Software Assignment Report

Image Compression

Vishwambhar - EE25BTECH11025

November 8, 2025

Singular Value Decomposition(SVD)

Singular Value Decomposition (SVD) is one of the most fundamental matrix factorizations in linear algebra and forms the foundation of this project. It expresses any matrix A as the product of three matrices U, Σ, and V^T:

$$A = U\Sigma V^T \tag{1}$$

where U and V are orthogonal matrices and Σ is diagonal.

The conceptual understanding presented here is based on the lecture by **Prof. Gilbert Strang (MIT)** [1], which provides a geometric interpretation of SVD as a transformation that stretches and rotates vectors along orthogonal directions. According to this explanation, the singular values in Σ represent the magnitudes of these stretchings, while the columns of U and V represent orthogonal bases for the output and input spaces, respectively.

Summary of the lecture: Prof. Strang demonstrates how SVD uncovers the fundamental structure of any linear transformation, showing the relationship between the range, null space, and rank of a matrix. This perspective was particularly helpful in understanding how image matrices can be compressed effectively by retaining only the largest singular values.

Algorithms to compute SVD

1.Power Iteration

Iteratively multiplying a random vector by A^TA to converge to dominanat right singular vectors and also using deflation for the matrix A^TA.

Pros

1. Simple and compact, minimal code is required.
2. Good for just a few top singular values

Cons

1. Slow when eigen values are almost same.
2. Needs normalizations to avoid overflow.

2.Jacobi Method

Iteratively orthogonalizing a vector with Jacobi rotations until they become right singular vectors.

Pros

1. Good accuracy.
2. Vectors are accurately orthogonal.

Cons
1. Slower, since has to find all k values.
2. More iterations required if singular values are almost same.

3. Golub-Reinsch algorithm

It first simplifies the matrix to a bidiagonal form using orthogonal transformations, then applies a fast QR- based method to extract the singular values and vectors.
Pros
1. Very stable and accurate.
2. Works for any dense matrix.
Cons
1. Computationally heavy for very large matrices.
2. Uses more memory.

4. Divide and Conquer SVD

This algorithm improves the classical Golub–Reinsch SVD by dividing the bidiagonal matrix into smaller subproblems, computing the SVD of each part separately, and then combining the results through a secular equation.
Pros
1. Much faster
2. Good accuracy
Cons
1. High memory usage.
2. More complex implementation.

Power method and its Advantages

The Power Method is an iterative algorithm used to compute the largest singular values and corresponsing singular vectors of a matrix.
It repeatedly multiplies a random vector by A^TA, normalizes it, and gradually aligns the vector with the direction of the dominant right singular vector.
The correspond singular value is obtained from the norm of $A\vec{v}$, and the left singular vector is derived as $\vec{u} = \frac{A\vec{v}}{\sigma}$.
By applying a deflation step, the algorithm can extract multiple singular values successively.

Advantages of Power method:

- Simple to implement(only uses matrix-vector multiplications).

- Fast if you only need a few singular values
- Low memory usage - unlike QR and divide-and-conquer it doesn't store big matrices.
- Can be adapted to sparse or large matrices easily.

How and Why I used the Power method?

In this project, the power method was applied to compute a truncated SVD for image compression.
Each color channel of the image was treated as a separate matrix.
The algorithm iteratively multiplied a random vector by A^TA to extract the dominant right singular vector, and used it to compute the corresponding singular value and left singular vector.
This process was repeated k times(with deflation) to obtain the top k singular values that represent the most significant features of the image.
The compresses image was reconstructed as $A_k = U_k\Sigma_k V_k$, and combining all three channels produced the final RGB output.
By varying k, we controlled the trade-off between image quality and compression level - smaller k resulted in higher compression but lower detail, while larger k produced near-original quality.

Why?

- **Simple and effective** - easy to code and understand.
- **Captures main features** - the top singular values preserve most of the image energy.
- **Compression control** - by changing k, I can control quality vs. size.
- **Memory-efficient** - Stores only a few vectors.
- **Not suitable** - if i have to find all singular values.

Explanation of the Code and the Math behind.

I have used the hybrid(C + python) option to compress the image. Python is used to read the iamge and store the values of intensity of each colour of pixels in a 3D matrix. And the C code is used to implement the logic of Power method as discussed above to find SVD. Python is also used to compare the runtime and accuracy of my algorithm and the standard python command(np.linalg.svd).(The codes are present in the codes folder)

Math behind the code

The task of the gievn code is to perform image compression using SVD.

