to image blocks, typically 8x8 or 16x16 pixels. The two-dimensional DCT is obtained by applying the one-dimensional DCT separately to each row and then to each column of the image block. The formula for the two-dimensional DCT of an image block f(x, y) of size N x N is:

$$F(u,v) = \sqrt{\frac{2}{N}} \cdot C(u) \cdot C(v) \cdot \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cdot \cos\left(\frac{\pi}{N} \cdot (x+0.5) \cdot u\right) \cdot \cos\left(\frac{\pi}{N} \cdot (y+0.5) \cdot v\right),$$
$$\text{for } u,v = 0 \text{ to } N-1,$$

where F(u, v) represents the transformed coefficients, C(u) and C(v) are the scaling factors, and (x, y) are the spatial coordinates of the image block. Similar to the one-dimensional DCT, F(0, 0) represents the DC component, and F(u, v) for u, v greater than 0 represents the AC components at different frequencies.

7.4 Vector Quantisation

In this section, we will explain the provided Python code for image compression using Vector Quantization. The code uses various libraries such as NumPy, PySimpleGUI, PIL, and sklearn.cluster.

The code includes several functions for performing Vector Quantization-based image compression:

- `generate_codebook(blocks, codebook_size)`: This function takes blocks of the image and a desired codebook size as input. It uses Mini-Batch K-means clustering to generate the codebook (representative vectors) based on the provided blocks.

```
def generate_codebook(blocks, codebook_size):
    # Flatten the blocks into vectors
    vectors = np.reshape(blocks, (-1, block_size * block_size * blocks.shape[3]))
```

```
        # Use Mini-Batch K-means clustering to generate the codebook
        kmeans = MiniBatchKMeans(n_clusters=codebook_size, random_state=0, batch_size=codebook_size
        kmeans.fit(vectors)
        codebook = kmeans.cluster_centers_

        # Reshape the codebook to match the shape of the blocks
        codebook = np.reshape(codebook, (codebook_size, block_size, block_size, blocks.shape[3]))

        return codebook
```

The function first flattens the blocks into vectors. Each block represents a pixel neighborhood in the image. The vectors are then used as input for the Mini-Batch K-means clustering algorithm, which generates the codebook (representative vectors) by grouping similar vectors together. The codebook is reshaped to match the shape of the blocks.

- `encode_image(blocks, codebook)`: This function encodes the image by finding the closest representative vector (codeword) from the codebook for each block. It returns the indices of the assigned codewords.

```
    def encode_image(blocks, codebook):
        indices = []
        for block in blocks:
            # Calculate the Euclidean distance between the block and each codeword
            distances = np.sum(np.square(block - codebook), axis=(1, 2, 3))

            # Find the index of the closest codeword
            index = np.argmin(distances)

            # Record the index of the assigned codeword
            indices.append(index)

        return indices
```

The function iterates over each block in the image and calculates the Euclidean distance between the block and each codeword in the codebook. This is done by subtracting the block from each codeword and squaring the result. The distances are then summed along the axes representing the spatial dimensions (width and height) and color channels. The index of the closest codeword is determined using `np.argmin`, and the indices of the assigned codewords are recorded.

- `load_image(image_path)`: This function loads an image from the specified `image_path` using the PIL library and converts it into a NumPy array.

```
    def load_image(image_path):
        image = Image.open(image_path)
        image = image.convert('RGB')
        image = np.array(image)

        return image
```

The function opens the image using `Image.open`, converts it to RGB color mode, and then converts it to a NumPy array.

- `save_image(image, save_path)`: This function saves the provided image array to the specified `save_path` using the PIL library.

```
    def save_image(image, save_path):
        # Convert the image array back to PIL Image object
        image = Image.fromarray(np.uint8(image))

        # Save the image
        image.save(save_path)
```

The function converts the image array back to a PIL Image object and saves it using the `save` method.

- `compress_image(image, codebook_size)`: This function performs the overall image compression process. It takes an image and the desired codebook size as input and returns the compressed image.

```
    def compress_image(image, codebook_size):
        # Preprocess the image into blocks
        blocks = preprocess_image(image)

        # Generate the initial codebook
        codebook = generate_codebook(blocks, codebook_size)

        # Encode the image using vector quantization
        indices = encode_image(blocks, codebook)

        # Decode the indices to reconstruct the compressed image
        reconstructed_image = decode_image(indices, codebook, image.shape[2], image.shape)

        # Remove singleton dimensions and convert to uint8 data type
        compressed_image = np.squeeze(reconstructed_image).astype(np.uint8)

        return compressed_image
```

The function first preprocesses the image by dividing it into blocks. Then, it generates the initial codebook using the `generate_codebook` function. Next, it encodes the image by finding the closest codeword for each block using the `encode_image` function. The indices of the assigned codewords are obtained. Finally, the indices are decoded to reconstruct the compressed image using the `decode_image` function. The reconstructed image is converted to the appropriate data type and the singleton dimensions are removed.

- `preprocess_image(image)`: This function preprocesses the image by splitting it into blocks.

```
    def preprocess_image(image):
        # Split the image into blocks
        blocks = []
        for i in range(0, image.shape[0], block_size):
            for j in range(0, image.shape[1], block_size):
                block = image[i:i+block_size, j:j+block_size, :]
                blocks.append(block)

        # Convert the blocks to a numpy array
        blocks = np.array(blocks)

        return blocks
```

The function iterates over the image in block-size increments and extracts each block. The blocks are then converted to a NumPy array.

