Software Assignment - Image Compression using truncated SVD

Aryansingh Sonaye

7 Nov 2025

Summary of Gilbert Strang's Video

In the lecture, I learned that any $m \times n$ matrix $\left(A\right)$ can be factored using the Singular Value Decomposition (SVD) as:

$$\left(A\right) = \left(U\right)\ \Sigma\ \left(V\right)^{\top}, \tag{1}$$

where Σ is a diagonal matrix containing the non-negative singular values

$$\sigma_1 \geq \sigma_2 \geq \cdots, \tag{2}$$

and $\left(U\right)$ and $\left(V\right)$ are orthogonal matrices containing the left and right singular vectors respectively.

One key idea from the video is how SVD explains the relationship between the **row space** and **column space** of $\left(A\right)$. The vectors $\mathbf{v}_i$ (columns of $\left(V\right)$) describe important directions in the row space, while the vectors $\mathbf{u}_i$ (columns of $\left(U\right)$) describe the corresponding directions in the column space.

When $\left(A\right)$ acts on a right singular vector $\mathbf{v}_i$, it **stretches** it by σ_i and maps it to the corresponding left singular vector $\mathbf{u}_i$:

$$\left(A\right)\ \mathbf{v}_i = \sigma_i \mathbf{u}_i. \tag{3}$$

The video also connects SVD with eigenvalue problems. Since $\left(A\right)^{\top}\left(A\right)$ and $\left(A\right)\left(A\right)^{\top}$ are symmetric matrices, they admit orthogonal eigenvectors:

$$\left(A\right)^{\top}\left(A\right)\ \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i, \tag{4}$$

$$\left(A\right)\left(A\right)^{\top}\ \mathbf{u}_i = \sigma_i^2 \mathbf{u}_i. \tag{5}$$

Thus, $\mathbf{v}_i$ are eigenvectors of $\left(A\right)^{\top}\left(A\right)$, and $\mathbf{u}_i$ are eigenvectors of $\left(A\right)\left(A\right)^{\top}$, with the singular values σ_i being the square roots of the corresponding eigenvalues.

Overall, the lecture shows why SVD is useful: it applies to any real matrix, provides orthogonal bases, identifies dominant structural directions in data, and forms the basis for dimensionality reduction and compression. The image compression method used in this assignment is directly based on this idea, where we reconstruct the image using only the top k singular values.

Mathematical Idea

We treat a greyscale image as a matrix :

$$\big(A\big) \in \mathbb{R}^{m \times n}, \tag{6}$$

We want the truncated SVD,

$$\big(A\big)_k = \sum_{i=1}^{k} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top, \tag{7}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots$ are the singular values, $\mathbf{u}_i \in \mathbb{R}^m$ are left singular vectors, and $\mathbf{v}_i \in \mathbb{R}^n$ are right singular vectors.

Instead of computing a full SVD, we iteratively recover the top k triplets $(\sigma_i, \mathbf{u}_i, \mathbf{v}_i)$ using power iteration and deflation.

Power Iteration

Start with a unit vector $\mathbf{v}$ and iterate:

$$\mathbf{u} \leftarrow \frac{\big(A\big)\,\mathbf{v}}{\| \big(A\big)\,\mathbf{v}\|}, \tag{8}$$

$$\mathbf{v} \leftarrow \frac{\big(A\big)^\top \mathbf{u}}{\| \big(A\big)^\top \mathbf{u}\|}, \tag{9}$$

$$\sigma \approx \| \big(A\big)\,\mathbf{v}\|. \tag{10}$$

Repeat until $\mathbf{v}$ converges.

Deflation

After finding $(\sigma, \mathbf{u}, \mathbf{v})$,

$$\big(A\big) \leftarrow \big(A\big) - \sigma \, \mathbf{u}\mathbf{v}^\top. \tag{11}$$

This removes the rank-1 contribution so the next iteration finds the next singular component.