1. SVD

Any real matrix A of size $m \times n$ can be decomposed into three matrices:

$$A = U\Sigma V^T \tag{2}$$

2. Connection between SVD and Eigen-decomposition

The singular vectors and values of A are directly related to the eigen vectors and eigenvalues of A^TA and AA^T.

- The columns of V are the eigen vectors of A^TA.
- The columns of U are the eigen vectorso f AA^T.
- The non-zero singular values in Σ are the square roots of the non-zero eigen values of both A^TA and AA^T.

3. The power Iteration Method

To find the eigenvalues and eigenvectorsof A^TA, the code uses the power iteration method. The algorithm works as follows:

1. Start with a random vector $\vec{b}$
2. In each iteration k, update the vector by multiplying it with the matrix $B(B = A^TA)$:$\vec{b}_{k+1} = B\vec{b}_k$.
3. Normalize the resulting vector to prevent it from growing or shrinking to zero.
4. After a sufficient number of iterations, the vector $\vec{b}_k$ will converge to the eigenvector correspponding to the dominant eigenvalue of B.
5. The dominant eigenvalue can then be calculated by using:

$$\lambda = \frac{\vec{b}_k^T B \vec{b}_k}{\vec{b}_k^T \vec{b}} \tag{3}$$

4. Deflation

Once the dominant eigenvalue is found, the code uses a technique called deflation. It modifies the original matrix B to remove the influence of the found eigenvector, so that the power method, when applied to the modified matrix, will conerge to the next dominant eigenvector.

The deflation is performed as shown below:

$$B_{def} = B - \lambda_1 \vec{v}_1 \vec{v}^T \tag{4}$$

5. Reconstructing the image

An image can be represented as a sum of rank-one matrices, formed by the singular vectors and values:

$$A = \sigma_1 \vec{u}_1 \vec{v}_1^T + \sigma_2 \vec{u}_2 \vec{v}_2^T + \dots \tag{5}$$

By keeping only the top k singular values and their corresponding vectors, we can create a low-rank approximation of the original image:

$$A_k \approx \sigma_1 \vec{u}_1 \vec{v}_1^T + \sigma_2 \vec{u}_2 \vec{v}_2^T + \cdots + \sigma_k \vec{u}_k \vec{v}_k^T \tag{6}$$

$$A_k \approx U_k \Sigma_k V_k^T \tag{7}$$

This compressed representation requires significantly less storage than the original image, and the reconstructed image is often visually very similar.

Pseudo Code

The following is the pseudo code for important function in C find_svd.

```
1    FUNCTION find_svd(Input_Image_Data A, rows r, columns c,
   needed_eigen_vales k, channels ch,  Output_Image_Data res){
2    For each channel{
3        A_ch = A_pointer = to start of respective channel's matrix of A
4        R = R_pointer = to start of respective channel's matrix of res
5        Allocate memory to B = B_Matrix
6        Allocate memory to B_def = A_Deflated_Matrix
7        Allocate memory to U = Left_Singular_Matrix
8        Allocate memory to V = Right_Singular_Matrix
9        Allocate memory to S = Singular_Matrix
10       Allocate memory to eigenvec = Right_singular_Vector
11       Allocate memory to left_s_vec = Left_Singular_Vector
12
13       Calling FUNCTION A transpose A
14       Calling copy_Matrix FUNCTION
15
16       FOR Index i FOR i less than k{
17           Eigenvalue Lambda = Calling Power_method FUNCTION
18           ith Index of S = square root of Lambda
19           FOR Index j FOR j less than c{
20               jth row and ith column of V = jth element of eigenvec
21           }
22
23           FOR Index p FOR p less than c{
24               FOR Index j FOR j less than c{
25                   pth row and jth column of B_def -= Lambda*(pth
   element of eigenvec)* (jth element of eigenvec)
26               }
27           }
28       }
29       FOR Index i FOR i less than k{
30           FOR Index j FOR j less than c{
31               eigenvec = ith column of V
32           }
33           left_s_vec = multiply A_ch and eigenvec
34           Normalize left_s_vec
35           FOR Index j FOR j<r{
36               ith column of U = left_s_vec
37           }
38       }
39       FOR Index i FOR i less than r{
```

```
            FOR Index j FOR j less than c{
                sum = 0
                FOR Index p FOR p less than k{
                    sum += (element of ith row and pth column of U)*(
   pth element of S)*(element of jth row and pth column of V)
                    (element of ith row and jth column of R) = sum
                }
            }
        }
        Freeing all the allocated memory
    }
    }
```

