Image Compression Using Truncated SVD

INDHIRESH S- EE25BTECH11027

CONTENTS

I. Basics behind an Image

- A full grayscale image is visible by the combination of varying single intensity (or brightness) values at each and every pixel.
- The pixel intensity can vary from 0 to 255 ,where 0 = *black* and 255 = *white*

II. grayscale image as a matrix

- A grayscale image can be represented as a matrix $A \in R^{m \times n}$,where each entry a_{ij} corresponds to the pixel intensity

III. Singular Value Decomposition (SVD)

- SVD is a factorization method for the factorization of matrix
- SVD is the "final and best factorization of a matrix". While other factorizations (like LU decomposition or eigendecomposition) have limitations.
- SVD works for any matrix A of any size $(m \times n)$
- According to SVD, the given matrix A can be written as

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

where

$$U = \begin{pmatrix} \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} & \dots & \mathbf{u_m} \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{pmatrix}$$

$$V = \begin{pmatrix} \mathbf{v_1} & \mathbf{v_2} & \mathbf{v_3} & \dots & \mathbf{v_n} \end{pmatrix}$$

The matrix A can be represented as:

$$A = \sigma_1 \mathbf{u_1} \mathbf{v_1}^T + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T + \cdots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T$$

where r is the rank of the matrix A

A. *The Three Components*

1) V(Right Singular Vectors):

- This is an $n \times n$ orthogonal matrix. Its columns $(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n)$ are the right singular vectors.
- The columns of V are the orthonormal eigenvectors of A^TA. (A^TA is a symmetric, positive semi-definite matrix, so its eigenvectors are always orthogonal).

2) U (Left Singular Vectors):

- This is an $m \times m$ orthogonal matrix. Its columns $(\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m)$ are the left singular vectors.
- The columns of U are the orthonormal eigenvectors of AA^T.

3) Σ (Singular Values):

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{pmatrix}$$

- This is an $m \times n$ rectangular diagonal matrix.
- Its diagonal entries $(\sigma_1, \sigma_2, \dots)$ are the singular values of A. They are always positive or zero and are sorted from largest to smallest.
- The singular values σ_i are the square roots of the non-zero eigenvalues of both $A^T A$ and AA^T (they share the same eigenvalues).

B. The Geometric Intuition

Geometrically,SVD breaks down any linear transformation (A) into three simple steps:

- V^T (Rotation): An orthogonal matrix that rotates the input space.
- Σ (Scaling/Stretching): A diagonal matrix that stretches or shrinks the space along its axes.
- U (Rotation): Another orthogonal matrix that rotates the output space.

C. SVD and the Four Subspaces

A key insight Strang emphasizes is that the SVD provides a perfect orthonormal basis for all four fundamental subspaces of the matrix A:

- The first r columns of V (where r is the rank) form a basis for the Row Space
- The remaining columns of V form a basis for the Nullspace.
- The first r columns of U form a basis for the Column Space.
- The remaining columns of U form a basis for the Left Nullspace.

D. Low-Rank Approximation (The Core of Compression)

The most important application, which is main focus of this project, is that the SVD writes the matrix A as a sum of rank-one matrices, sorted by importance.

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \sigma_3 \mathbf{u}_3 \mathbf{v}_3^T + \cdots + \sigma_r \mathbf{u}_r \mathbf{v}_r^T$$

- The first term $(\sigma_1 \mathbf{u}_1 \mathbf{v}_1^T)$ is the most important rank-one piece of the matrix because σ_1 is the largest singular value.
- The second term is the next most important, and so on.
- A truncated SVD (your A_k) is created by simply stopping this sum after k terms. This A_k is the best possible rank-k approximation of the original matrix A.

IV. Implemented algorithm to find SVD

- The algorithm used in this project is **Power iteration and deflation**.
- The core objective is to find the best rank-k approximation, A_k, of an $m \times n$ image matrix A
- The approximation A_k is defined as the sum of the top k rank-one components:

$$A_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

- To find these components, this implementation ,uses the Power Iteration with Deflation method

A. *Why is this method chosen over other methods?*

My choice to implement the Power Iteration with Deflation method was chosen over other methods because of direct comparison of its trade-offs against the others, specifically for this project's requirements.

*1) **Simplicity of Implementation**:*

- The core logic is simple. It relies only on fundamental matrix operations
- Other methods like the Jacobi Algorithm are far more complex. They require implementing numerically sensitive "Givens rotations" and a "sweep" strategy to diagonalize a matrix. Methods like Subspace Iteration or Randomized SVD are more stable but require implementing a full QR decomposition (e.g., Gram-Schmidt), which is a complex algorithm in its own right.

