SOFTWARE PROJECT
REPORT

AI25BTECH11018 - M.HEMANTH REDDY

• Summary of Strang's video on SVD :

This video introduces the Singular Value Decomposition (SVD), a way to factor any matrix, including rectangular ones, into three pieces: U, Σ, and V^T

Here's a breakdown of the SVD:
U is an orthogonal matrix
Σ is a diagonal matrix, containing the singular values
V^T is also an orthogonal matrix.

The video explains how SVD differs from eigenvalue decomposition ($V\lambda V^{-1}$) because it uses two different orthogonal matrices (U and V) and can be applied to rectangular matrices. The singular values in Σ are the square roots of the eigenvalues of A^TA . U's columns are eigenvectors of AA^T, and V's columns are eigenvectors of A^TA . SVD can be seen as a sequence of geometric transformations: a rotation (V^T), a stretching/scaling (Σ), and another rotation (U).

The video discusses an important application of SVD: Principal Component Analysis (PCA). It explains how the largest singular value (Σ_1) and its corresponding singular vectors (U_1 and V_1) represent the most significant information.SVD is crucial for analyzing large rectangular data matrices, identifying principal components, and extracting the most important parts of the data.

• Explanation of the implemented algorithm (math + pseudocode) :

The code implements a method for Truncated Singular Value Decomposition (SVD) based on the Eigen-Decomposition of the Gram Matrix (A^TA), specifically utilizing the Power Iteration Method with Orthogonal Deflation to find the top k components.

This approach is chosen because A^TA is a symmetric matrix, and its eigenvectors directly relate to the right singular vectors of A.
The fundamental connection between SVD and Eigen-Decomposition is:

1. **Right Singular Vectors (V):** The eigenvectors ($\mathbf{v}_j$) of the symmetric matrix $\mathbf{S} = \mathbf{A}^\mathsf{T}\mathbf{A}$ are the Right Singular Vectors of **A**.
2. **Singular Values (Σ):** The eigenvalues (λ_j) of **S** are related to the singular values (σ_j) of **A** by: $\sigma_j = \sqrt{\lambda_j}$.
3. **Left Singular Vectors (U):** Once $\mathbf{v}_j$ and σ_j are known, the Left Singular Vectors are computed directly: $\mathbf{u}_j = \frac{1}{\sigma_j}(\mathbf{A}\mathbf{v}_j)$.

The final goal is to reconstruct the compressed matrix $\mathbf{A}_k$ using the outer product sum:

$$\mathbf{A}_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^\mathsf{T}$$

• Implemented Algorithm: Power Iteration with Deflation
The core of the `truncated svd` function is an iterative loop that finds one eigenvector/ eigenvalue pair at a time and then ”removes” its effect from the matrix **S**.

1. Pre-Computation (Initialization)

1. **Compute S = $\mathbf{A}^\top\mathbf{A}$**: The function `matmul_AtA` explicitly computes the $n \times n$ Gram matrix **S**. (The initial complexity is dominated by this step).
2. **Allocate Storage**: Allocate memory for k eigenvectors **V**, k eigenvalues $\mathbf{\Lambda}$, k singular values $\mathbf{\Sigma}$, and k left singular vectors **U**.

2. Iterative Eigen-Decomposition (The `comp` Loop)

The main loop runs k times to find the top k components.

A. Power Iteration (Inner Loop)

Inside the main loop, Power Iteration is used to find the dominant eigenvector **v** and eigenvalue λ of the current, deflated matrix **S**.

- **Iteration**: $\mathbf{y} = \mathbf{S}\mathbf{b}_{\text{new}}$ (using deflation, explained below).
- **Normalization**: $\mathbf{b}_{\text{new}} = \mathbf{y}/||\mathbf{y}||$. The vector **b** iteratively converges to the true eigenvector $\mathbf{v}_j$.
- **Rayleigh Quotient**: The corresponding eigenvalue is estimated using the Rayleigh Quotient: $\lambda_{\text{new}} = \mathbf{b}_{\text{new}}^\top(\mathbf{S}\mathbf{b}_{\text{new}})$.
- **Convergence**: The iteration stops when λ stabilizes.

