Software Assignment-An Image Compression Model Using SVD

Kishora Karthik-EE25BTECH11034

Contents

1 Summary of Video

The **Singular Value Decomposition (SVD)** is a fundamental factorization that applies to any general matrix A of dimensions m*n. The factorization is given by

$$A = U\Sigma V^T$$

.

The SVD breaks down the matrix A into three matrices:

1. U: An orthogonal matrix whose columns (u_i) are the left singular vectors.
2. Σ: A diagonal matrix containing the singular values (σ_i).
3. V: An orthogonal matrix whose columns (v_i) are the right singular vectors.

The main idea is to find an **orthonormal basis** in the row space (V) that gets mapped to an **orthonormal basis** in the column space (U).

The relationship connecting these vectors is

$$Av_i = \sigma_i u_i$$

. This equation shows that the matrix A takes a vector v_i from the row space, stretches it by a factor σ_i, and points it in the direction of u_i in the column space.

The SVD provides the perfect orthonormal bases for the **four fundamental subspaces**:

- **Row Space:** Basis is $v_1, \ldots, v_r$
- **Column Space:** Basis is $u_1, \ldots, u_r$
- **Null Space:** Basis is $v_{r+1}, \ldots, v_n$
- **Left Null Space ($N(A^T)$):** Basis is $u_{r+1}, \ldots, u_m$

2 SVD: Core Equations and Derivation

1. **The Singular Value Decomposition (SVD):**

$$A = U\Sigma V^T$$

2. **Components:**

- A: Any $m \times n$ matrix.
- U: An $m \times m$ orthogonal matrix ($U^TU = I$). Its columns are the left singular vectors.
- Σ: An $m \times n$ diagonal matrix. Its diagonal entries $\sigma_1, \sigma_2, \ldots, \sigma_r$ are the singular values.
- V^T: An $n \times n$ orthogonal matrix ($V^TV = I$). Its rows are the transpose of the right singular vectors.

3. **Finding V and Σ (from A^TA):**

$$A^TA = (U\Sigma V^T)^T(U\Sigma V^T) = V\Sigma^TU^TU\Sigma V^T = V(\Sigma^T\Sigma)V^T$$

This is the eigen decomposition of the symmetric matrix A^TA.

- The columns of V are the eigenvectors of A^TA.
- $\Sigma^T\Sigma$ is an $n \times n$ diagonal matrix with entries σ_i^2, which are the eigenvalues of A^TA.

4. **Finding U (from AA^T):**

$$AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma V^T V\Sigma^T U^T = U(\Sigma\Sigma^T)U^T$$

This is the eigen decomposition of the symmetric matrix AA^T.

- The columns of U are the eigenvectors of AA^T.
- $\Sigma\Sigma^T$ is an $m \times m$ diagonal matrix with entries σ_i^2, the same non-zero eigenvalues as A^TA.

3 Eigenvalue Calculation via Jacobi's Method

I have chosen the Jacobi's Method for finding the eigenvectors and eigenvalues. The Jacobi eigenvalue algorithm is an iterative method for finding the eigenvalues and eigenvectors of a real symmetric matrix A. This algorithm diagonalizes a matrix by applying a sequence of similarity transformations (also called rotations) which iteratively turn the matrix into diagonal form as eigenvalues of a matrix remain unchanged even after undergoing similarity transformations.

The algorithm works by applying a sequence of orthogonal similarity transformations, $A_{k+1} = J_k^T A_k J_k$, where each J_k is a special orthogonal matrix called a **Givens rotation**. The goal is to choose J_k at each step to eliminate a specific off diagonal element a_{pq}.

3.1 Givens Rotation

A Givens rotation matrix $J(p, q, \theta)$ is an identity matrix, except for four entries in rows and columns p and q:

$$J(p,q,\theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & \cos\theta & \dots & \sin\theta & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -\sin\theta & \dots & \cos\theta & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow \text{row } p \\ \\ \leftarrow \text{row } q \\ \\ \\ \end{matrix}$$

$$\begin{matrix} & & & \uparrow & & \uparrow & \\ & & & \text{col } p & & \text{col } q & \end{matrix}$$

This transformation $A' = J^TAJ$ only affects rows and columns p and q of A. The 2×2 submatrix formed by these rows and columns is transformed as:

$$\begin{pmatrix} a'_{pp} & a'_{pq} \\ a'_{qp} & a'_{qq} \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

Since A is symmetric, $a_{pq} = a_{qp}$. We should choose θ such that the new off-diagonal elements a'_{pq} and a'_{qp} are zero. The calculation for a'_{pq} yields:

$$a'_{pq} = (a_{qq} - a_{pp})\sin\theta\cos\theta + a_{pq}(\cos^2\theta - \sin^2\theta)$$

Using the trignometrtic identities $\sin(2\theta) = 2\sin\theta\cos\theta$ and $\cos(2\theta) = \cos^2\theta - \sin^2\theta$, we get:

$$a'_{pq} = \frac{1}{2}(a_{qq} - a_{pp})\sin(2\theta) + a_{pq}\cos(2\theta)$$

Put $a'_{pq} = 0$ to find the required angle:

$$-a_{pq}\cos(2\theta) = \frac{1}{2}(a_{qq} - a_{pp})\sin(2\theta)$$

$$\frac{\cos(2\theta)}{\sin(2\theta)} = \cot(2\theta) = \frac{a_{pp} - a_{qq}}{2a_{pq}}$$

We can solve for $\cos\theta$ and $\sin\theta$ from this relation.