Python Code Explanation

1. Load libraries
The code starts by importing necessary Python libraries: ctypes for interfacing with C, numpy for numerical operations, matplotlib for plotting results, and PIL for image handling.
2. Interface with C code
The code loads the shared C library and specifies the data types of the arguments for the C function find_svd. This ensures that Python and C can correctly exchange data.
3. Image loading
An image file is opened, converted to the RGB color model, and then transformed into a numpy array of floating point numbers, which is a format suitable for mathematical computations.
And it also checks whether the given image is greyscale or color and does the calculation according to that.
4. Visual Demonstration
The code then generates a series of compressed images. It iterates through a list of k values.
5. Performance and Accuracy Analysis
Explained in the next section.
6. Output Results
The code then prints the runtime of both the C and numpy methods, as well as the calculated errors. It also displays the visual comparison of the original and compressed images.
It also prints forbenius error for each value of k for selected image as shown in the above sample outputs.

Performance and Accuracy Analysis

Runtime

I used the time library in python to compare the runtime(for $k = 50$) of my algorithm and python standard command.

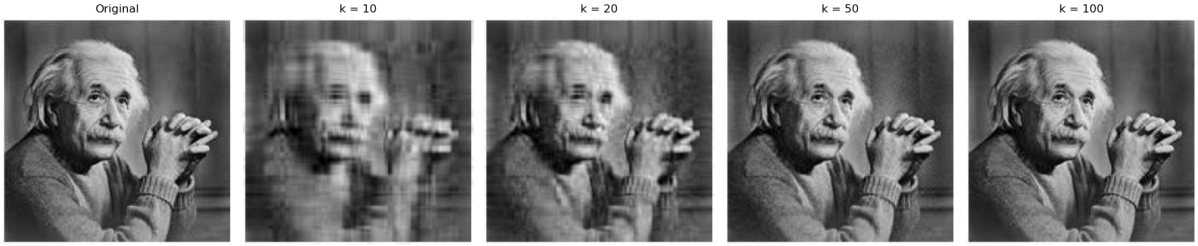
Error

I also used python to calculate the relative error and mean squared error between my algorithm and python standard command.Formulas that I used are as follows:

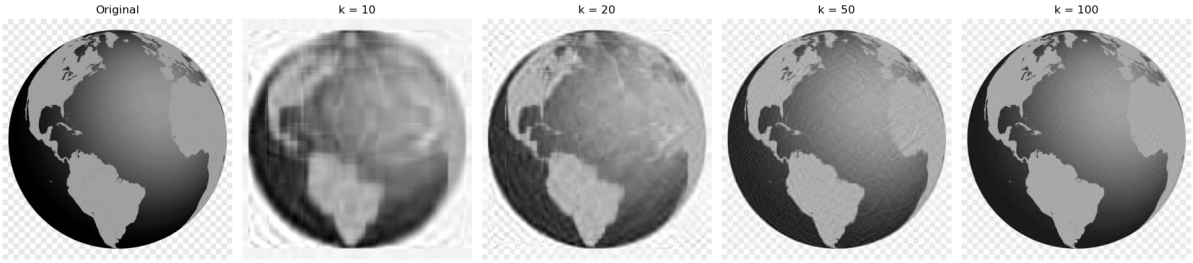
1. Mean Sqaured Error $= \frac{\Sigma_{i=1}^{N}(a_{np}-a_{Ci})^2}{N}$
2. Relative Error $= \frac{||A_{np}-A_C||_F}{||A_{np}||_F}$