B. Orthogonal Deflation

To find the second, third, and subsequent eigenvectors, their effects must be removed from matrix **S** before starting the next power iteration. This is done implicitly inside the sym_mat_vec_minus_projections function using the **deflation** step:

The matrix-vector product $\mathbf{y} = \mathbf{S}\mathbf{x}$ is modified to project **x** away from all already found eigenvectors ($\mathbf{v}_0, \mathbf{v}_1, \ldots$):

$$\mathbf{y} \leftarrow \mathbf{S}\mathbf{x} - \sum_{i=0}^{\text{found}-1} \lambda_i(\mathbf{v}_i^\top\mathbf{x})\mathbf{v}_i$$

3. Final Computation and Reconstruction

After the k eigenvectors (**V**) and eigenvalues ($\mathbf{\Lambda}$) are found:

1. **Singular Values**: Compute $\sigma_j = \sqrt{\lambda_j}$.
2. **Left Singular Vectors**: Compute $\mathbf{u}_j = (\mathbf{A}\mathbf{v}_j)/\sigma_j$.
3. **Reconstruction**: The final image $\mathbf{A}_k$ is built by iterating through the top k components and summing the outer products:

$$\mathbf{A}_{\text{out}} = \sum_{j=0}^{k-1} \sigma_j(\mathbf{u}_j\mathbf{v}_j^\top)$$

Pseudocode

function truncated svd(A, m, n, k):
// 1. Pre-computation
S = matmul AtA(A, m, n) // S = A^TA (n x n)
V = Allocate(n * k) // Storage for right singular vectors
Lambda = Allocate(k) // Storage for eigenvalues

// 2. Iterative Eigen-Decomposition with Deflation
for comp from 0 to k-1:
b = Initialize Random Vector(n) // Current eigenvector estimate
lambda = 0.0

for iter from 0 to 2000:
// Apply S, subtracting projections onto previously found vectors (V,Lambda)
y = sym mat vec minus projections(S, b, V, Lambda, comp)

// Normalize y tmp (New eigenvector estimate)
norm y = norm(y)
tmp = y/norm y

// Rayleigh Quotient Estimate
y deflated = sym mat vec minus projections(S, tmp, V, Lambda, comp)
new lambda = dot vec(tmp, y deflated)

// Check Convergence
if new lambda is close to lambda:
lambda = new lambda
b = tmp // Converged eigenvector
break

lambda = new lambda
b = tmp // Continue iteration

// Store the result
V[comp] = b
Lambda[comp] = lambda

// 3. Final Computation and Reconstruction
sigma = Allocate(k)
for j from 0 to k-1:
sigma[j] = sqrt(max(Lambda[j], 0))
sigma out[j] = sigma[j] // Output singular values

U = Allocate(m * k) // Left singular vectors
for j from 0 to k-1:
// u_j = (A * v_j) / sigma j
u_j = (A * V[j]) / sigma[j]
U[j] = u_j

// 4. Final Reconstruction A k
Ak out = Zero Matrix(m, n)
for j from 0 to k-1:
// A k += sigma j * u_j * u_j^T
Ak out += sigma[j] * outer product(U[j], V[j])

• Comparing different algorithms and explain why did I choose a particular algorithm :
There are several algorithms to compute SVD.The algorithms are :

Jacobi SVD
Golub–Reinsch (QR + bidiagonalization)
Power Iteration (with Orthogonal Deflation)
Randomized SVD
Lanczos Bidiagonalization

• Jacobi SVD: Iteratively diagonalizes AA^T or A^TA using plane rotations until off-diagonal elements become small.

• (QR + bidiagonalization) : Reduces A to bidiagonal form using Householder transformations, then applies QR iterations to find singular values.

• Power Iteration (with Orthogonal Deflation) : Iteratively finds the largest singular value/vector pair, then deflates A to find the next.

• Lanczos Bidiagonalization : Builds orthogonal bases for A and A^T iteratively, forming a smaller bidiagonal matrix from which singular values are extracted .

