Image Compression using Singular Value Decomposition

EE25BTECH11042 - Nipun Dasari

1. Introduction

An image can be represented as a matrix A, where each entry is a pixel intensity. The Singular Value Decomposition (SVD) expresses A as:

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

where U and V are orthonormal matrices and Σ contains singular values in decreasing order. By keeping only the top r singular values and vectors, we obtain a rank-r approximation:

$$A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^T.$$

This allows image compression.

2. Summary of Strang's Explanation of SVD

Gilbert Strang highlights the geometric meaning of SVD:

1. V rotates the coordinate system into principal input directions.
2. Σ stretches or shrinks along those axes (singular values).
3. U rotates the output back to image space.

Thus, A maps a unit sphere to an ellipsoid whose axes are the singular vectors and lengths are singular values. Keeping only r largest σ_i preserves most structure while discarding fine details.

Important properties to be kept on mind are that:

1. U is the orthogonal matrix obtained during eigenvalue decomposition of $\mathbf{A}^\top\mathbf{A}$
2. V is the orthogonal matrix obtained during eigenvalue decomposition of $\mathbf{A}^\top\mathbf{A}$
3. Σ is a diagonal matrix with square roots of eigenvalues.

In general SVD, we consider the column space and row space of the given matrix, obtain a basis for column space of $\mathbf{A}$ and a basis for row space of $\mathbf{A}$. Then find the sigma values using the basis obtained. The columns of V and U are represented as basis vectors of column space and row space.

3. Algorithm Used: Power Iteration SVD

Instead of computing full SVD, we compute top r singular vectors using repeated multiplication:

$$v \leftarrow \frac{A^T(Av)}{||A^T(Av)||}.$$

After convergence,

$$u = \frac{Av}{||Av||}, \qquad \sigma = ||Av||.$$

Orthogonalization is applied to ensure vectors remain independent.

4. Pseudo Code With Explanation

Let A be an $m \times n$ image matrix, and r be the rank chosen for compression.

```
1:  Choose the target rank $r$
2:  for $col = 1$ to $r$ do
3:      Initialize a random vector $v \in \mathbb{R}^n$ (this is a guess for a right singular vector)
4:      if  then
            r ¿ min(n,m)$r = min(n, m)$
5:          for iteration = 1 to 25 do
                Power Iteration step
6:              $Av \leftarrow A \cdot v$
7:              Normalize $Av$ so that $||Av|| = 1$
8:              $A^TAv \leftarrow A^T \cdot Av$ This applies $A^TA$ to $v$
9:              Orthogonalize: Remove components in directions of previously found vectors
                    $$A^TAv \leftarrow A^TAv - \sum_{p=1}^{col-1} (A^TAv \cdot v_p)v_p$$
10:             Normalize $A^TAv$ so that $||A^TAv|| = 1$
11:             Update $v \leftarrow A^TAv$
12:         end for
13:         Store the converged vector $v$ as $v_{col}$ (a right singular vector)
14:         Compute $u = A \cdot v_{col}$
15:         $\sigma_{col} = ||u||$ (this is the singular value)
16:         Normalize $u$ so that $||u|| = 1$
17:         Store $u$ in $U$ and $\sigma_{col}$ in $\Sigma$
18: end for
19: Reconstruction:
        $$A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$
```