- `decode_image(indices, codebook, num_channels, image_shape)`: This function decodes the indices to reconstruct the compressed image. It takes the indices, codebook, number of color channels, and the original image shape as input.

```
def decode_image(indices, codebook, num_channels, image_shape):
    reconstructed_blocks = []
    for index in indices:
        # Retrieve the corresponding codeword from the codebook
        codeword = codebook[index]

        # Reshape the codeword into a block shape
        block = np.reshape(codeword, (block_size, block_size, num_channels))

        # Append the reconstructed block
        reconstructed_blocks.append(block)

    # Convert the reconstructed blocks into an array
    reconstructed_image = np.array(reconstructed_blocks)

    # Reshape the reconstructed image to the original shape
    reconstructed_image = np.reshape(reconstructed_image, image_shape)

    return reconstructed_image
```

Retrieve the corresponding codeword from the codebook for each index: $\mathbf{codeword} = \mathbf{codebook}[\text{indices}]$

$$\text{Reshape the codeword into block shape: } \mathbf{block} = \text{reshape}(\mathbf{codeword})$$

The function iterates over each index and retrieves the corresponding codeword from the codebook. The codeword is reshaped into a block shape. The reconstructed blocks are then converted into a NumPy array and reshaped to match the original image shape.

7.5 Graphical User Interface (GUI)

The code also includes a GUI using the PySimpleGUI library to provide an interactive interface for compressing and saving images. The GUI layout consists of an image display for the original and compressed images, a file browse button to select an image, a compress button, and a save button.

The GUI functionality is implemented in the following code:

- The code defines a GUI layout using a list of lists, where each list represents a row of elements.
- The `compress_image_gui(image)` function compresses the provided image using the `compress_image` function and updates the GUI with the compressed image.
- The `draw_uniform_box(image)` function draws a uniform box around the provided image using the PIL library. This is used to visually distinguish the compressed image.
- The main event loop of the GUI continuously listens for events and handles them accordingly. Events such as closing the window, selecting a file, compressing the image, and saving the compressed image are handled using if conditions.
- When a file is selected, the original image is loaded and displayed in the GUI.
- Clicking the compress button triggers the compression process. This is done in a separate thread to keep the GUI responsive.

- After compression, the compressed image is displayed in the GUI, and the save button is enabled.
- Clicking the save button prompts the user to select a save path for the compressed image. The compressed image is then saved using the `save_image` function.

8 Conclusion

In conclusion, our image compression project explored various techniques including Singular Value Decomposition (SVD), wavelet transform, discrete cosine transform (DCT), and vector quantization. Each of these methods offers unique advantages and considerations for image compression.

SVD proved to be a powerful tool for dimensionality reduction and image approximation. By retaining only a subset of singular values, we were able to achieve significant compression while preserving the essential information in the image. SVD also offered a geometric interpretation of the image data, aiding in understanding the effect of the decomposition on the image's shape and orientation.

Wavelet transform provided a multiresolution analysis, allowing us to capture both local and global image features. By decomposing the image into different frequency bands, we achieved efficient compression by focusing on preserving the most relevant information in each band. The flexibility and adaptability of wavelet transform made it particularly effective in applications where preserving fine details is crucial.

DCT, on the other hand, offered a more compact representation of the image by transforming it into a frequency domain. By concentrating the majority of the image energy in a few low-frequency coefficients, DCT allowed for efficient compression with minimal loss of perceptual quality. DCT-based methods have been widely adopted in various image and video compression standards.

Vector quantization employed the concept of clustering to represent image patches or vectors with a reduced number of codewords. This approach allowed for efficient encoding and decoding by storing only the indices of codewords instead of the actual image data. Vector quantization performed well in scenarios where the emphasis was on achieving high compression ratios with acceptable quality degradation.

The provided code demonstrates the implementation of image compression using Vector Quantization. It includes functions for generating a codebook, encoding an image, loading and saving images, and the overall compression process. The code also incorporates a GUI for an interactive compression experience. The linear algebra concepts used include calculating Euclidean distances between vectors, clustering using K-means, reshaping arrays, and reconstructing the compressed image using codewords from the codebook.

Throughout our project, we observed that each technique had its strengths and limitations in terms of compression performance, computational complexity, and visual quality. The choice of the appropriate method depends on specific requirements, such as target compression ratio, acceptable loss in image quality, available computational resources, and the nature of the image data itself.

In conclusion, our exploration of SVD, wavelet transform, DCT, and vector quantization highlighted the versatility and effectiveness of these techniques in image compression. The understanding gained from this project provides a valuable foundation for further research and application of these methods in real-world image compression scenarios. By leveraging these techniques, we can achieve significant reductions in storage and transmission requirements while balancing the trade-off between compression efficiency and image fidelity.

9 Citation/Related Works

To better understand the current state of image compression techniques, we conducted a literature review and analyzed several research papers. Some notable works in this field include:

- Hu, Y., Chen, W., Lo, C. and Chuang, J.. "Improved vector quantization scheme for grayscale image compression" Opto-Electronics Review, vol. 20, no. 2, 2012, pp. 187-193. https://doi.org/10.2478/s11772-012-0016-z
- Khan, Sulaiman, et al. "An efficient JPEG image compression based on Haar wavelet transform, discrete cosine transform, and run length encoding techniques for advanced manufacturing processes." Measurement and Control 52.9-10 (2019): 1532-1544.
- Starosolski, Roman. "Hybrid adaptive lossless image compression based on discrete wavelet transform." Entropy 22.7 (2020): 751.
- Renkjumnong, Wasuta -., "SVD and PCA in Image Processing." Thesis, Georgia State University, 2007. doi: https://doi.org/10.57709/1059687

These papers provided valuable insights into the various techniques and algorithms employed in image compression, specifically those that utilize linear algebra.