• Randomized SVD : Projects A onto a lower-dimensional random subspace and computes SVD on this smaller matrix.

Why Power Iteration with Orthogonal Deflation Was Chosen:

1. Simplicity and Implementability
2. Efficiency for Truncated SVD
3. Numerical Stability and Orthogonality
4. Good Trade-off Between Accuracy and Speed

For this project, the Power Iteration with Orthogonal Deflation algorithm was chosen to implement truncated SVD. Compared to classical algorithms like Golub–Reinsch or Jacobi methods, it offers the best balance between computational efficiency, ease of implementation in C, and adequate accuracy for image compression.

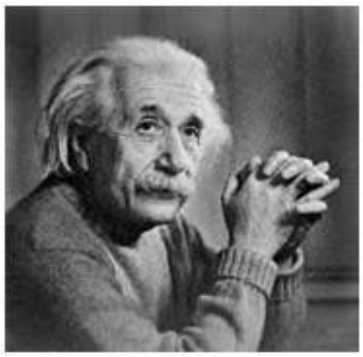
Since only the top k singular values and vectors contribute significantly to visual image quality, this algorithm efficiently captures the most important image features with minimal computation.

• Reconstructed images for different k :

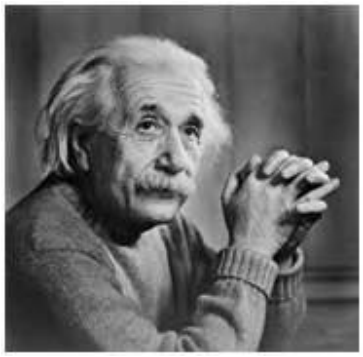
Einstein :
For k = 5

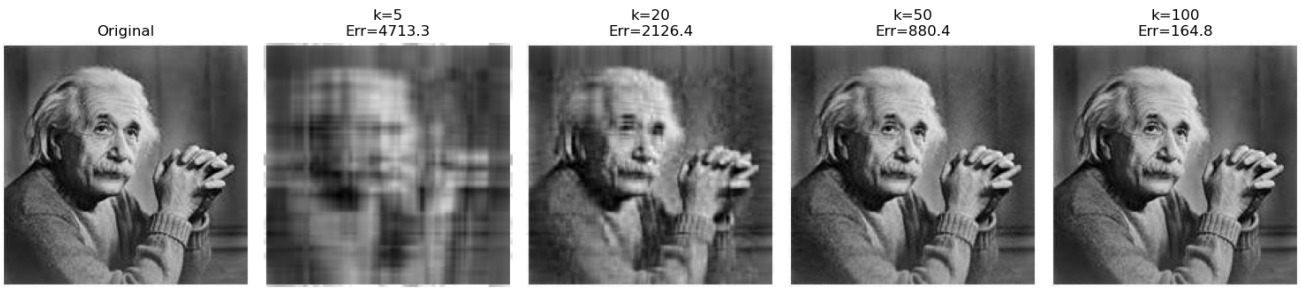
For k = 20

For k = 50



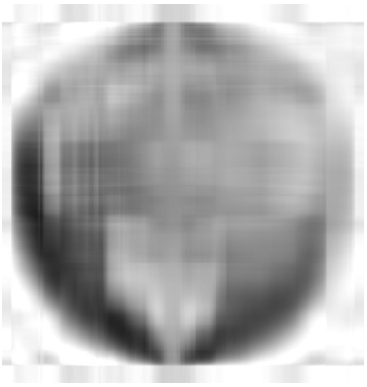
For k = 100





Globe :

