Course Project (Software): Image Compression using Truncated SVD

EE25BTECH11055 - Subhodeep Chakraborty

I. INTRODUCTION

The project's idea is to perform rudimentary compression on an image by representing it as a matrix (or in the case of a colour image, 3 matrices) and applying SVD upon it. By keeping only the top k singular values, a low rank approximation of the image matrix can be obtained, which upon conversion back to the source format results in a smaller file size due to the format's compressive algorithms.

II. BACKGROUND

A. Mathematical Intuition

Professor Gilbert Strang's lecture on the SVD provides an intuition to deal with the problem of working with the SVD. He answers two big questions: What is it, how do we find it.

SVD aims to find orthogonal bases for the input and output spaces for any matrix $\mathbf{A}$, such that when it acts on any vector in the input basis, it produces a scaled vector in the output basis whose scaling factor is called a singular value.

For finding the SVD, Prof. Strang uses the square symmetric matrices $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}\mathbf{A}^\top$. By writing SVD equation for $\mathbf{A}^\top\mathbf{A}$, we find the equation to result in the eigendecomposition of $\mathbf{A}^\top\mathbf{A}$, giving us $\mathbf{V}$ as its eigenvectors, and the singular values as the square roots of its eigenvalues. Similarly, we can find $\mathbf{U}$ using $\mathbf{A}\mathbf{A}^\top$.

Then, Prof. Strang shows why he calls SVD the final and best decomposition: it represents all 4 fundamental subspaces of the matrix. Finally he talks about how $\mathbf{A}$ can also be represented as a sum instead of a product: it is the sum of the outer products of the left and right simgular vectors, multiplied by the singular value. This representation is especially useful in data science, for image compression, noise reduction, recommendation systems etc.

B. Theory

The **Singular Value Decomposition (SVD)** of a matrix is a factorisation of any matrix into the product of an orthogonal, diagonal and orthogonal matrix.

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

Here,

- $\mathbf{U}$ (**Left Singular Vectors**): An orthogonal $m \times m$ matrix
- $\mathbf{\Sigma}$ (**Singular Values**): A diagonal $m \times n$ matrix. It contains the singular values along its diagonal, in decreasing order. The larger values are more important, i.e., represent more information about the overall matrix.
- $\mathbf{V}$ (**Right Singular Vectors**): An orthogonal $n \times n$ matrix

This factorisation can also be interpreted as a transformation, which consists of a rotation, followed by a rescaling, and then finally another rotation.

C. Application

An image is basically a grid of pixels. Thus, a grayscale image can be represented by a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, where each element corresponds to the intensity of a pixel. Similarly for colour images, which can be broken into 3 matrices for Red, Green, and Blue intensities. A low-rank approximation can be obtained by keeping only the top k singular values. Thus, we obtain 3 new matrices:

- $\mathbf{U}_k$: The first k columns of $\mathbf{U}$.
- $\mathbf{\Sigma}_k$: k largest singular values
- $\mathbf{V}_k^\top$: First k rows of $\mathbf{V}^\top$

So the reconstituted matrix is

$$\mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top \tag{2}$$

Even with very small k, good image can be obtained because the top singular values store most of the data.

III. IMPLEMENTATION

A. Algorithm

This project implements an algorithm of the author's own implemenation. The algorithm is inspired from Power Iteration and the Restarted Lanczos Bidiagonalisation. The trade-offs made using this algorithm and possible improvements can be found in section IV-C.

1) Mathematical Logic:

Given: Matrix A (size $m \times n$), rank k

For $r = 0$ to $k - 1$:

$$\mathbf{v}^{(0)} \leftarrow \text{random } n \times 1 \text{ vector}$$

$$\mathbf{v}^{(0)} \leftarrow \frac{\mathbf{v}^{(0)}}{||\mathbf{v}^{(0)}||_2}$$

(for $i = 0, 1, \ldots,$ max_iters):

$$\mathbf{v}_{\text{old}} \leftarrow \mathbf{v}^{(i)}$$

$$\hat{\mathbf{u}} \leftarrow A\mathbf{v}^{(i)}$$

$$\hat{\mathbf{u}}_{\text{ortho}} \leftarrow \hat{\mathbf{u}} - \sum_{j=0}^{r-1} (\hat{\mathbf{u}}^T \mathbf{u}_j)\mathbf{u}_j$$

