Singular Value Decomposition in Image Compression

Aritra Mukhopadhyay
(Roll. No.: 2011030)

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:

$$A_{4\times4} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 5 & 10 & 15 & 20 \\ 10 & 20 & 30 & 40 \end{pmatrix}$$

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.

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

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

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

of σ_i decrease, the contribution of the corresponding term to the matrix A also decreases.

2.2 SVD of rectagular matrices

Till now we had been working with square matrices only. Now let's see how we can apply SVD to a rectangular matrix. Consider the following $n \times m$ matrix A:

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

Here, the S matrix actually had $min(m,\ n)$ number of non-zero elements as it's diagonal elements. All other elements are 0. So, depending the values of m and n, the S matrix will be of the following forms

When $m < n$:

$$S_{n\times m} = \begin{pmatrix} \sigma_{11} & 0 & \cdots & 0 \\ 0 & \sigma_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{mm} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

Here, while multiplying S with V. We will only need the first m columns of the V matrix. The rest of the columns are anyway going to be multiplied by zero. So, we can discard them. Similarly, while multiplying U with S, we will only need the first m rows of the U matrix. The rest of the rows are going to be multiplied by zero. So, we can discard them.

When $n < m$:

$$S_{n\times m} = \begin{pmatrix} \sigma_{11} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_{22} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_{nn} & 0 & \cdots & 0 \end{pmatrix}$$

Similarly, here we can clearly see that we only need the first n rows and columns of U and V respectively. So, we can discard the rest of the rows and columns.

So, practically, by getting rid of all the unnecessary parts, we can emperically write the following equation:

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

Now, while compression, let us use only the first k rows and columns of U and V respectively. So, we can tweak the above equation to:

$$A_{n\times m} = U_{n\times k} \cdot S_{k\times k} \cdot V_{k\times m} \quad [where\ k \leq m]$$

2.3 Colour Images

Colour images have 3 values for every pixel – Red(R), Green(G) and Blue(B). It can be thought of as 3 grayscale images. So, we can apply SVD to each of the 3 images separately. This will give us 3 matrices each for R, G and B. We can do all the similar operations on all these 3 images and then while displaying, combine them to get the final image.

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.

```
1 from library.DIY import GrayscaleImageSVD,
    show_image
2 pic = "monkey"
3 img = GrayscaleImageSVD(f"static/{pic}.jpg")
4 img.display("Original Square Grayscale Image")
```

Listing 1: Creating GrayscaleImageSVD class

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

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

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 takes it back to C for doing the calculations and is highly optimized. So, it is much faster than our own implementation (our function is almost 2x slower than the numpy 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 `numpy.linalg.svd()` function from numpy without reinventing the wheel.

```
n_terms = 10  # taking upto 10 terms
A2, ratio, error = img.reduce(terms = n_terms)
print(f"size = {ratio}% of original image")
print(f"RMS deviation in pixel values from the
  original is: {error}")
show_image(A2)
```

Listing 2: Reducing the image

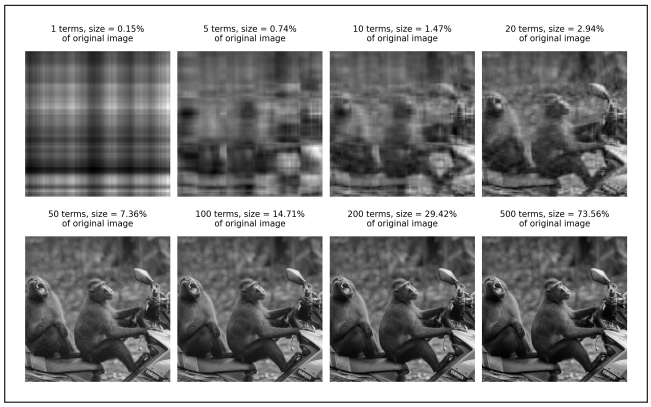

Figure 2: **Reduced image comparison with different number of terms and sizes**.

(Above comparison image might be of low quality. Find HD version **here**.)

3.1.1 Discussion

As we can clearly see from the images above, our image compression is a tradeoff with quality of the image. So, how many terms should we take? The answer to this question solely depends upon the application.

- **1 term:** We can only see some illuminated lines in the image.
- **5 terms:** Now, the image definitely contains more information, but no clear structure can be seen.
- **10 terms:** Now, we can see some structure in the image. But the image is still very noisy.
- **20 terms:** Now, the structures in the image are more clear. One can tell that it is a monkey riding a scooty.
- **50 terms:** At this point, the image is pretty clear. The name on the scooty is also readable, but the image lacs details.
- **100 and 200 terms:** The background becomes less noisy. Details like individual strands of hair become clearer.
- **500 terms:** This image is almost as good as the original image. No difference can be seen on the picture alone. If one compares it side by side with the original image, they can easily spot a difference in the contrast of the two images.

3.2 Grayscale Rectangular Images

The `GrayscaleImageSVD()` function is a generalised one. It works on both square and rectangular grayscale images. Let's see how it looks on a rectangular image.

The original lion image can be found **here**.