3.2 The Cyclic Iterative Process

Unlike the classical Jacobi method which finds the largest off-diagonal element at each step, the Cyclic Jacobi method simply sweeps through all off-diagonal elements in a fixed order. A common way is to iterate row by row, targeting each element a_{pq} where $p < q$.

A single sweep consists of performing a rotation for all $n(n-1)/2$ off-diagonal pairs. The algorithm then repeats these sweeps until the matrix is sufficiently diagonal.

The sum of the squares of the off-diagonal elements, $S(A) = \sum_{i \neq j} a_{ij}^2$, must decrease with each rotation as:

$$S(A_{k+1}) = S(A_k) - 2a_{pq}^2$$

This guarantees that the sequence of matrices A_k converges to a diagonal matrix Λ.

$$\lim_{k\to\infty} A_k = \Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$$

The diagonal entries of Λ are the eigenvalues of the original matrix A.

3.3 Convergence Checks

Main Check (Diagonal Stability): The most common and computationally efficient check is to check the change in the diagonal elements. After a full sweep, the algorithm stores the current diagonal elements. After the next sweep, it compares the new diagonal elements to the stored ones. If the change in sum of diagonal elements is below a chosen tolerance ϵ, the algorithm is considered to have converged.

$$\sum_{i}^{n} |d_i^{(\text{new sweep})} - d_i^{(\text{old sweep})}| < \epsilon$$

This method is efficient as it only requires storing and comparing n values per sweep ($O(n)$ operations).

Alternative Check (Off-Diagonal Sum): An alternative method is to monitor the sum of the squares of the off-diagonal elements, $S(A)$. The algorithm stops when $S(A)$ is below a tolerance.

$$S(A) = \sum_{i \neq j} a_{ij}^2 < \epsilon$$

While this is a direct measure of diagonality, it is more computationally intensive as it requires a full-matrix summation ($O(n^2)$ operations) at the end of each sweep.

Eigenvectors: The matrix of eigenvectors Q is the product of all the Givens rotations applied:

$$Q = J_1 J_2 J_3 \ldots J_k \ldots$$

We must initialize an eigenvector matrix $V = I$ (the identity matrix) and update it at each step:

$$V_{k+1} = V_k J_k$$

When the algorithm converges, V will have accumulated all the rotations, and its columns will be the eigenvectors of A.

4 Pseudocode for Cyclic Jacobi Method

```
FUNCTION jacobi(n, arr, V):
// arr is the n x n symmetric matrix to be diagonalized
// V is an n x n matrix, initialized to Identity, to store eigenvectors

CONSTANT EPSILON = 1.0e-10
CONSTANT MAX_SWEEPS = 100

// Create 'old_diagonal' array of size n
DECLARE old_diagonal AS ARRAY[n]

// Store the initial diagonal
FOR i FROM 0 TO n-1:
    old_diagonal[i] = arr[i, i]
END FOR

// Start the main iteration (sweeps)
FOR s FROM 0 TO MAX_SWEEPS-1:

    // Iterate over all unique off-diagonal pairs (i, j)
    FOR i FROM 0 TO n-1:
        FOR j FROM i+1 TO n-1:

            // 1. Check if rotation is needed
            IF absolute_value(arr[i, j]) < EPSILON:
                CONTINUE // Element is already "zero", skip
            END IF

            // 2. Calculate rotation angle (theta)
            IF absolute_value(arr[i, i] - arr[j, j]) < EPSILON:
                // Diagonal elements are too close; use a fixed 45-degree angle
                IF arr[i, j] > 0:
                    theta = PI / 4
                ELSE:
                    theta = -PI / 4
                END IF
            ELSE:
                // Standard calculation for theta
                theta = 0.5 * arc_tangent2(2 * arr[i, j], arr[i, i] - arr[j, j])
            END IF

            // 3. Get sin(theta) and cos(theta)
            c = cosine(theta)
            s = sine(theta)

            // 4. Apply rotation A' = R^T * A * R
            //    (This updates the matrix 'arr')

            // Store old values
            arr_ii = arr[i, i]
```

```
        arr_jj = arr[j, j]
        arr_ij = arr[i, j]

        // Update 2x2 diagonal block
        arr[i, i] = c*c*arr_ii + 2.0*s*c*arr_ij + s*s*arr_jj
        arr[j, j] = s*s*arr_ii - 2.0*s*c*arr_ij + c*c*arr_jj

        // Zero out the off-diagonal pair
        arr[i, j] = 0
        arr[j, i] = 0

        // Update the i-th and j-th rows and columns
        FOR k FROM 0 TO n-1:
            IF k != i AND k != j:
                // Store old values
                arr_ki = arr[k, i]
                arr_kj = arr[k, j]

                // Update row k (A' = R^T * A)
                arr[k, i] = c*arr_ki + s*arr_kj
                arr[k, j] = -s*arr_ki + c*arr_kj  // <-- Corrected rotation

                // Update column k (A' = ... * R)
                arr[i, k] = arr[k, i]
                arr[j, k] = arr[k, j]
            END IF
        END FOR

        // 5. Apply rotation V' = V * R
        //    (This accumulates the rotations into 'V')
        FOR k FROM 0 TO n-1:
            // Store old values
            v_ki = V[k, i]
            v_kj = V[k, j]

            // Update columns i and j of V
            V[k, i] = c*v_ki + s*v_kj
            V[k, j] = -s*v_ki + c*v_kj
        END FOR

    END FOR // end j
END FOR // end i

// 6. Check for convergence after a full sweep
total_diagonal_change = 0.0
FOR i FROM 0 TO n-1:
    total_diagonal_change += absolute_value(arr[i, i] - old_diagonal[i])
    old_diagonal[i] = arr[i, i] // Store new diagonal for next sweep
END FOR

IF total_diagonal_change < EPSILON:
```

```
        BREAK // Converged, exit sweep loop
    END IF

END FOR // end s (sweeps)


END FUNCTION
```