$$\mathbf{u}^{(i+1)} \leftarrow \frac{\hat{\mathbf{u}}_{\text{ortho}}}{||\hat{\mathbf{u}}_{\text{ortho}}||_2}$$

$$\hat{\mathbf{v}} \leftarrow A^T \mathbf{u}^{(i+1)}$$

$$\hat{\mathbf{v}}_{\text{ortho}} \leftarrow \hat{\mathbf{v}} - \sum_{j=0}^{r-1} (\hat{\mathbf{v}}^T \mathbf{v}_j)\mathbf{v}_j$$

$$\sigma_r \leftarrow ||\hat{\mathbf{v}}_{\text{ortho}}||_2$$

$$\mathbf{v}^{(i+1)} \leftarrow \frac{\hat{\mathbf{v}}_{\text{ortho}}}{\sigma_r}$$

If $(\mathbf{v}^{(i+1)})^T \mathbf{v}_{\text{old}} \approx 1$, break iteration (converged).

Store final components:

$\mathbf{u}_r \leftarrow \mathbf{u}^{(i+1)}$ (Store as column r of U_k)

$\mathbf{v}_r \leftarrow \mathbf{v}^{(i+1)}$ (Store as column r of V_k)

$(\Sigma_k)_{r,r} \leftarrow \sigma_r$

Result: $A_k = U_k \Sigma_k V_k^T = \sum_{r=0}^{k-1} \sigma_r \mathbf{u}_r \mathbf{v}_r^T$

2) Pseudocode Implementation:

```
let U be an m x k matrix (for left vectors)
let V be an n x k matrix (for right vectors)

for r from 0 to k−1:
    v = random normalised n−dimensional vector

    for i from 0 to some_limit:
```

```
        u = A * v

        // Orthogonalize u against previous U vectors (cleaning components in that direction)
        for j from 0 to r-1:
            u_prev = column j of U
            projection = dot(u, u_prev)
            u = u - projection * u_prev

        u = norm(u)

        w = A^T * u

        // again clean w
        for j from 0 to r-1:
            v_prev = column j of V
            projection = dot(w, v_prev)
            w = w - projection * v_prev

        sigma = norm(w)
        if sigma is near-zero:
            break // This component is negligible
        v = w / sigma

        // Check for convergence
        if dot(v, v_prev) is close to 1:
            break

    set column r of U = u
    set column r of V = v

return U, V
```

B. Code

The implemenation is done in pure C. Image handling is done using stbimage.h. Other than that, only standard C libraries are used. The project also uses the standard OpenMP libary to implement intelligent multi-threaded processing of the loops, which highly speeds up the SVD process.

IV. RESULTS

A. *Example Outputs*

For each of the given input images, the following images were reconstructed:

Fig. 0: einstein.jpg: Original, followed by reconstructions for ranks 5, 10, 25, 50, 100 and 200