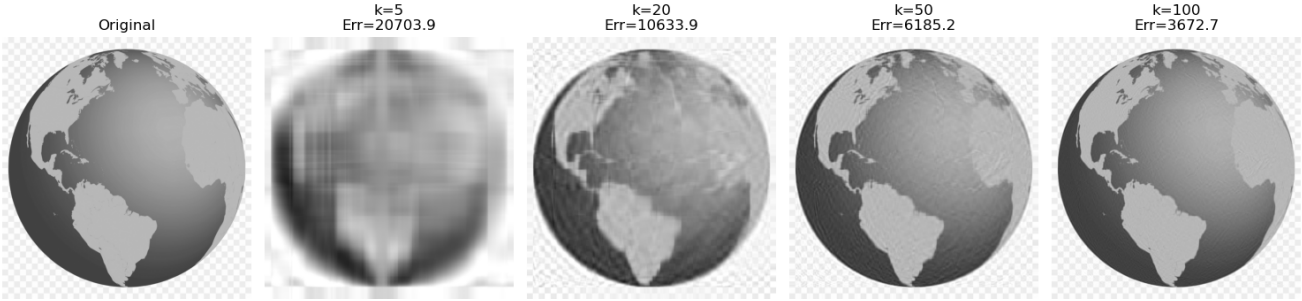
For k = 5



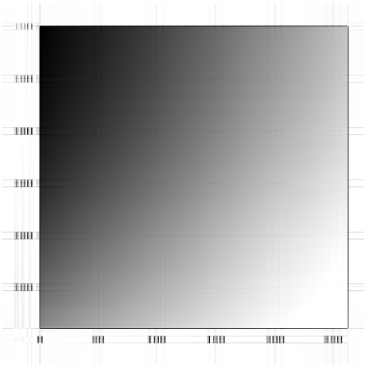
For k = 20

For k = 50

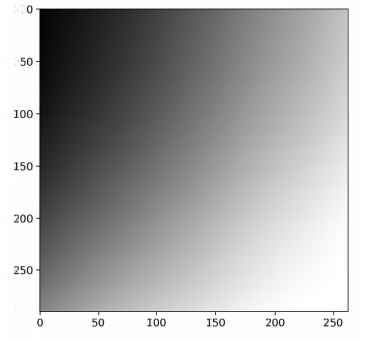
For k = 100



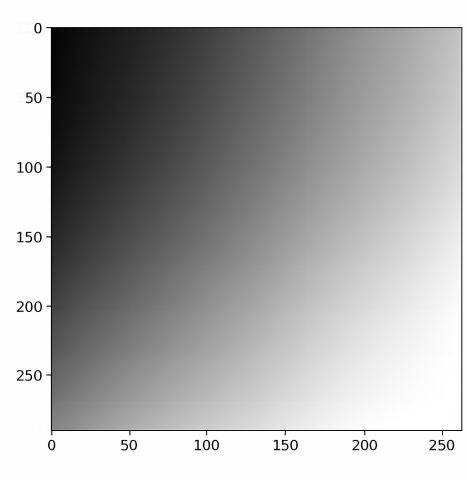
Greyscale :
For k = 5



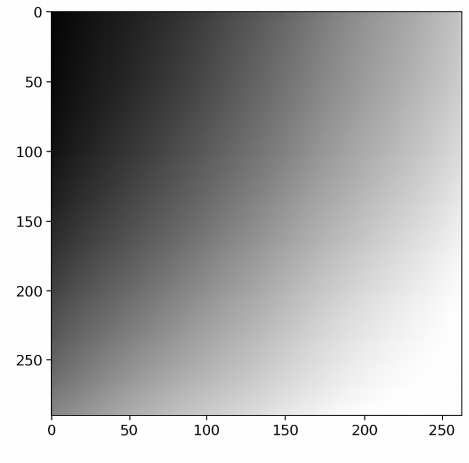
For k = 20

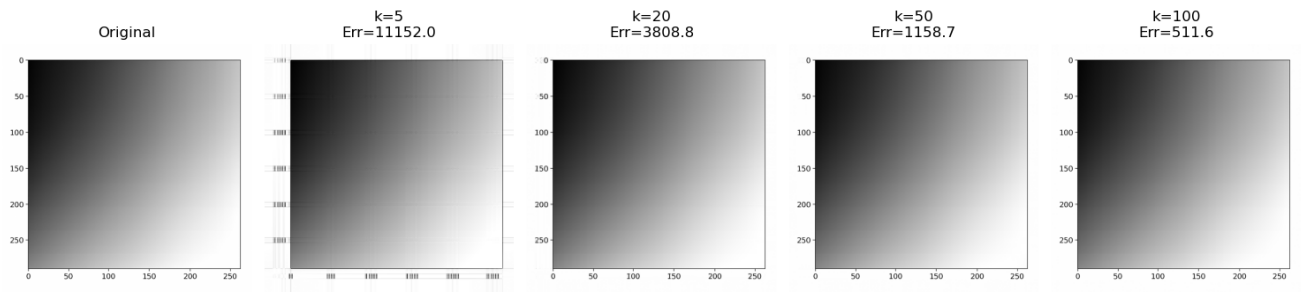


For k = 50



For k = 100





• Error analysis :