*2) **Directly finding Truncated SVD**:*

- The project asks for the "top k singular values/vectors". The Power Iteration method is designed for this. It finds the components one by one, from σ_1 down. I can simply stop the main loop after k iterations.
- Other methods like Jacobi Algorithm is a "full SVD" method. To find the top 50 components of a 1024x1024 image, it would waste an enormous amount of computation finding all 1024 components, only for me to throw most of them away. This is computationally inefficient for this project's goal.

*3) **Avoids Forming the A^TA Matrix**:*

- This method never builds the A^TA matrix.It only computes the action of this matrix on a vector.
- Using this matrix-free method saves the $O(n^2)$ memory required to store the full A^TA matrix.
- It's also gives numerical stability.Forming A^TA can be numerically unstable.It can lead to a significant loss of precision for the smaller singular values.

V. Math behid the implemented algorithm

- The first step of the algorithm is to find the largest singular value σ_1 and its corresponding right and left singular vectors v_1 and u_1
- The first step is achieved by the Power iteration of the original given matrix
- Here v_1 is the eigenvector of A^TA corresponding to the largest eigenvalue .

A. *The Power Iteration*

- In power iteration, the first step is to create a random $n \times 1$ guess vector $\mathbf{v}$ and normalize it as:

$$\mathbf{v} = \frac{\mathbf{v}}{||\mathbf{v}||}$$

- After assuming a random vector, the following operation should be done to amplify the part of $\mathbf{v}$ that points in the most stretched direction ($\mathbf{v_1}$) and shrinks all the other parts.

$$\mathbf{v_{new}} = (A^T A)\mathbf{v}$$
$$\mathbf{v_{new}} = \frac{\mathbf{v_{new}}}{||\mathbf{v_{new}}||}$$

- Iterate the above operation untill the vector $\mathbf{v_{new}}$ stops converging.
- Now the $\mathbf{v_{new}}$ obtained after all iterations is the required right singular vector $\mathbf{v_1}$
- After finding $\mathbf{v_1}$, the other components σ_1 and $\mathbf{u_1}$ can be calculated by the following steps:
  By definition

$$A\mathbf{v_1} = \sigma_1 \mathbf{u_1}$$

  By taking norm on both sides:

$$||A\mathbf{v_1}|| = ||\sigma_1 \mathbf{u_1}|| = \sigma_1 ||\mathbf{u_1}||$$

  Since u_1 is a unit vector

$$||A\mathbf{v_1}|| = \sigma_1$$

  For finding the u_1, we simply rearrange the definition

$$\mathbf{u_1} = \frac{A\mathbf{v_1}}{\sigma_1}$$

B. *Deflation*

For finding $\mathbf{v_2}$ we can't run the power iteration again because it'll again find the $\mathbf{v_1}$. So we need to deflate the original matrix .
The full matrix A is given by :

$$A = \sigma_1 \mathbf{u_1} \mathbf{v_1}^T + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T + \cdots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T$$

Now implementing the deflation method to the original matrix we get the modified new matrix A_{new}
A_{new} can be calculated by

$$A_{new} = A - \sigma_1 \mathbf{u_1} \mathbf{v_1^T}$$

$$A_{new} = \sigma_2 \mathbf{u_2} \mathbf{v_2}^T + \cdots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T$$

Now by applying the power iteration to the deflated matrix (A_{new}) we will get $\mathbf{v_2}$,σ_2 and $\mathbf{u_2}$
The above two processes(Power iteration and deflation) are repeated k times

C. *Forming the final comopressed matrix*

The final compressed matrix A_k can be given by:

$$A_k = \sum_{i=1}^{k} \sigma_i \mathbf{u_i} \mathbf{v_i}^T$$

VI. Pseudo Code

A. *Main function*

```
PROGRAM SVD_COMPRESS

  // --- 1. Initialization ---
  k = 20 // Set desired rank

  // Load image, get dimensions m and n
  a, m, n = imageload("greyscale.png")
  IF a is NULL THEN EXIT

  // --- 2. K-Value Constraint ---
  max_k = MIN(m, n)
  IF k > max_k THEN k = max_k

  // --- 3. Memory Allocation ---
  sigma = AllocateVector(k)
  a2 = AllocateMatrix(m, n)
  a3 = AllocateMatrix(m, n)
  t = AllocateMatrix(m, n)
  aT = AllocateMatrix(n, m)
  v_old = AllocateVector(n)
  v1 = AllocateMatrix(n, k)
  u = AllocateMatrix(m, k)

  // --- 4. Backup and Zero Initialization ---
  FOR i FROM 0 TO m-1:
    FOR j FROM 0 TO n-1:
      a2[i*n + j] = a[i*n + j] // Backup original matrix
      t[i*n + j] = 0.0 // Zero out the final matrix
    END FOR
  END FOR

  // --- 5. Main SVD Loop (Iterate k times) ---
  FOR i FROM 0 TO k-1:
    v = CreateRandomVector(n)

    // Calculate the transpose of the current (deflated) matrix 'a'
```

```
    trans(m, n, a, aT)

    // Find the dominant singular vector 'v'
    converge(m, n, a, aT, v, v_old)

    // Store 'v' in the 'v1' matrix
    FOR j FROM 0 TO n-1:
      v1[j*k + i] = v[j]
    END FOR

    // Calculate sigma and u
    temp = matvec(m, n, a, v)
    sigma[i] = norm(temp)

    // Normalize 'temp' to get 'u'
    IF sigma[i] > 1e-9 THEN
      temp = temp / sigma[i]
    END IF

    // Store 'u' (which is 'temp') in the 'u' matrix
    FOR j FROM 0 TO m-1:
      u[j*k + i] = temp[j]
    END FOR

    // --- Deflation and Reconstruction ---
    a1 = AllocateMatrix(m, n)

    // 1. Calculate the rank-one component: a1 = sigma * u * v^T
    rankonecomponent(m, n, sigma[i], temp, v, a1)

    // 2. Deflate 'a': a = a - a1
    matrixsubtract(m, n, a, a1)

    // 3. Reconstruct 't' (A_k): t = t + a1
    FOR p FROM 0 TO m-1:
      FOR q FROM 0 TO n-1:
        t[p*n + q] = t[p*n + q] + a1[p*n + q]
      END FOR
    END FOR

    free(a1)
  END FOR

// --- 6. Error Calculation ---
  FOR i FROM 0 TO m-1:
```

```
    FOR j FROM 0 TO n−1:
      a3[i∗n + j] = a2[i∗n + j] − t[i∗n + j]
    END FOR
  END FOR

  errornorm = frobeniusnorm(m, n, a3)
  originalnorm = frobeniusnorm(m, n, a2)
  percentageerror = (errornorm / originalnorm) ∗ 100

  // −−− 7. Output −−−
  PRINT "absolute error:", errornorm
  PRINT "percentage error:", percentageerror

  pngimage("einstein_k20.png", m, n, t)
  jpgimage("einstein_k20.jpg", m, n, t)

  // −−− 8. Cleanup −−−
  free(a, a2, a3, t, aT, v1, u, v_old)

END PROGRAM
```

B. Helper functions

```
// −−− Image I/O −−−

FUNCTION imageload(filename, m_out, n_out):
  // Load the image file, forcing it to 1 channel (grayscale)
  image_data, width, height = stbi_load(filename, 1)

  IF image_data is NULL THEN
    PRINT "Error loading image"
    RETURN NULL
  END IF

  // Allocate memory for the double matrix
  matrix = AllocateMatrix(width, height)
  IF matrix is NULL THEN
    stbi_image_free(image_data)
    RETURN NULL
  END IF

  // Copy and convert pixel data from char to double
  FOR i FROM 0 TO height−1:
```

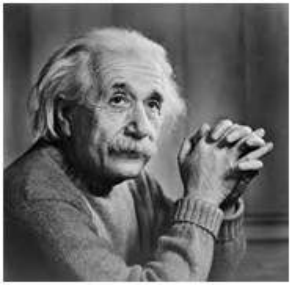
```
    FOR j FROM 0 TO width−1:
      pixel_value = image_data[i ∗ width + j]
      matrix[i ∗ width + j] = (double)pixel_value
    END FOR
  END FOR

  stbi_image_free(image_data)
  ∗m_out = height
  ∗n_out = width
  RETURN matrix
END FUNCTION

PROCEDURE pngimage(filename, m, n, matrix):
  // Allocate temporary buffer for 8−bit pixels
  out_pixels = AllocateVector(m ∗ n)

  // Convert and clamp the double matrix back to 8−bit char
  FOR i FROM 0 TO (m∗n)−1:
    pixel_value = matrix[i]
    IF pixel_value < 0 THEN pixel_value = 0
    IF pixel_value > 255 THEN pixel_value = 255
    out_pixels[i] = (char)round(pixel_value)
  END FOR

  // Save the 8−bit data as a PNG
  stbi_write_png(filename, n, m, 1, out_pixels)

  free(out_pixels)
END PROCEDURE

PROCEDURE jpgimage(filename, m, n, matrix):
  // Allocate temporary buffer for 8−bit pixels
  out_pixels = AllocateVector(m ∗ n)

  // Convert and clamp the double matrix back to 8−bit char
  FOR i FROM 0 TO (m∗n)−1:
    pixel_value = matrix[i]
    IF pixel_value < 0 THEN pixel_value = 0
    IF pixel_value > 255 THEN pixel_value = 255
    out_pixels[i] = (char)round(pixel_value)
  END FOR

  // Save the 8−bit data as a JPG with quality 30
  SET quality = 30
  stbi_write_jpg(filename, n, m, 1, out_pixels, quality)
```