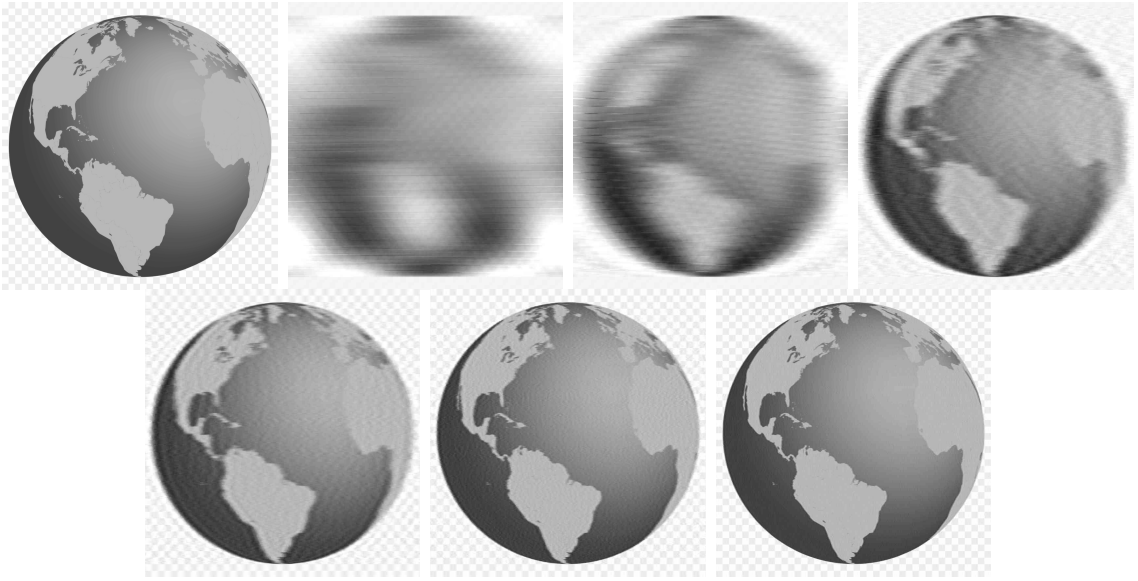

Fig. 0: globe.jpg: Original, followed by reconstructions for ranks 5, 10, 25, 50, 100 and 200

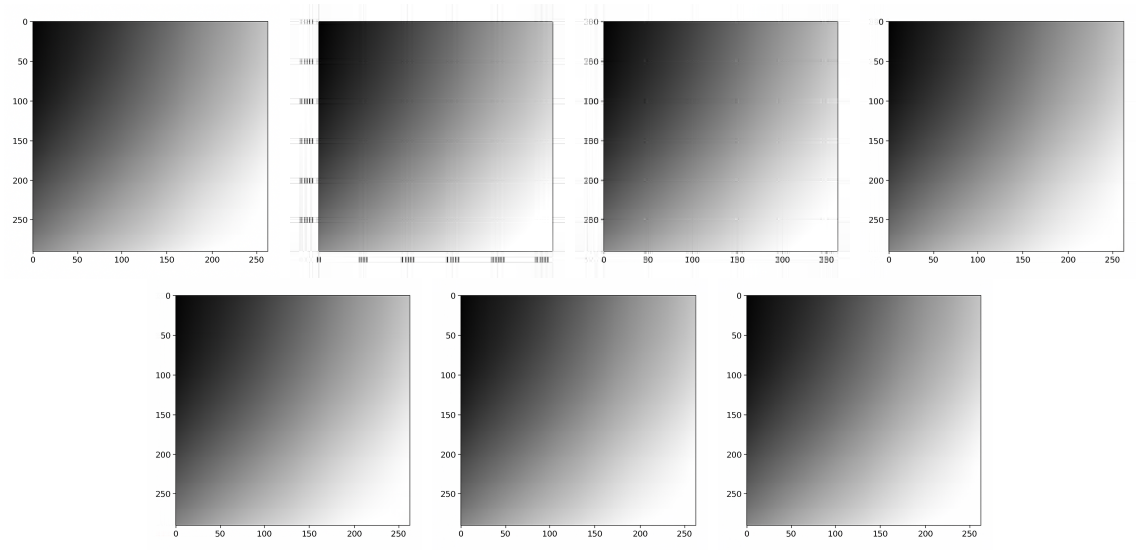

Fig. 0: greyscale.png: Original, followed by reconstructions for ranks 5, 10, 25, 50, 100 and 200