5. C Code Used for Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
void normalize(double *v,int n){
double s=0; for(int i=0;i<n;i++) s+=v[i]*v[i];
s=sqrt(s); if(s<1e−12)s=1e−12;
//to avoid division with 0 if it occurs
for(int i=0;i<n;i++) v[i]/=s;}
int main(){
char in[100], out[100], magic[2];
int r,w,h,maxv;
printf("Input PGM filename: "); scanf("%s",in);
printf("Rank r: "); scanf("%d",&r);
printf("Output PGM filename: "); scanf("%s",out);
FILE *f=fopen(in,"rb");
if(!f){printf("Cannot open input file.\n"); return 1;}
fscanf(f,"%2s %d %d %d",magic,&w,&h,&maxv);
fgetc(f);
int m=h,n=w,N=m*n;
\\for easier understanding that its mxm matrix
unsigned char *img=malloc(N);+
fread(img,1,N,f);
fclose(f);
\\used 8 in place of sizeof(double)
double *A=malloc(8*N);
for(int i=0;i<N;i++) A[i]=img[i];
double *U=malloc(m*r*8), *S=malloc(r*8), *V=malloc(n*r*8), *v=malloc(n*8), *Av=
    malloc(m*8), *AtAv=malloc(n*8);
for(int col=0;col<r;col++){
for(int i=0;i<n;i++) v[i]=((double)rand()/RAND_MAX)−0.5;
for(int it=0;it<25;it++){
for(int i=0;i<m;i++){
        double s=0;
        for(int j=0;j<n;j++) s+=A[i*n+j]*v[j];\\finding matrix multiplication, since
            2−D arrays in C are row dominant we use this. When i=0 it gives jth
            element of 1st row
        Av[i]=s;}
        for(int j=0;j<n;j++){
double s=0;
        for(int i=0;i<m;i++) s+=A[i*n+j]*Av[i];
        AtAv[j]=s;}
for(int p=0;p<col;p++){
                double d=0;
        for(int i=0;i<n;i++) d+=AtAv[i]*V[i+p*n];
                for(int i=0;i<n;i++) AtAv[i]−=d*V[i+p*n];}
                normalize(AtAv,n);
        for(int i=0;i<n;i++) v[i]=AtAv[i];}
        for(int i=0;i<n;i++) V[i+col*n]=v[i];
        double sigma_sq=0;
        for(int i=0;i<m;i++){
                double s=0;
                for(int j=0;j<n;j++) s+=A[i*n+j]*v[j];
                U[i+col*m]=s;
                sigma_sq+=s*s;}