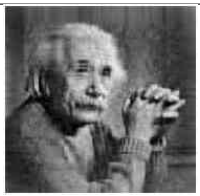
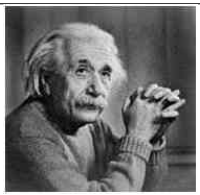
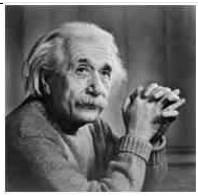
```
  free(out_pixels)
END PROCEDURE
// --- Core SVD Logic ---

PROCEDURE converge(m, n, a, aT, v, v_old):
  normalizeVector(v)
  SET max_iterations = 25
  SET tolerance = 1e-4

  FOR i FROM 0 TO max_iterations-1:
    v_old = COPY(v)

    // Power Iteration step: v = (A^T * A) * v
    b = matvec(m, n, a, v)
    c = matvec(n, m, aT, b)
    v = c
    normalizeVector(v)

    // Check for convergence
    diff = NORM(v - v_old)
    flip_diff = NORM(v + v_old)
    IF diff < tolerance OR flip_diff < tolerance THEN
      BREAK
    END IF
  END FOR
END PROCEDURE

PROCEDURE rankonecomponent(m, n, sigma, u, v, k_matrix):
  FOR i FROM 0 TO m-1:
    FOR j FROM 0 to n-1:
      k_matrix[i*n + j] = sigma * u[i] * v[j]
    END FOR
  END FOR
END PROCEDURE

// --- Matrix Utilities ---

PROCEDURE matrixsubtract(m, n, a, b):
  FOR i FROM 0 TO (m*n)-1:
    a[i] = a[i] - b[i]
  END FOR
END PROCEDURE

FUNCTION frobeniusnorm(m, n, matrix):
```

```
  sum = 0
  FOR i FROM 0 TO (m*n)-1:
    sum = sum + matrix[i] * matrix[i]
  END FOR
  RETURN sqrt(sum)
END FUNCTION
```

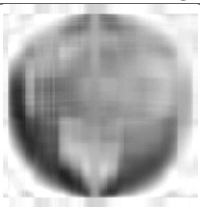
VII. Reconstructed Image for different k values and corresponding errors

A. *EXAMPLE-1*

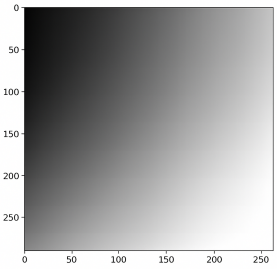


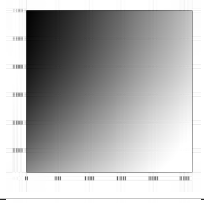
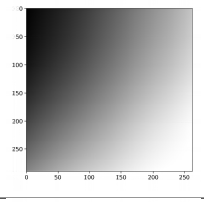
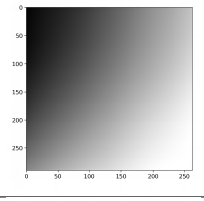
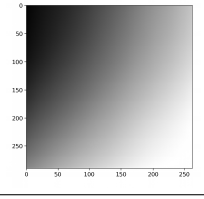
K-value	Reconstructed Image	Frobenius Error	Percentage Error
5		4714.5822	21.6173%
20		2126.5948	9.7508%
50		880.6793	4.0380%
100		164.7852	0.7555%

B. EXAMPLE-2

K-value	Reconstructed Image	Frobenius Error	Percentage Error
5		20704.2745	13.0780%
20		10635.0552	6.7177%
50		6187.1154	3.9081%
100		3673.3421	2.3203%

C. *EXAMPLE-3*



K-value	Reconstructed Image	Frobenius Error	Percentage Error
5		11146.3103	5.7585%
20		3808.1983	1.9674%
50		1160.1552	0.5993%
100		512.3602	0.2647%

VIII. Trade-offs

1) The chosen method is the most straightforward to implement from scratch. Its logic is built using only fundamental functions (matrix-vector multiply, transpose, norm, and subtract).
2) This method's main weakness is the deflation step. Any tiny error in finding the first component ($\sigma_1, \mathbf{u}_1, \mathbf{v}_1$) is "baked into" the matrix when it get subtracted . This error propagates and gets worse with each step.