Sample outputs

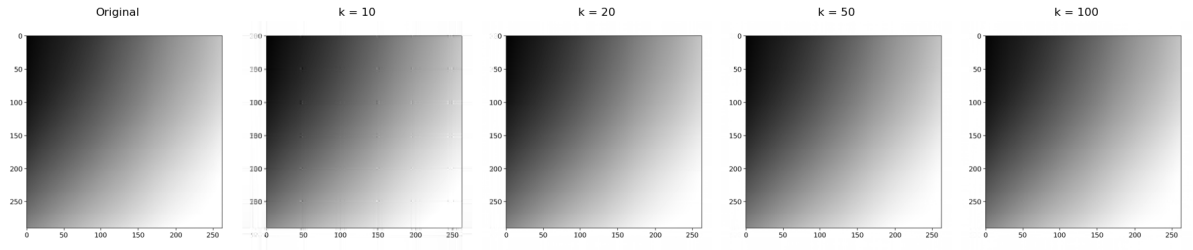

```
Please enter which image do you want to compress(DONT FORGET TO ENTER ALONG WITH IMAGE TYPE! i.e., .jpg/.png/.jpeg): flower.png
Enter whether the given image is color or greyscale type: color

----The following are the frobenius errors for the chosen k values----
Frobenius Error of the image for k = 10 is 16851.704311190926
Frobenius Error of the image for k = 20 is 10868.12399434431
Frobenius Error of the image for k = 50 is 5501.228529452448
Frobenius Error of the image for k = 100 is 2534.17979907418

----The following are the stats to compare between my algorithm and python standard command----
C Power Method runtime: 5.0698 s
NumPy SVD runtime: 0.6465 s
Mean Squared Error: 75.574688
Relative Error: 5.140426e-02
```



```
Please enter which image do you want to compress(DONT FORGET TO ENTER ALONG WITH IMAGE TYPE! i.e., .jpg/.png/.jpeg): einstein.jpg
Enter whether the given image is color or greyscale type: greyscale

----The following are the frobenius errors for the chosen k values----
Frobenius Error of the image for k = 10 is 3256.367331051335
Frobenius Error of the image for k = 20 is 2137.106609597725
Frobenius Error of the image for k = 50 is 886.6559095607082
Frobenius Error of the image for k = 100 is 563.1380186714294

----The following are the stats to compare between my algorithm and python standard command----
C Power Method runtime: 0.1656 s
NumPy SVD runtime: 0.0249 s
Mean Squared Error: 3.107933
Relative Error: 1.488485e-02
```




```
Please enter which image do you want to compress(DONT FORGET TO ENTER ALONG WITH IMAGE TYPE! i.e., .jpg/.png/.jpeg): globe.jpg
Enter whether the given image is color or greyscale type: greyscale

----The following are the frobenius errors for the chosen k values----
Frobenius Error of the image for k = 10 is 15071.10034774486
Frobenius Error of the image for k = 20 is 10649.779168539842
Frobenius Error of the image for k = 50 is 6201.8364730024
Frobenius Error of the image for k = 100 is 3704.5879176239378

----The following are the stats to compare between my algorithm and python standard command----
C Power Method runtime: 5.1147 s
NumPy SVD runtime: 1.9425 s
Mean Squared Error: 125.589119
Relative Error: 6.087327e-02
```




```
Please enter which image do you want to compress(DONT FORGET TO ENTER ALONG WITH IMAGE TYPE! i.e., .jpg/.png/.jpeg): greyscale.png
Enter whether the given image is color or greyscale type: greyscale

----The following are the frobenius errors for the chosen k values----
Frobenius Error of the image for k = 10 is 7190.343508068035
Frobenius Error of the image for k = 20 is 3809.385500845386
Frobenius Error of the image for k = 50 is 1181.0557843772513
Frobenius Error of the image for k = 100 is 694.5397236749487

----The following are the stats to compare between my algorithm and python standard command----
C Power Method runtime: 11.3013 s
NumPy SVD runtime: 3.6594 s
Mean Squared Error: 0.121125
Relative Error: 1.838758e-03
```

Frobenius Norm

The logic to compute Frobenius norm is written in C and the python code calls the function and prints the error as shown in the above images.

$$||A - A_k|| \tag{8}$$

$$||.||_F = \sqrt{\Sigma_{i=1}^{m}\Sigma_{j=1}^{n}(a_{ij} - a_{k(ij)})^2} \tag{9}$$

S.No	Value of k	Flower	Einstein	Globe	Greyscale
1	10	16851.5585	3256.6316	15071.1003	7190.3435
2	20	10868.6316	2137.8577	10649.7791	3809.3855
3	50	5501.2285	886.6302	6201.8364	1181.0557
4	100	2534.1797	563.4606	3704.58791	694.5397

Conclusion

This project implemented SVD for image compression using Power Method in C, integrated with python for visualization and analysis. Image compression is useful for storing data in less space.
From above sample outputs we can see that smaller values of k yield images with higher compression and lower picture quality, whereas larger values of k reproduces almost the same image therefore reducing compression. Although finding singular values using power method is not as efficient as using numpy's SVD, it helps in understanding the mathematical background of computing SVD and using it for image compression.
Overall, the project demonstrated how SVD can be used for image compression while retaining image quality, and also offered insights into iterative methods and matrix based image processing.

References

[1] G. Strang, "The Singular Value Decomposition (SVD)," *MIT OpenCourseWare: Linear Algebra (18.06)*, Lecture 29, Massachusetts Institute of Technology, Available at: `https://www.youtube.com/watch?v=YzZUIYRCE38`