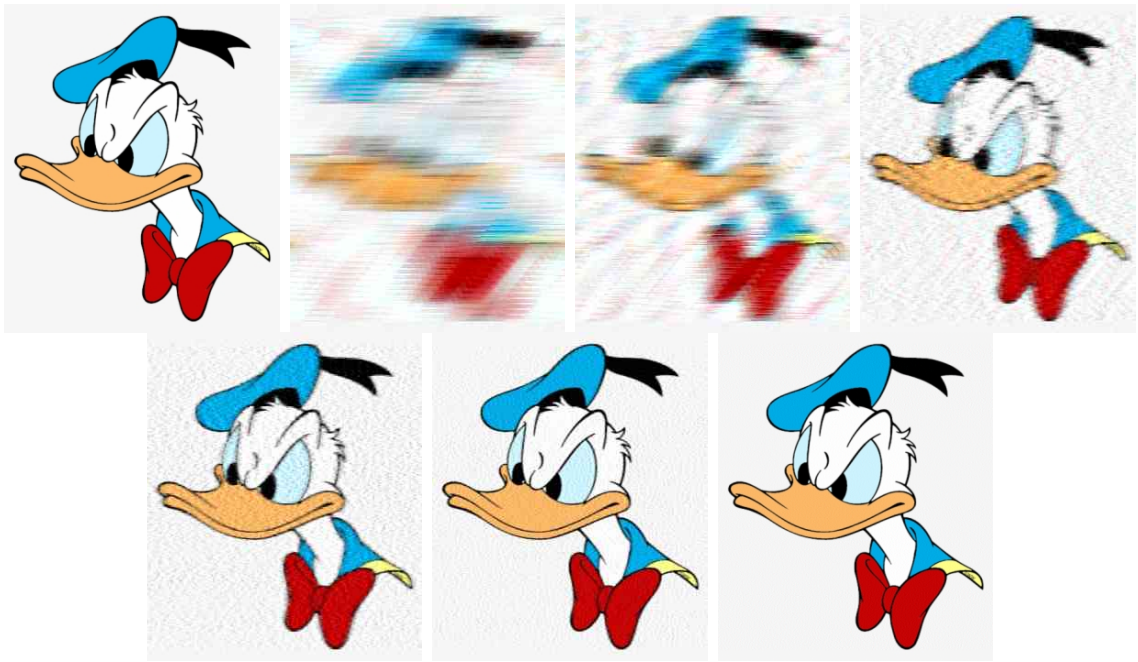

Fig. 0: Colour example: Original, followed by reconstructions for ranks 5, 10, 25, 50, 100 and 200

B. Error Analysis

Error analysis was performed by applying the Frobenius norm on the difference of the source and approximated matrices:

TABLE 0: Errors for: einstein.jpg, globe.jpg, greyscale.png and colour image

Rank	Frobenius norm
5	7303.538038
10	6021.902101
25	4084.188014
50	2148.156405
100	655.483641
200	0.000000

Rank	Frobenius norm
5	28650.542401
10	21599.781109
25	13738.847451
50	9008.290321
100	5803.348162
200	3121.293407

Rank	Frobenius norm
5	11157.807285
10	7190.420786
25	2986.154345
50	1208.301906
100	655.032286
200	618.148769

Rank	Frobenius norm
5	16112.313118
10	13018.889120
25	9057.415979
50	6092.687019
100	3005.471591
200	601.275092

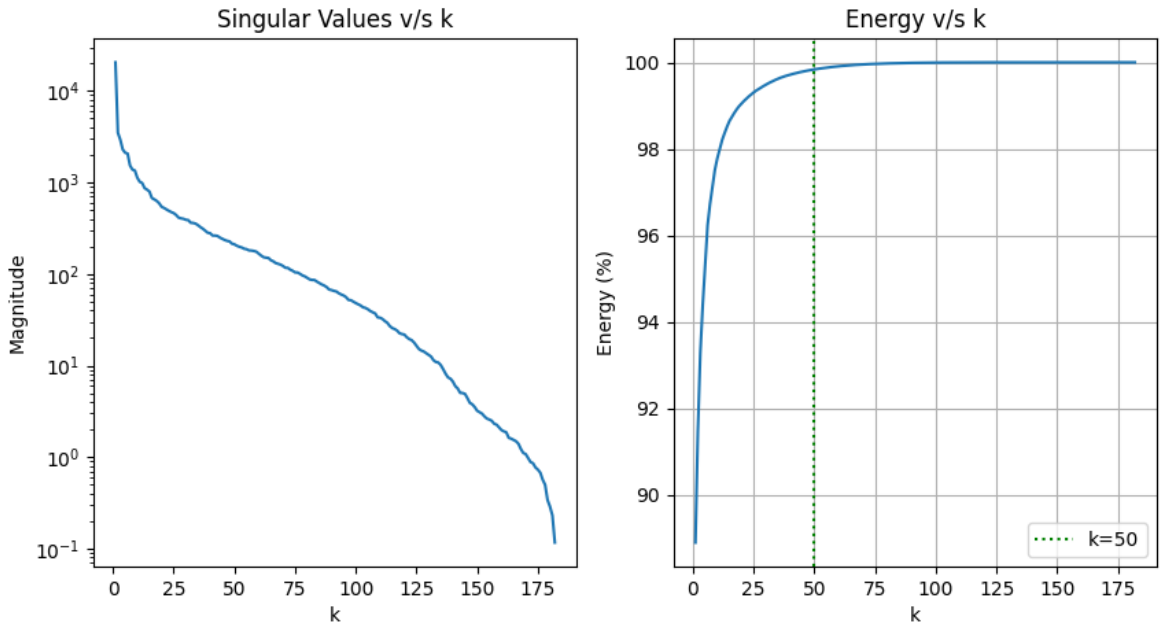

Fig. 0: Visualisation of drop in significance of singular values versus k

C. *Reflections*

The project implements the following **bonus features**:

1) A Full C implementation for the entire core program code. It uses `stbimage.h` to interface with the images directly.
2) Highly efficient colour image extension: Instead of running the expensive SVD algorithm on the 3 colour channels independently, the program runs SVD once on the grayscale data and then uses the similarity in image structure to project the truncated matrix onto each of the 3 colour channels.
3) Comparison of runtime and accuracy with `numpy.linalg.svd`. Runtime for the standard library remained under a second, while error in each case was 2 to 4 times lesser than that for the self-implemented algorithm. (Example: for `einstein.jpg` at $k = 50$, runtime is 0.45 s, and error is 880.35115, compared to our algorithm giving 2148.25802 error in 3.184 s.)
4) Visualisation of singular values across ranks in two ways. First, a logarithmic graph of singular values v/s k (since linear drops far too fast to draw any conclusions). Second, a plot of how image data/energy is stored across k. (Energy is proprtional to sum of squares of singular values, as per 1) Both these graphs show that around $k = 50$ there is sudden drop in magnitude of singular values, as well as a decreasing return on image data. Thus, $k = 50$ is an appropriate balancing point, which is also proven experimentally as given in point 2 in the next section.
5) Self-developed algorithm that uses Power Iteration to find only the top k singular values instead of an inefficient full SVD, and also uses reorthogonalisation inspired from the Restarted Lanczos Bidiagonalisation algorithm to maintain numerical stability in floating-point arithmetic.

6) High performance parallel processing is used on all the computationally expensive tasks, leading to massive increase in speed on multi-threaded hardware.
7) Flexible image I/O that is format independent.
8) Clean, user configurable command line interface.

Variation of output with k:

It is noted that:

1) For PNG files, the relation between k and error seems to be almost an inverse proportion, while for JPEG files the decrease in error with increase in rank is more gradual.
2) In terms of visual quality, there is little to no discernible increase in quality after $k = 50$ onwards. Thus, it shows that the importance of the singular values after the 50th one drops rapidly.
3) An interesting correlation was found between rank and image file size for different file formats. For PNG files, there was a steady increase in file size as rank increased, which is as expected.
4) However, in the case of JPG files, there is actually a drop in the file size after a certain rank. This seems contrary to expectation. One hypothesis is that the inbuilt JPG compression algorithm is far better than the one implemented here, thus for higher ranks, and thus better conditioned data, it is able to achieve higher quality compression. However, without further study into the JPEG compression algorithm, it is not possible to conclude for sure.

Areas for further improvement

1) The parallelism could be implemented better. Right now, all it does is split the loop iterations onto separate threads. However it would probably help the speed if a manual algorithm was used, prioritising the first few values at a time such that the later calculations could then be carried out in advance, such that the initial iteration remains the only bottleneck in the program.
2) There is an inefficiency introduced by having to write to an intermediate .ppm file in the middle of the program. While the output .ppm file could be eliminated successfully, eliminating the first one would require greater understanding of the `stbimage.h` libary and its return data.
3) The biggest question is the upper limit of the main iteration loop in the svd algorithm. It will require testing against a much larger dataset to find an appropriate value that does not waste compute time, while also not incorrectly calculating an imprecise singular value.

V. Conclusions

This project offered an immense learning opportunity in various number of fields. It involved concepts starting from basic matrix identities to the "final and best" factorisation of a matrix, covering many major aspects. It also gave a good idea into how to build a codebase with best practices such as modularity. But the biggest lesson learnt from this project was in optimisation. Since the SVD is generally a very expensive computation to implement, the main challenge lay in tweaking the parameters. Even small things such as changing the order of nested loops brought up to 10 percent increase in speed due to being able to access the continuous memory locations when looping through a matrix. This and more innumerable optimisations made this, overall, a very rich learning experience.

VI. References

1) Introduction to Linear Algebra (Ed 5) (Gilbert Strang)
2) Linear Algebra for Everyone (The Gilbert Strang Series) (Gilbert Strang)
3) Gilbert Strang SVD Lecture
4) Extended Lanczos bidiagonalization algorithm for low rank approximation and its applications (Xuansheng Wang, Francois Glineur, Linzhang Lu, Paul Van Dooren)