5 Comparison of SVD Algorithms

Several SVD and eigenvalue algorithms were researched while making a choice:

- **Jacobi's Method:** An iterative method that uses a sequence of 2D plane rotations (Givens rotations) to eliminate off diagonal elements, iteratively making the matrix diagonal.

  Pros: It has very high numerical stability and very high accuracy as well. It is a relatively simple concept to understand and easy to implement.

  Cons: Slow for large matrices ($O(n^3 \times \text{sweeps})$), not competitive with modern methods in terms of speed for large matrices.

- **QR Algorithm (e.g., Golub-Kahan-Reinsch):** It is the industry standard, used in libraries like LAPACK (which powers MATLAB, NumPy, etc.). It first reduces the matrix to a bidiagonal form using Householder reflections, then iteratively applies QR shifts to find the singular values.

  Pros: Very good numerical stability and convergence. Very fast and efficient for finding all singular values of a dense matrix ($O(n^3)$ total).

  Cons: Very complex to implement.

- **Lanczos Algorithm:** An iterative method that projects the matrix onto a smaller "Krylov" subspace. It is very good at finding the eigen values with highest magnitude.

  Pros: Extremely fast and memory-efficient for finding the **top k** singular values ($k \ll n$). It is the standard for large, *sparse* matrices because it only requires matrix-vector products.

  Cons: Not designed to find *all* singular values. Can have numerical stability issues without complex re-orthogonalization steps. It is difficult to ensure numerical stability in implementation.

  Preferred For: Large, **sparse** matrices (e.g., social networks) where you only need the most important singular values.

- **Block Power Iteration:** An extension of the simple "power method." It iteratively multiplies a block of k random vectors by the matrix $(A^T A)$, which causes the vectors to converge to the top k eigenvectors.

  Pros: Conceptually simpler than Lanczos or QR. Easy to implement and it has complexity of $O(kn^2)$ so can be used for large matrices as well.

  Cons: Numerically unstable. The k vectors will all collapse onto the single dominant eigenvector unless a full QR decomposition is performed *inside every single iteration*, which is very expensive. provides us with top k eigen vectors instead of full SVD decomposition.

- **Randomized SVD (R-SVD):** A modern, probabilistic approach. It is preferred for big data problems involving massive, dense matrices where a low-rank approximation is enough.

  Pros: Very fast for large, dense matrices

  Cons: It is an approximation, not an exact decomposition (like block power iteration) though accuracy is high.

5.1 Reason for the choice

I have gone forward with implementation using the Jacobi's method as it is easy to understand conceptually and simple to implement. This makes it easier to debug the code in case any issues arise. Jacobi's method also gives us very good accuracy in the eigenvalues and eigenvectors. Since Jacobi's method provides us with the full SVD decomposition, we can create a different function to find the top k singular values. This would be useful if we need to reconstruct the matrix using multiple k values.
Other algorithms such as Lanczos are much faster but are difficult to implement.

6 Results and Error Analysis

- **Einstein** The original image and reconstructed images for different values of k are given below:

7 Analysis of Result

The reconstructed images show that we can use SVD in practice to compress images.

8 Trade-off in k vs Image Quality

As the number of top singular values(k) increases the image quality increases noticably. Howvever, it appears that the rate of increase in quality reduces at higher k values.This can be shown by the data presented in the tables as well.
At lower k values, we can observe the general shape but the image is very blurry and nor sharp. At higher k values the image is much more refines and sharp outlines can be visible. At the higher end, there is almost no change in the error.

9 Reflections and Conclusion

Hence, we can conclude that it is feasible to implement an image processing algorithm using the core and fundamental mathematical concepts of Singular Value Decomposition(SVD). This is also a good exercise to get familiar to the image libraries.