Software Project: Image Compression using SVD

ee25btech11009 – Anshu kumar ram

Contents

1 Overview and SVD basics

Summary of Strang's lecture.
Any real matrix A admits a factorization

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

where $U^\top U = I$, $V^\top V = I$, and Σ is diagonal with nonnegative entries:

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \end{pmatrix}, \qquad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0.$$

The σ_i are the singular values of A.

Conceptually:

- The columns of V provide an orthonormal basis for the row space (right singular vectors).
- The columns of U provide an orthonormal basis for the column space (left singular vectors).
- Each right singular vector v_i is mapped by A to $\sigma_i u_i$:

$$Av_i = \sigma_i u_i.$$

To compute these components one inspects

$$A^\top A = V\Sigma^\top \Sigma V^\top, \tag{2}$$

$$AA^\top = U\Sigma\Sigma^\top U^\top, \tag{3}$$

so the eigenvectors of $A^\top A$ are the columns of V with eigenvalues σ_i^2, and similarly for U.

A special case occurs when A is symmetric positive definite: then $U = V$ and the SVD reduces to the eigendecomposition $A = Q\Lambda Q^\top$ with positive eigenvalues.

2 SVD for image compression

A grayscale image of size $m \times n$ can be encoded as a matrix $A \in \mathbb{R}^{m\times n}$, where each entry represents the pixel intensity (typically in $[0, 255]$). The aim of truncated SVD compression is to approximate A by a lower-rank matrix A_k that captures most of the visually-important structure:

$$A \approx A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\top. \tag{4}$$

Using only the top k singular triplets reduces storage while preserving the principal image features.

2.1 Mathematical formulation

Let $r = \operatorname{rank}(A)$. The full SVD is $A = U\Sigma V^\top$. The optimal (in Frobenius norm) rank-k approximation is

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\top. \tag{5}$$

This minimizes $\|A - A_k\|_F$ among all matrices of rank at most k.

3 Implementation : power iteration + deflation

Function: `compute_svd()`

This implementation computes the top-k singular triplets of a matrix A using power iteration applied to the symmetric matrix $A^\top A$. (compute $A^\top A$, extract eigenpairs with power iteration and deflation on $A^\top A$, then form U from A, V, and the singular values).
The main steps are:

- Compute the transpose $A^\top$ and form the symmetric Gram matrix $A^\top A$.
- Allocate storage for the top k eigenvalues and the matrix of right singular vectors $V \in \mathbb{R}^{n\times k}$ (where n is the number of columns of A).
- For $i = 1 \ldots k$ do:
  - Use the Power Iteration method on $A^\top A$ to approximate the i^{th} dominant eigenvalue λ_i and eigenvector v_i.
  - Store v_i as the ith column of V.
  - Deflate the matrix $A^\top A$ by removing the contribution of the found eigenpair so the next iteration recovers the next eigenvector.
- Convert eigenvalues to singular values via $\sigma_i = \sqrt{|\lambda_i|}$. Taking absolute value is a defensive choice against tiny negative round-off errors.
- Compute the left singular vectors U from A, V and Σ using

  $$U = AV\Sigma^{-1},$$

  i.e. compute each column $u_i = \frac{Av_i}{\sigma_i}$. Columns corresponding to $\sigma_i \approx 0$ must be handled carefully (skip or orthonormalize).
- Pack the results into an `SVD` structure (or equivalent) containing U, σ, V, and k.
- Free temporary matrices and arrays (e.g., $A^\top$, $A^\top A$, intermediate eigenvalue arrays) to avoid memory leaks.

These steps match the provided implementation where power iteration and deflation are performed on $A^\top A$, and U is computed afterward from A and the computed V, Σ.