Reconstruction

$$\big(A\big)_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top. \tag{12}$$

Frobenius Error

$$\| (A) - (A)_k \|_F, \qquad \text{Relative Error} = \frac{\| (A) - (A)_k \|_F}{\| (A) \|_F}. \tag{13}$$

Compression Ratio

$$\text{ratio} = \frac{mn}{k(m+n+1)}. \tag{14}$$

Pseudocode

```
READ_AND_NORMALIZE_IMAGE(filename,w,h):
    data, (w, h) ← stbi_load(filename, channels=1)
    A ← array of size w*h
    for each pixel i:
        A[i] = data[i] / 255.0
    return (A)

WRITE_IMAGE_JPG(filename, img, w, h):
    convert img values back in [0,255]
    stbi_write_jpg(...)

DOT(n, x, y) = sum over i of x[i]*y[i]
NORM(n, x) = sqrt(DOT(x,x))

MATVEC: y = A*x
MATVEC_T: y = A^T*x

RANDOM_UNIT_VECTOR(n, v):
    fill v with random values in [-1,1]
    normalize v

POWER_ITERATION(A,m,n,u,v,max,err):
    RANDOM_UNIT_VECTOR(v)
    repeat max times:
        u = A*v ; normalize u
        v = A^T*u ; normalize v
        sigma = ||A*v||
        stop if v converges
    return sigma
```

```
ADD_RANK1(A, m, n, sigma, u, v):
    A = A + sigma * u * v^T

TOP_K_SVD(A, m, n, k, Sigma, U, V, max, err):
    for i = 1..k:
        sigma, u, v = POWER_ITERATION
        store u, v, sigma
        A = A - sigma*u*v^T   // deflation

RECONSTRUCT(Ak, k, Sigma, U, V):
    Ak = sum_{i=1..k} sigma_i * u_i * v_i^T

MAIN:
    load input image -> A
    save A copy as A0
    run TOP_K_SVD(A)
    reconstruct Ak
    compute errors and compression ratio
    write Ak to JPG
```

Convergence of the Power Method (Proof)

Let $(A) \in \mathbb{R}^{n\times n}$ be diagonalizable with eigenpairs

$$(A)\,\mathbf{x}_i = \lambda_i \mathbf{x}_i, \qquad i = 1, \ldots, n, \tag{15}$$

and let the eigenvalues be ordered by magnitude

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|. \tag{16}$$

Suppose the initial guess $\mathbf{x}_0$ can be written as

$$\mathbf{x}_0 = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n, \qquad c_1 \neq 0. \tag{17}$$

Then the iterates $(A)^k\,\mathbf{x}_0$ approach a nonzero multiple of the dominant eigenvector $\mathbf{x}_1$. Equivalently,

$$\frac{(A)^k\,\mathbf{x}_0}{\|\,(A)^k\,\mathbf{x}_0\|} \;\longrightarrow\; \pm\frac{\mathbf{x}_1}{\|\mathbf{x}_1\|} \qquad \text{as } k \to \infty. \tag{18}$$

Since the eigenvectors form a basis, write

$$\mathbf{x}_0 = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n. \tag{19}$$

Multiplying repeatedly by (A) gives

$$(A)^k \mathbf{x}_0 = c_1\lambda_1^k\mathbf{x}_1 + c_2\lambda_2^k\mathbf{x}_2 + \cdots + c_n\lambda_n^k\mathbf{x}_n \tag{20}$$

$$= \lambda_1^k \left(c_1\mathbf{x}_1 + c_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k \mathbf{x}_2 + \cdots + c_n \left(\frac{\lambda_n}{\lambda_1}\right)^k \mathbf{x}_n \right). \tag{21}$$

Since $|\lambda_i/\lambda_1| < 1$ for $i \geq 2$,

$$\left(\frac{\lambda_i}{\lambda_1}\right)^k \longrightarrow 0 \qquad \text{as } k \to \infty. \tag{22}$$

Thus,

$$(A)^k \mathbf{x}_0 \longrightarrow \lambda_1^k c_1 \mathbf{x}_1. \tag{23}$$

Normalizing both sides gives

$$\frac{(A)^k \mathbf{x}_0}{\| (A)^k \mathbf{x}_0 \|} \longrightarrow \pm \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|}, \tag{24}$$

which completes the proof.

