Linear Algebra Project: Image Compression

International Institute of Information Technology, Hyderabad

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 Vector Quantization

1.3 DCT/Wavelet Transform

2 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.

3 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.

4 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.

5 Individual Work Contribution

Each team member will do the following works:

- Vinit Mehta: Prepared Interim Report in Latex. Will conduct the literature review, analyse research papers, and contribute to the implementation of the image compression techniques. Exploring the application of Singular Value Decomposition(SVD) and implementing the Imgae compression using Singular Value Decomposition algorithm.
- 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.

6 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.

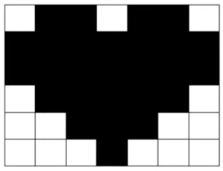
6.1 SVD

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

```
# 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.

```
# 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)

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

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

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

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.

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

Visualization

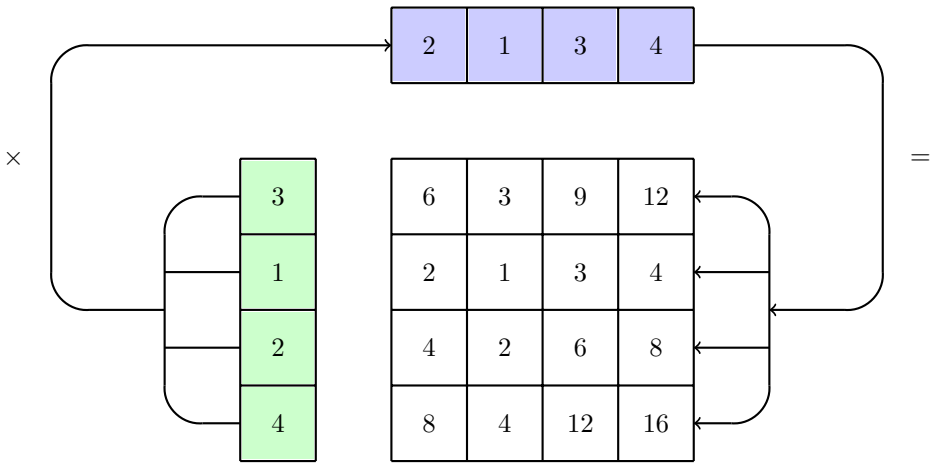


Fig: How outer product of two vectors is calculated

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

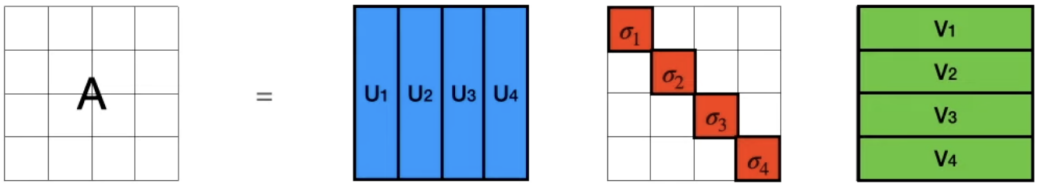
$$U\Sigma V^\top = \begin{bmatrix}\sigma_1\vec{u}_1 & \sigma_2\vec{u}_2 & \dots & \sigma_m\vec{u}_m\end{bmatrix} \begin{bmatrix}\vec{v}_1^\top \\ \vec{v}_2^\top \\ \vdots \\ \vec{v}_n^\top\end{bmatrix}$$

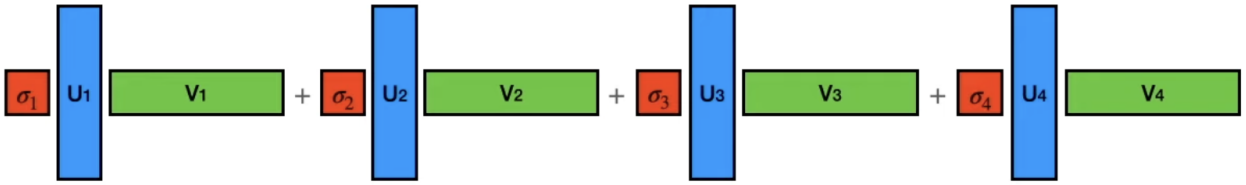
$$\therefore [U\Sigma V^\top]_i = \sigma_i(\vec{u}_i \otimes \vec{v}_i^\top)$$

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:

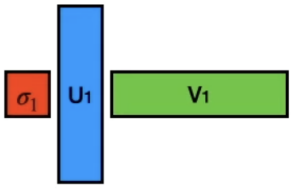
```python
# code for calculating the outer product of two vectors in python
# All the layers in python can be generated by this code (taking outer product)
# array to store all the layers
array_of_layers = []
for layer in range(len(U[0])): # number of layers = number of columns of U
    img_layer = [] # 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



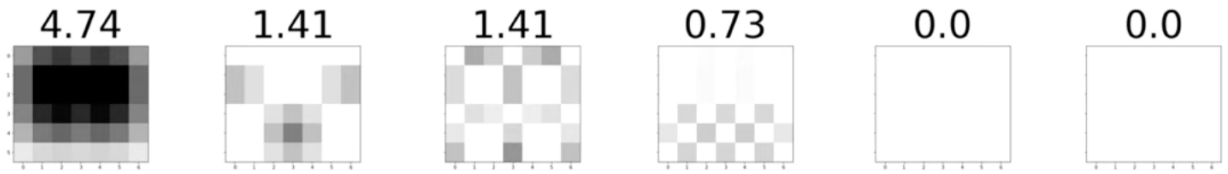
$$= (4.74) \begin{bmatrix} -0.36 \\ -0.54 \\ -0.54 \\ -0.45 \\ -0.28 \\ -0.08 \end{bmatrix} \begin{bmatrix} -0.23 & 0.5 & 0.39 & -0.22 & 0.56 & -0.42 & -0.12 \end{bmatrix}$$

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

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.

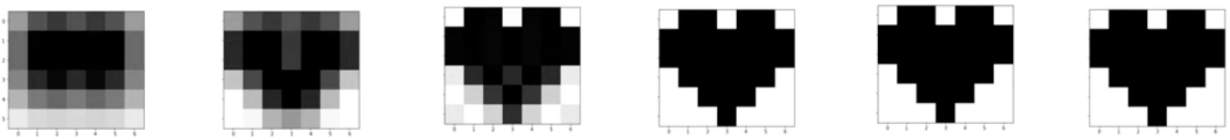
```
# All the layers in python can be generated by this code (taking outer product)
# array to store all the layers
array_of_layers = []
for layer in range(len(U[0])): # number of layers = number of columns of U
    img_layer = [] # each layer would be a 2D matrix
    colU = []
    rowV = V[layer]
 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



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:

```
# 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)
```



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 Key Objectives

7.1 SVD

7.2 DCT/Wavelet Transform

7.3 Vector Quantization

8 Conclusion

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
- Jolliffe Ian T. and Cadima Jorge 2016Principal component analysis: a review and recent developmentsPhil. Trans. R. Soc. A.3742015020220150202 http://doi.org/10.1098/rsta.2015.0202
- 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.