3.1 Pseudocode (implementation-accurate)

```
Input: image file -> matrix A (m x n)
Compute AT = transpose(A)
Compute ATA = AT * A   # symmetric n x n matrix
Allocate arrays: eigenvalues[1..k], V[n x k]
for i = 1..k:
  v = random nonzero vector in R^n
  normalize v
  repeat for max_iteration:
    v = ATA * v         # power iteration on ATA
    normalize v
  end repeat
  lambda = v^T * (ATA * v)   # Rayleigh quotient (approx eigenvalue)
  store lambda in eigenvalues[i]
```

```
  store v as column i of V
  if i < k:
    ATA = ATA - lambda * (v * v^T)   # deflation on ATA
end for
sigma[i] = sqrt(abs(eigenvalues[i])) for i=1..k
Compute U: for i=1..k: u_i = (A * v_i) / sigma[i]
Package SVD = {U, sigma, V, k}
Free temporary memory

Write compressed representation using top-k triplets
```

4 Mathematical concept for Power Iteration

Let M be a matrix (here $M = A^\top A$). Power iteration finds the eigenvector corresponding to the largest-in-magnitude eigenvalue of M. Starting from an initial vector v_0 with a nonzero component along the dominant eigenvector, repeated multiplication by M amplifies the dominant mode.

Assume an eigendecomposition $M = X\Lambda X^{-1}$ with $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. Writing

$$v_0 = \sum_j c_j x_j,$$

we have

$$v_k = M^k v_0 = \sum_j c_j \lambda_j^k x_j.$$

Dividing by $\|v_k\|$ isolates the x_1 direction as k grows. Normalization each step prevents overflow and stabilizes convergence.

5 Error metric

We measure error by the Frobenius norm:

$$E = \|A - A_k\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij} - (A_k)_{ij})^2}. \tag{6}$$

6 Why power iteration was chosen

The implementation favors power iteration because:

- It is straightforward to implement: only repeated matrix-vector multiplications and scalings are needed.
- It has a low memory footprint since it stores only a few vectors rather than full orthogonal bases.
- It is efficient when one needs only the top k singular components instead of the entire SVD.

6.1 Why some alternatives were not used

Jacobi method: converges slowly for large matrices, needs multiple sweeps, and scales poorly (roughly $O(n^3)$).

Golub–Kahan bidiagonalization: offers robust convergence but requires more complex orthogonal transformations and storage of basis vectors.

Divide-and-conquer SVD: involves recursive decomposition and is both programmatically and memory intensive for general images.

7 Practical choices: iterations and stability

In the implementation each singular vector is refined using a fixed number of power iterations (e.g., 50). This fixed cap balances runtime and accuracy: for many images convergence occurs in fewer iterations, but the upper limit ensures stability even for ill-conditioned problems.

Normalization at each iteration is essential to prevent overflow/underflow and to stabilize convergence:

$$v \leftarrow \frac{v}{\|v\|}.$$

8 Observations, limitations and trade-offs

- **Convergence:** power iteration converges quickly when the dominant singular value is well separated; convergence slows if leading singular values are close.
- **Numerical effects:** normalization mitigates growth/decay of iterates, but round-off may still accumulate for very large matrices.
- **Perceptual vs numeric trade-off:** beyond a certain k, additional singular components yield diminishing perceptual benefits while increasing storage and time.

8.1 Rank k vs compression

The storage needed to represent A_k using the compact SVD form is

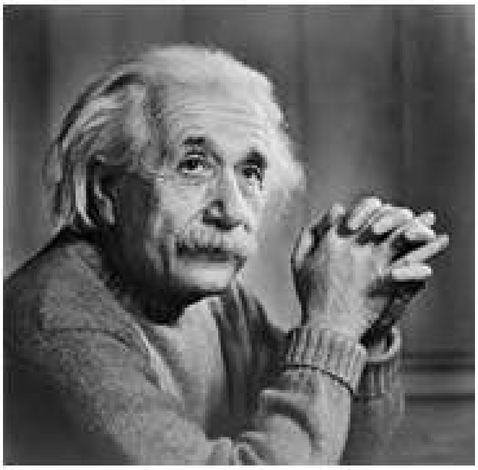
$$N_k = k(m + n + 1),$$

while the original requires $N_{\text{orig}} = mn$ scalars. The compression ratio is therefore