The Frobenius norm is widely used in machine learning and data analysis as a measure of error. For example, in matrix approximation problems, minimizing the Frobenius norm of the difference between the original matrix and its approximation ($||A - \hat{A}||_F$) is the standard goal of the optimization.
For globe:
Computing SVD with k = 5
Frobenius error $||A - \hat{A}||_F$ = 20703.8928
Relative error = 0.1308
Computing SVD with k = 20
Frobenius error $||A - \hat{A}||_F$ = 10633.9230
Relative error = 0.0672
Computing SVD with k = 50
Frobenius error $||A - \hat{A}||_F$ = 6185.2136
Relative error = 0.0391
Computing SVD with k = 100
Frobenius error $||A - \hat{A}||_F$ = 3672.6687
Relative error = 0.0232

For einstein:
Computing SVD with k = 5
Frobenius error $||A - \hat{A}||_F$ = 4713.3385
Relative error = 0.2161
Computing SVD with k = 20
Frobenius error $||A - \hat{A}||_F$ = 2126.4391
Relative error = 0.0975
Computing SVD with k = 50
Frobenius error $||A - \hat{A}||_F$ = 880.3512
Relative error = 0.0404
Computing SVD with k = 100
Frobenius error $||A - \hat{A}||_F$ = 164.7819
Relative error =0.0076

For greyscale:
Computing SVD with k = 5
Frobenius error $||A - \hat{A}||_F$ = 11151.9803
Relative error = 0.0575
Computing SVD with k = 20
Frobenius error $||A - \hat{A}||_F$ = 3808.7538
Relative error = 0.0197
Computing SVD with k = 50
Frobenius error $||A - \hat{A}||_F$ = 1158.6843
Relative error = 0.0060
Computing SVD with k = 100
Frobenius error $||A - \hat{A}||_F$ = 511.5542
Relative error = 0.0026

• Discussion of trade-offs and reflections on implementation choice:

This project perfectly demonstrates the fundamental trade-off in SVD-based compression.

The Trade-off :

1. **k (Number of Singular Components):** This is the ”knob” you turn. It represents the number of ”image layers” you are using for the reconstruction.

2. **Image Quality (Error):** Image quality is inversely proportional to the Frobenius norm error ($||\mathbf{A} - \mathbf{A}_k||_F$).
   Low k (e.g., 5, 20): A low k value uses only the few most dominant singular values. These capture the largest, most significant features of the image (e.g., overall brightness, major shapes). The resulting image has very low accuracy, appears ”blocky” and the error $||\mathbf{A} - \mathbf{A}_k||_F$ is high.

   High k (e.g., 50, 100): A high k value adds in more singular values, which correspond to finer details, textures, and subtle shadows. The image quality becomes much higher, and the error $||\mathbf{A} - \mathbf{A}_k||_F$ becomes low. As k approaches the full rank r, the error approaches zero.

3. **Compression (Storage):** The storage required for the original $m \times n$ image is $\mathbf{m} \cdot \mathbf{n}$ values. The storage for the truncated SVD components is $\mathbf{k} \cdot (\mathbf{m} + \mathbf{n} + \mathbf{1})$ values (for $\mathbf{U}_k$, $\mathbf{V}_k$, and $\mathbf{\Sigma}_k$).
   Low k: Storage is very small. The compression ratio $\frac{mn}{k(m+n+1)}$ is very high.
   High k: Storage increases linearly with k. The compression ratio is low. If $k > \frac{mn}{m+n+1}$, you actually experience data expansion (no compression).