        S[col]=sqrt(sigma_sq);
        for(int i=0;i<m;i++) U[i+col*m]/=S[col];}
        double *R=malloc(8*N);
        for(int i=0;i<m;i++)
        for(int j=0;j<n;j++){
        double s=0;
                for(int t=0;t<r;t++) s+=U[i+t*m]*S[t]*V[j+t*n];
                R[i*n+j]=s;}
        for(int i=0;i<N;i++){
        double v=R[i]; if(v<0)v=0; if(v>255)v=255;
        img[i]=(unsigned char)(v+0.5);}
        FILE *g=fopen(out,"wb");
        fprintf(g,"P5\n%d´%d\n255\n",w,h);
        fwrite(img,1,N,g);
        fclose(g);
        double sum=0.0;
        for(int i=0;i<N;i++)
        sum+=(A[i]−R[i])*(A[i]−R[i]); sum = sqrt(sum); printf("%lf\n", sum);
        return 0;}
```

6. Comparison of Algorithms

- **Power Iteration (one I used)** â Simple for coding, only computes top *r* values, slower convergence. The reason I chose this is because it produces accurate enough results for the pictures provided for the assignment and saves time as the implementation in C is relatively simple.
- **Jacobi iteration** : Jacobi algorithm for SVD is an iterative method that computes the singular values and singular vectors of a matrix by repeatedly applying orthogonal rotations to eliminate the off-diagonal terms of $\mathbf{A}^\top\mathbf{A}$. It has tough implementation
- **Randomized SVD** : Faster for large images. Coding logic is critical and even the slightest bugs can lead to divergence. Computationally very efficient as only random matrix values are considered though only top eigenvalues come out as a consequence.
- **Full SVD** â Most accurate, slowest, unnecessary for compression. Finds all eigenvectors irrespective of how many eigenvectors you want to compute. Computationally very inefficient

7. Results

Original picture 1:

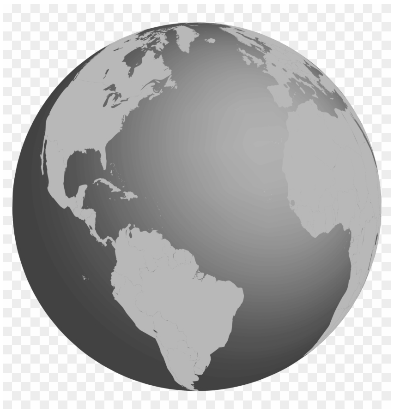

Fig. 3.1: Original Image

Rank = 5:

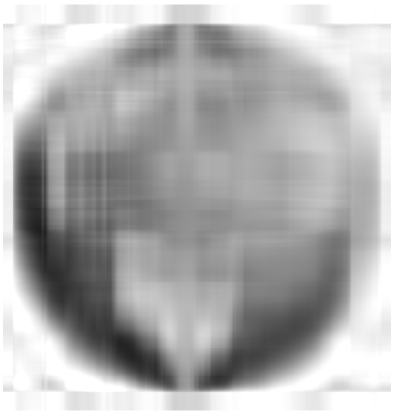

Fig. 3.2: Reconstruction with Rank $r = 5$