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

9 Results (example tables and images)

Images
(a) Original Einstein Image

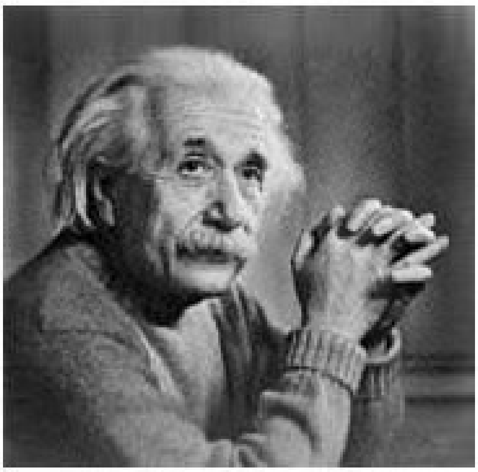


(a) Original Image

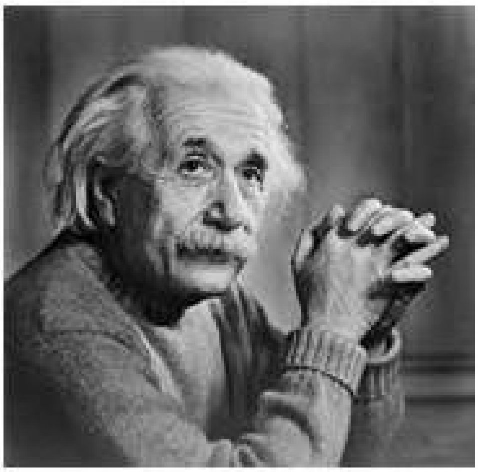
Compressed Images

(b) $k = 5$

(c) $k = 20$



(d) $k = 50$



(e) $k = 100$

Figure 1: SVD-based image compression results for different values of k.

k	**Compression Ratio**	**Runtime (s)**	**Frobenius Error**
5	18.35	0.146	4713.5869
20	4.59	0.520	2126.5615
50	1.83	1.366	880.5452
100	0.92	2.735	165.0320

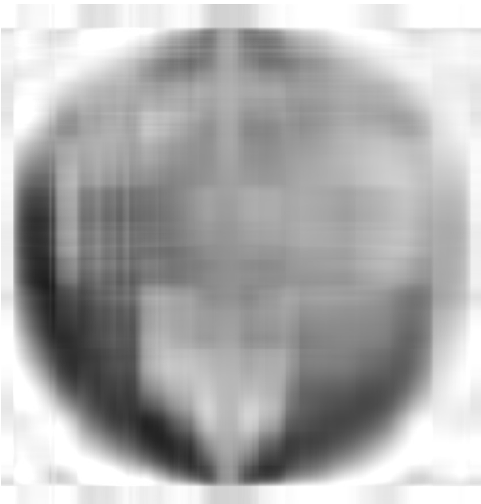
Table 1: SVD-based image compression performance for different values of k of einstein.jpg.

Images

(a) Original Globe Image

(a) Original Image

Compressed Images



(b) $k = 5$

(c) $k = 20$

(d) $k = 50$

(e) $k = 100$

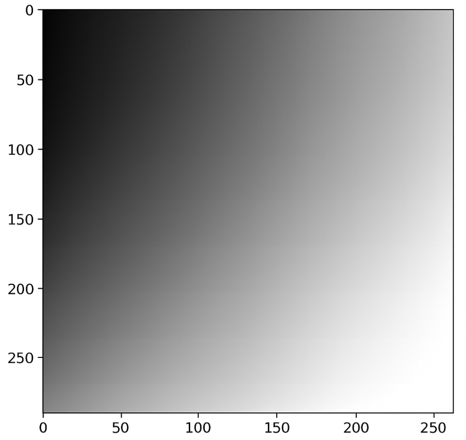
Figure 2: SVD-based image compression results for the Globe image using different values of k.

k	**Compression Ratio**	**Runtime (s)**	**Frobenius Error**
5	85.86	9.158	20704.2746
20	21.46	18.604	10634.4134
50	8.59	36.708	6186.0730
100	4.29	63.059	3673.9582

Table 2: SVD-based image compression performance for the `globe.jpg` image.