Comparison of Different SVD Algorithms

1. Golub–Kahan Bidiagonalization + QR (Standard SVD)

How it works: The matrix A is first reduced to bidiagonal form using Householder transformations. Then, the QR algorithm is applied to extract singular values and singular vectors.

- **Accuracy:** Very high
- **Speed:** Fast when using optimized BLAS/LAPACK libraries
- **Memory Use:** Moderate
- **Difficulty:** Hard to implement from scratch
- **Advantages:**
  - Most accurate and numerically stable method
  - Works for any shape matrix
- **Disadvantages:**
  - Code is long and complex
  - Usually depends on external numerical libraries

2. Jacobi SVD

How it works: Uses a sequence of orthogonal plane rotations to gradually diagonalize the matrix while maintaining orthogonality.

- **Accuracy:** Very high
- **Speed:** Slow for large matrices
- **Memory Use:** High
- **Difficulty:** Medium
- **Advantages:**
  - Conceptually simple
  - Very numerically stable
- **Disadvantages:**
  - Extremely slow for large images
  - Not suitable for practical compression here

3. Randomized SVD

How it works: Multiplies A with a random matrix to estimate the dominant subspace, then performs SVD on a much smaller projected matrix.

- **Accuracy:** High (approximate)
- **Speed:** Very fast for large datasets
- **Memory Use:** Low/Medium
- **Difficulty:** Medium
- **Advantages:**
  - Scales well to large matrices
  - Low memory footprint
  - Efficient in machine learning applications
- **Disadvantages:**
  - Slight loss of accuracy
  - Output can vary slightly due to randomness

4. Power Iteration + Deflation (Method Used in This Project)

How it works: Power iteration is used to find the largest singular value σ_1 and associated vectors u_1, v_1. Then the matrix is deflated by subtracting $\sigma_1 \mathbf{u_1} \mathbf{v_1}^\top$. The process is repeated to compute the top-k singular components.

- **Accuracy:** Good for top singular values
- **Speed:** Slower when k is large (computes one component at a time)
- **Memory Use:** Very low
- **Difficulty:** Easy
- **Advantages:**
  - Easy to understand and code
  - No external numerical libraries required
  - Illustrates SVD structure directly
  - Very memory efficient
- **Disadvantages:**
  - Converges slowly if singular values are close
  - Deflation introduces small numerical errors

Why This Algorithm Was Chosen

I chose the Power Iteration with Deflation algorithm because it is straightforward to implement in C and does not require external numerical libraries.

$$A^T A\,\mathbf{v} = \sigma^2 \mathbf{v}, \quad A\mathbf{v} = \sigma \mathbf{u}, \tag{25}$$

It clearly demonstrates how singular vectors and singular values arise. This method computes only the top k singular components rather than performing a full SVD, which matches the goal of image compression, where only the dominant structure of the image is needed. It is also memory efficient since it updates the matrix in place during deflation.

Comparison with NumPy's `svd`

To check how our SVD method compares with a standard implementation, we looked at `numpy.linalg.svd`. NumPy uses highly optimized code, so it is usually very fast and very accurate.

Our method uses power iteration and deflation to get only the top k singular values. This means we are not computing the full SVD, only the important parts needed for compression.

Accuracy: For larger k (for example $k = 50$ or $k = 100$), our reconstructed images look almost the same as the original. For small k (like $k = 2$ or $k = 5$), the result is blurry, while NumPy still captures more detail since it does the full SVD.

Speed: NumPy is faster overall because it is written in optimized C/Fortran. Our method is slower when k is large because each additional singular value needs more power iteration steps. But when k is small, our method is still reasonably quick.

Memory: Our method uses less memory because we only store k singular vectors, not the full U and V matrices. This is useful when working with very large images.

In summary: NumPy's SVD is the best choice when you want full accuracy and high speed. Our method is simpler to understand and is a good choice when we only need the top k components for image compression.