Image Compression w/ Singular Value Decomposition

EE25BTECH11062 - Vivek K Kumar

1 Summary of Strang's Video

If we take any $m \times n$ matrix $\mathbf{A}$, and take a look at its row space, and column space (which is of R dimensions, where R is the rank of the matrix) we can get R linearly independent vectors.

We can orthogonalize the basis in the row space and also the column space by Gram-Schmidt orthogonalization. So now, we have R orthogonal unit vectors $\mathbf{v_1}, \mathbf{v_2} \dots \mathbf{v_r}$. Then we can take $\mathbf{u_1}, \mathbf{u_2} \dots \mathbf{u_r}$ which are orthogonal unit vectors such that $\sigma \mathbf{u_i} = \mathbf{A}\mathbf{v_i}$ for $i \in \{1, 2, 3, \dots r\}$

We can combine the R equations above to make a matrix multiplication of the following form

$$\mathbf{A}\begin{pmatrix}\mathbf{v_1} & \mathbf{v_2}\dots & \mathbf{v_r}\end{pmatrix} = \begin{pmatrix}\sigma_1\mathbf{u_1} & \sigma_2\mathbf{u_2}\dots & \sigma_r\mathbf{u_r}\end{pmatrix} \tag{0.1}$$

Setting $\mathbf{V} = \begin{pmatrix}\mathbf{v_1} & \mathbf{v_2}\dots & \mathbf{v_r}\end{pmatrix}$ and $\mathbf{U} = \begin{pmatrix}\mathbf{u_1} & \mathbf{u_2}\dots & \mathbf{u_r}\end{pmatrix}$, and $\mathbf{\Sigma}$ is the diagonal rectangular matrix containing $\sigma_1, \sigma_2, \dots \sigma_r$

$$\mathbf{AV} = \mathbf{U\Sigma} \tag{0.2}$$

Since $\mathbf{U}, \mathbf{V}$ are orthogonal, Any $m \times n$ matrix $\mathbf{A}$ can be expressed as

$$\mathbf{A} = \mathbf{U\Sigma V}^\top \tag{0.3}$$

This is called the singular value decomposition. Here, $\mathbf{U}$ and $\mathbf{V}^\top$ are matrices with left singular vectors and right singular vectors respectively, and $\mathbf{\Sigma}$ is a matrix with diagonal elements containing singular values.

Now for computing the matrices, We take $\mathbf{A}^\top\mathbf{A}$ which is positive semi-definite and symmetric, and also $\mathbf{A}\mathbf{A}^\top$ with similar properties.

$$\mathbf{A}^\top\mathbf{A} = \mathbf{V\Sigma}^\top\mathbf{U}^\top\mathbf{U\Sigma V}^\top \tag{0.4}$$
$$= \mathbf{V\Sigma}^2\mathbf{V}^T \tag{0.5}$$
$$= \mathbf{V\Lambda V}^T \tag{0.6}$$

which is basically the eigenvalue decomposition. We know that it exists because of the symmetric nature of the matrix.

Similarly,

$$\mathbf{A}\mathbf{A}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top\mathbf{V}\mathbf{\Sigma}^\top\mathbf{U}^\top \tag{0.7}$$

$$= \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T \tag{0.8}$$

$$= \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \tag{0.9}$$

So, by basically doing eigenvalue decomposition for $A^\top A$ and $AA^\top$ we get the 3 matrices. The singular values are obtained by taking the square root of the eigenvalues.

$$\mathbf{\Lambda} = \mathbf{\Sigma}^2 \tag{0.10}$$

It was also summarized in the video that the vectors in the null space of $\mathbf{A}$ and $\mathbf{A}^\top$, are $\left(\mathbf{v_r}, \mathbf{v_{r+1}}, \dots \mathbf{v_n}\right)$ and $\left(\mathbf{u_r}, \mathbf{u_{r+1}}, \dots \mathbf{u_m}\right)$ respectively.

$$\mathbf{A}\mathbf{v_{r+1}} = 0 \tag{0.11}$$

$$\mathbf{A}\mathbf{v_{r+2}} = 0 \tag{0.12}$$

$$\dots\mathbf{A}\mathbf{v_n} = 0 \tag{0.13}$$

2 Math behind Image Compression

A grayscale image can be represented as a matrix of pixels, where each pixel can have integer values ranging from 0 to 255.