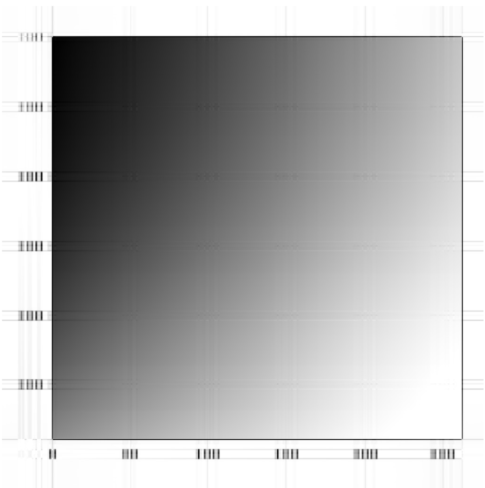
Images

(a) Original Greyscale Image

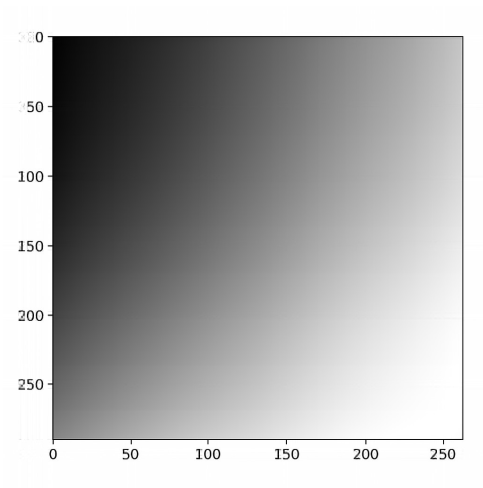


(a) Original Image

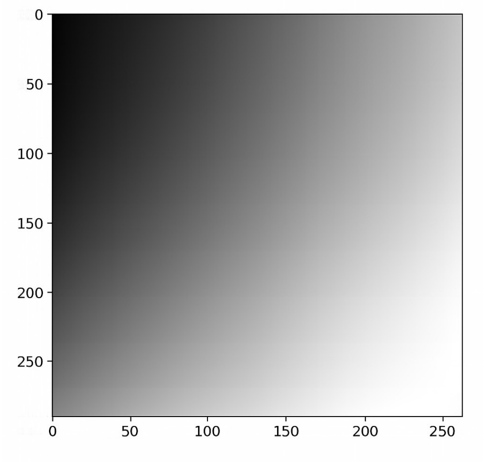
Compressed Images



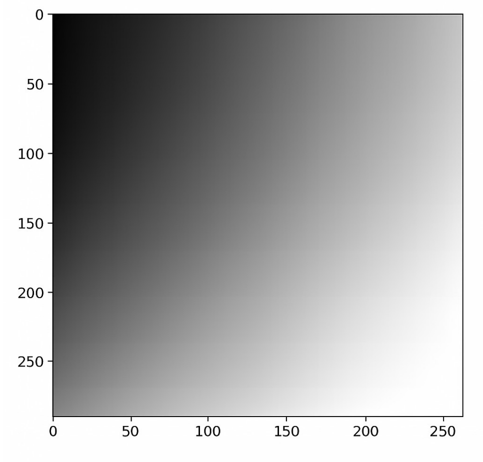
(b) $k = 5$



(c) $k = 20$



(d) $k = 50$



(e) $k = 100$

Figure 3: SVD-based image compression results for the greyscale image using different values of k.

k	**Compression Ratio**	**Runtime (s)**	**Frobenius Error**
5	102.35	17.570	11146.3094
20	25.59	28.696	3808.1952
50	10.24	56.157	1160.1419
100	5.12	100.557	512.4441

Table 3: SVD-based image compression performance for the `greyscale.png` image.

10 Conclusion

Truncated SVD offers an effective way to compress grayscale images: storing the top k singular triplets typically captures the bulk of perceptual detail with far fewer numbers than the full image. Power iteration with deflation is an accessible, memory-light strategy for extracting these dominant components when a full SVD is not necessary.