Singular Value Decomposition in Image Compression

Aritra Mukhopadhyay
(Roll. No.: 2011030)

November 16, 2022

1 Introduction

We work a lot with images in our daily life. We use them for various purposes like storing memories, sharing them with our friends and family, etc. But, the problem is that images are very large in size. For example, a 1080p uncompressed colour image is around 6 MB[1] in size. So, if we want to store a lot of images, we need a lot of storage space. Moreover, larger images will be read slower from the disk and any operations done on it will be slower too. Can we avoid this? Is there a way to store the same 6 MB image in a smaller space? In this project, we will discuss how we can compress images and store them in a smaller space using the singular value decomposition (SVD) method.

The SVD is a factorization of a real or complex matrices. It is widely used in numerical linear algebra, and is one of the most important matrix factorizations. The SVD is used to solve a wide range of problems in science, engineering, and mathematics. It is also used in **image compression**, recommender systems, and in numerical methods for solving partial differential equations. We will learn how to use SVD to compress image data with no or very small loss in quality.

2 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a method of factorising a matrix A (say) into three of its components.

$$A_{n\times n} = U_{n\times n} \cdot S_{n\times n} \cdot V_{n\times n}$$

S is a diagonal matrix with all non-diagonal elements zero. The diagonal elements of S are called the **singular values** of A. The columns of U and V are called the **left and right singular vectors** of A respectively.

Here, U and V are unitary matrices and S is a diagonal matrix. The diagonal elements of S are called the singular values of A. The columns of U are called the left singular vectors of A and the columns of V are called the right singular vectors of A. The singular values of A are the square roots of the eigenvalues of A^*A and AA^*.

Note that S is a diagonal matrix with all non-diagonal elements zero. So, we we can write S as a vector of singular values σ_i.

2.1 Building the Intuition

Let us start by having an intuitive idea of what we are trying to achieve. Consider this matrix:

$$A_{4\times 4} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 5 & 10 & 15 & 20 \\ 10 & 20 & 30 & 40 \end{pmatrix}$$

This is a (4×4) matrix. So it seems like, we need $(4 \times 4 =)$ 16 memory units to store the entire matrix in its true form.

On the other hand, we can clearly see that this matrix can be writen in a slightly different way:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 5 & 10 & 15 & 20 \\ 10 & 20 & 30 & 40 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 5 \\ 10 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

So, instead of storing the actual matrix, if we store these two matrices we won't need 16 memory units, only 8 will be sufficient. So, we have reduced the memory requirement by a factor of 2. Now, this was just a 4×4 matrix, imagine if we can express a 1000×1000 matrix (which was supposed to take 10^6 memory units) as a product of a row and a column matrix, we can store it using only 2000 units of memory (500 times reduction in size). This is what we are trying to achieve in this project.

Now obviously we won't always be fortunate enough to get a matrix which can be expressed as a product of a row and a column matrix, but it can be proved that for any matrix $A_{n\times n}$ we can always express it like this:

$$A_{n\times n} = \sum_{i=1}^{n} (\sigma_i C_i \times R_i)$$

Here, σ_i is a real number (where $\sigma_a > \sigma_b$ if $a < b$). C_i and R_i are column and row matrices respectively of sizes n.

[1]A 1080p image is 1920x1080 pixels. Each pixel has 3 values (RGB). Considering this as an 8 bit (= 1 byte) image:

$$(1920 \times 1080 \times 3)Bytes = 6220800Bytes \approx 6MB$$

Now, according to the theory, we need n terms to express a $n \times n$ matrix. But, in practical scenario, we can get away with much less terms. This is because, as the values of σ_i decrease, the contribution of the corresponding term to the matrix A also decreases.

3 How it Works

3.1 Grayscale Square Images

We will first try our hands upon a grayscale images. Grayscale images are images in which every pixel has a single brightness value. So it is easier to work with.

After Importing all the classes and functions[2] from the library file, we can make an instance of the **GrayscaleImageSVD** class. This class takes the location of the image as an input. When the instance is constructed, it loads the image to it's memory as a numpy array. Then it callculates the U, S and V matrices using the **numpy.linalg.svd()** function from the **NumPy** library.

Using numpy.linalg.svd() function: We can use the *numpy.linalg.svd*() function to compute the SVD of a matrix. The function returns the 3 matrices: U, S and V. In the S matrix, we get the singular values in descending order. The function returns the matrix S not in the form of a diagonal matrix but as a vector of singular values. So, we will have to convert it into a diagonal matrix (by *numpy.diag*() function) before we can use it to reconstruct the matrix A.

```python
from library.DIY import GrayscaleImageSVD
pic = "monkey"
img = GrayscaleImageSVD(f"static/{pic}.jpg")
img.display("Original Square Grayscale Image")

```

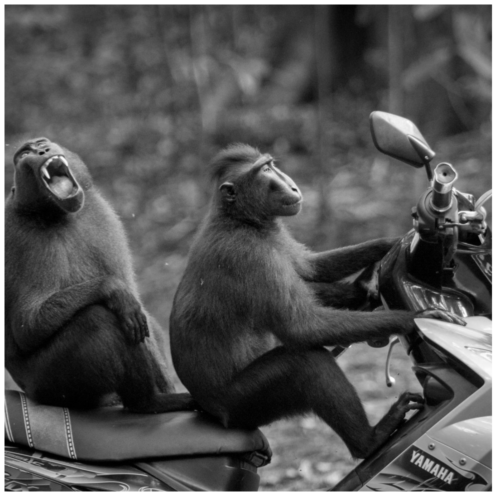
Figure 1: Original Square Grayscale Image

Next, we can call the **GrayscaleImageSVD.reduce()** method to get the reduced image. This function takes the number of singular values to be used as an input. It then uses the first n singular values to reconstruct the image. The function returns the reconstructed image as a numpy array. We can then use the **show_image()** function to display the image. Note that, the returned array is not smaller in size than the original image; it is a full sized image. But this array can be constructed using arrays of much smaller sizes.

Although we have writen our own SVD function (demonstration **here**), we won't be actually using that for this project.

Why are we using the SVD function from numpy and not using the SVD function which is our own? This is because the SVD function from numpy is written in C and is highly optimized. So, it is much faster than our own implementation. Moreover finding SVD of a matrix involves finding eigenvalues and eigenvectors of the matrix. We know that the algorithms studied in this class become inefficient if we are working with a large matrix. The later values have a lot of error in them. So, writing the SVD function is itself a huge problem to be solved. On the other hand, finding SVD of a matrix is a very well studied problem in literature and there are many efficient algorithms to find it. So, we can use the SVD function from numpy without reinventing the wheel.

[2] All the relevant functions and classes have been kept in: the Library File