Rank = 10:

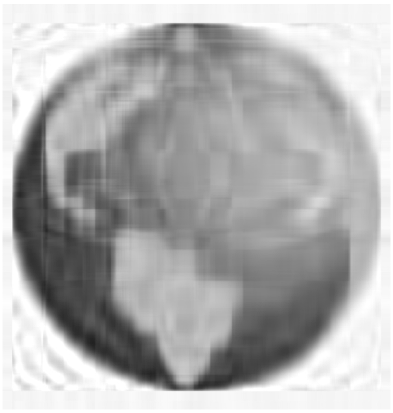

Fig. 3.3: Reconstruction with Rank $r = 10$

Rank = 100:

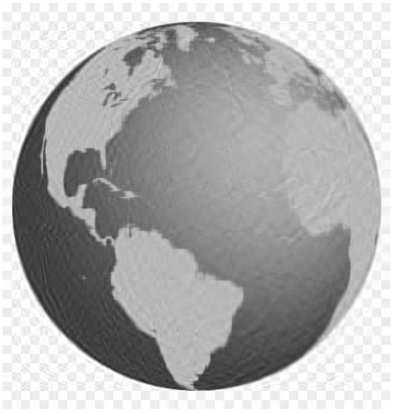

Fig. 3.4: Reconstruction with Rank $r = 100$

Rank = 200:

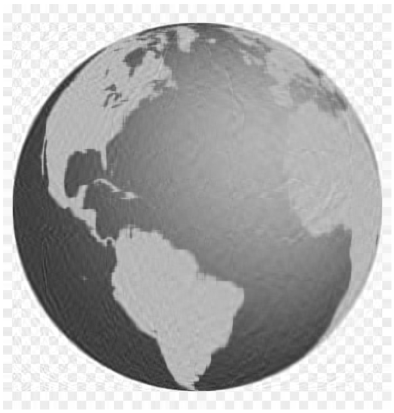

Fig. 3.5: Reconstruction with Rank r = 200

Original picture 2:

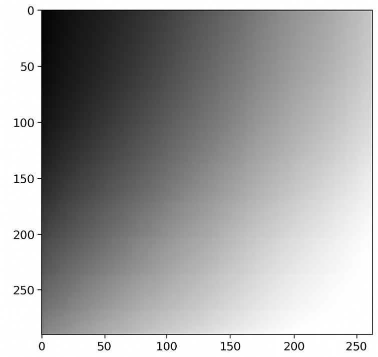

Fig. 3.6: Original Image

Rank = 5:

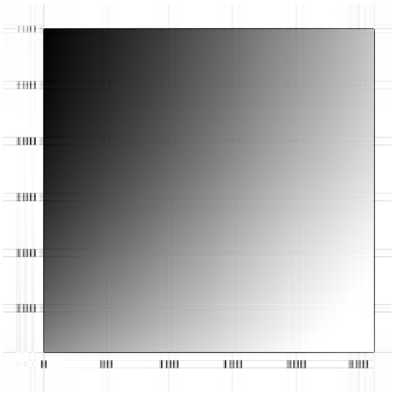

Fig. 3.7: Reconstruction with Rank $r = 5$

Rank = 10:

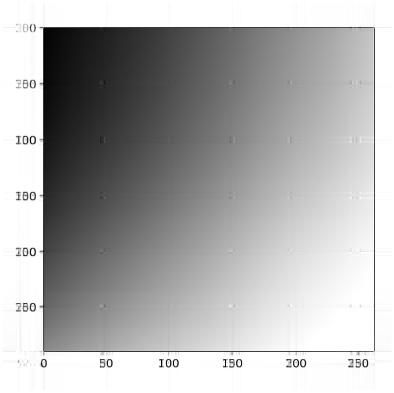

Fig. 3.8: Reconstruction with Rank $r = 10$

Rank = 100:

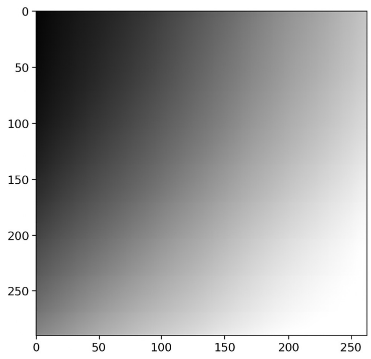

Fig. 3.9: Reconstruction with Rank $r = 100$

Rank = 200:

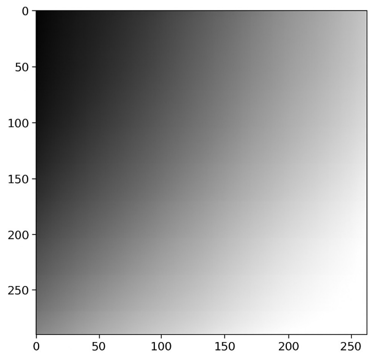

Fig. 3.10: Reconstruction with Rank $r = 200$

Original picture 3:

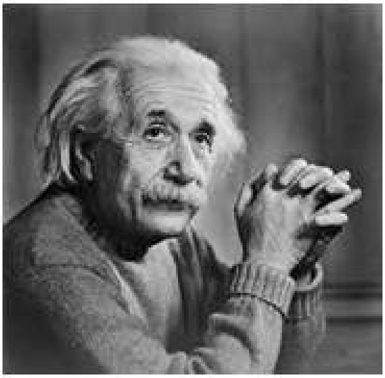

Fig. 3.11: Original Image

Rank = 5:

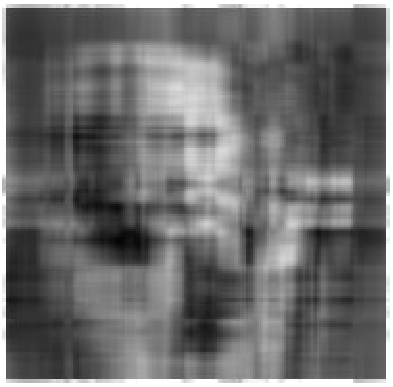

Fig. 3.12: Reconstruction with Rank $r = 5$

Rank = 10:

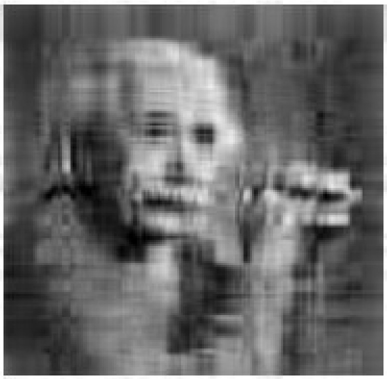

Fig. 3.13: Reconstruction with Rank r = 10

Rank = 100:

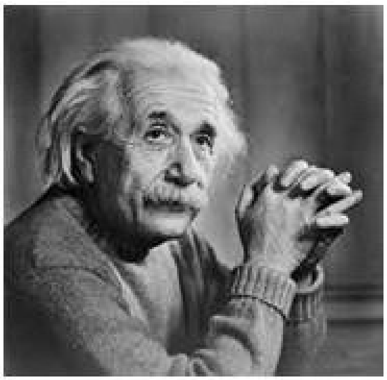

Fig. 3.14: Reconstruction with Rank r = 100

Rank = 150:

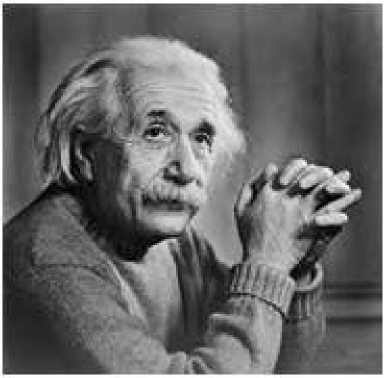

Fig. 3.15: Reconstruction with Rank $r = 150$

8. Error

Globe pic	
Assumed Rank	**Error$\|\|$A − R$\|\|$**
5	19614.792633
10	13575.944376
100	276.517125
200	175.206135
Greyscale	
Assumed Rank	**Error$\|\|$A − R$\|\|$**
5	11155.922808
10	7188.996389
100	572.409611
200	454.728867
Einstein pic	
Assumed Rank	**Error$\|\|$A − R$\|\|$**
5	4713.636021
10	3248.816601
100	164.797839
150	9.655560

Table : Error calculated via Frobenius norm

9. Conclusion

The power iteration method provides effective low-rank image compression. Quality increases with rank r, but computation time also rises. For most images, $r \in [20, 150]$ gives good results.