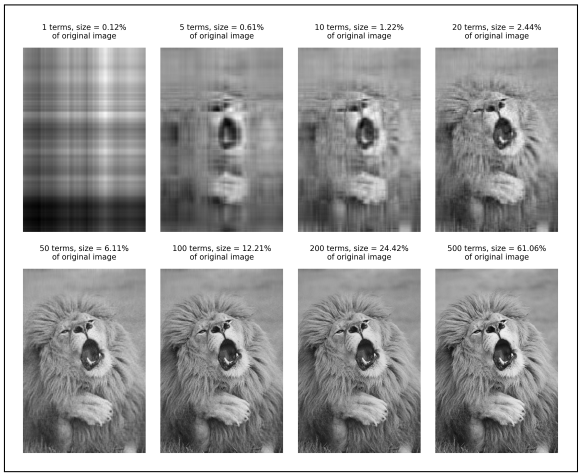

Figure 3: **Reduced rectangular image comparison with different number of terms and sizes**.

(Above comparison image might be of low quality. Find HD version **here**.)

3.3 Color Images

Colour images consist of 3 channels of data: Red, Green and Blue. So, for colour images we need to perform SVD on each channel separately. The `ColourImageSVD()` class does exactly that. Let's see how it looks on a color image.

```
import library.DIY as DIY
img = DIY.ColourImageSVD(f"static/crab.jpg")
img.display("Original Square Grayscale Image")
A2, ratio, error = img.reduce(terms = 10)
DIY.show_image(A2)
```

Listing 3: Working with colour images

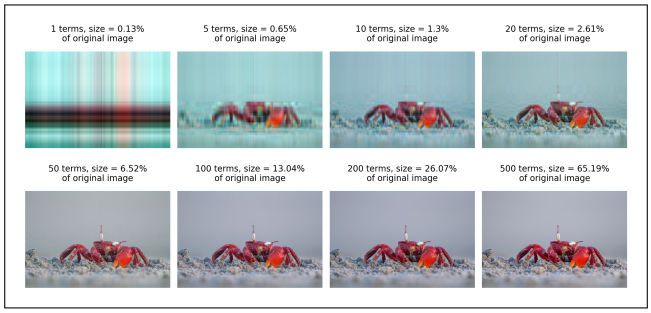

Figure 4: **Reduced colour image comparison with different number of terms and sizes**.

(Above image might be of low quality. Original image can be found **here**.)

4 Error Analysis

We can find the RMS error in the reconstrcted matrix by using the following formula:

$$\text{RMS Error} = \sqrt{\frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij} - A_{ij}^{r})^2}$$

Here, A_{ij} is the original matrix and A_{ij}^{r} is the reconstructed matrix.

$$\%\ \text{RMS Error} = \frac{\text{RMS Error}}{256}100\%$$

We divide the RMS error by 256 because the pixel values range from 0 to 255. So, the RMS error will be in the range of 0 to 255. We multiply the RMS error by 100 to get the percentage error.

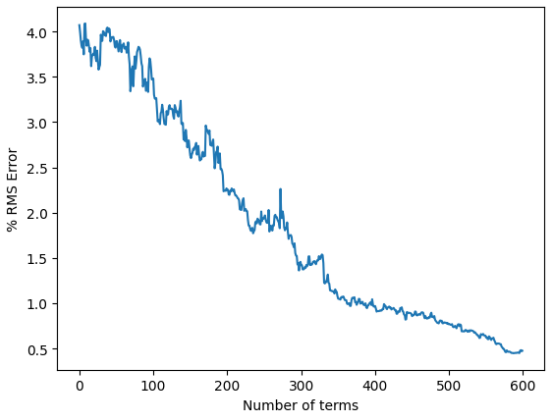

Figure 5: number of terms taken vs % RMS error for the grayscale monkey image

We have written a `get_rms_error()` function for calculating the percentage RMS error. It takes the original matrix and the reconstructed matrix as input and returns the percentage RMS error.

We have created a graph for the percentage RMS error vs the number of singular values (terms) used. The graph for the grayscale monkey image is shown in **Figure 5** and for the color monkey image is shown in **Figure 6**.

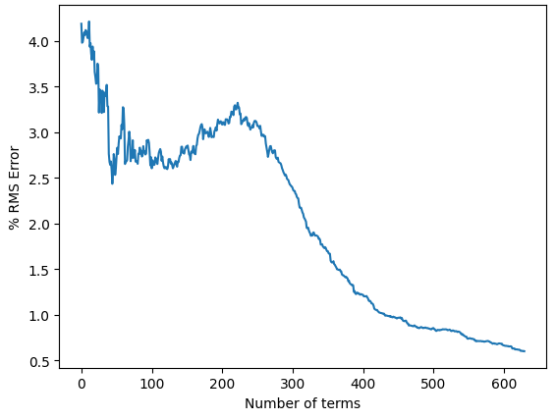

Figure 6: number of terms taken vs % RMS error for the colour crab image

5 Resources

- Library file at ./library/DIY.py
- `SVD()` function demonstration at DIY Project/trying_SVD.ipynb
- All other bits of codes I had to write for this project are in ./DIY.ipynb

6 Conclusion

References

[pho,] Comedy wildlife photography awards. `https://www.facebook.com/comedywildlifephotoawards/`.

[wik, 2022] (2022). Singular value decomposition. `https://en.wikipedia.org/wiki/Singular_value_decomposition`.

[from Serrano.Academy, 2020] from Serrano.Academy, L. S. (2020). Singular value decomposition (svd) and image compression. `https://www.youtube.com/watch?v=DG7YTlGnCEo&t=824s`.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.