The image sizes will be very large if we try to store the data of each and every pixel. Hence, we need to optimize the space each image takes. One way to do this is by approximating the matrix(usually full rank) with a smaller k rank matrix.

2.1 Eckart-Young Theorem

The Eckart-Young theorem states the following

$$||\mathbf{A} - \mathbf{B}|| \geq ||\mathbf{A} - \mathbf{A_k}|| \tag{0.14}$$

Where, $\mathbf{B}$ is rank k and $||\mathbf{A} - \mathbf{B}||$ can be a

- Frobenius norm
- L2 norm
- Nuclear norm

$$\mathbf{A_k} = \sum_{i=1}^{k} \sigma_i \mathbf{u_1}\mathbf{v_1}^\top \tag{0.15}$$

$$\sigma_1 > \sigma_2 > \sigma_3 \dots \sigma_k \tag{0.16}$$

Here, $\mathbf{u_i}, \mathbf{v_i}$ are left and right singular vectors.

Hence, a good approximation can be obtained by performing a **truncated singular value decomposition**

2.2 Computing top K singular values

The top k singular values and vectors can be computed by computing the top k highest eigenvalues (and its corresponding eigenvectors) of $\mathbf{M} = \mathbf{A}^\top\mathbf{A}$. By doing this, we obtain

1) The top k singular values
2) Corresponding right orthonormal singular vectors

Given a singular value and its corresponding right singular vector, the left singular vector can be easily computed by the following equation

$$\mathbf{Av} = \sigma\mathbf{u} \text{ or} \tag{2.1}$$

$$\mathbf{u} = \frac{\mathbf{Av}}{||\mathbf{Av}||} \tag{2.2}$$

We can hence obtain a rank-k approximation of $\mathbf{A}$ by following the equation

$$\mathbf{A_k} = \sum_{i=1}^{k} \sigma_i \mathbf{u_1}\mathbf{v_1}^\top \tag{2.3}$$

$$\sigma_1 > \sigma_2 > \sigma_3 \ldots \sigma_k \tag{2.4}$$

The above representation is also referred to as truncated singular value decomposition.

One of the ways to find the top k highest eigenvalues is by finding the highest eigenvalue, subtracting the SVD component corresponding to that eigenvalue from $\mathbf{A}$, and finding the highest eigenvalue again. This process is repeated again and again to finally find the top k eigenvalues of $\mathbf{A}^\top\mathbf{A}$. The following subsection explains how this can be done.

2.3 Power Iteration

Power iteration is implemented for finding the highest eigenvalue and its corresponding eigenvector for a given matrix $\mathbf{M}$.

In this case we want to compute the eigenpair for $\mathbf{M}$ where,

$$\mathbf{M} = \mathbf{A}^\top\mathbf{A} \tag{2.5}$$

$$\mathbf{Mv_1} = \lambda\mathbf{v_1} \tag{2.6}$$

$$\mathbf{Mv_1} = \lambda\mathbf{v_1} \tag{2.7}$$

λ is the highest eigenvalue

The implementation of power iteration involves starting with a **random unit vector** $\mathbf{b_0}$, and finding $\mathbf{b_1} = \frac{\mathbf{Mb_0}}{||\mathbf{Mb_0}||}$. We keep doing this, until $\mathbf{b_p}$ converges to $\mathbf{v_1}$
$\mathbf{v_1}$ is the eigenvector corresponding to highest eigenvalue of $\mathbf{M}$

This can be represented as

$$||\mathbf{b_o}|| = 1, \mathbf{b_o} \in \mathbb{R}^n \tag{2.8}$$

$$\mathbf{b_{p+1}} = \frac{\mathbf{Mb_p}}{\left\|\mathbf{Mb_p}\right\|} \tag{2.9}$$

$$\mathbf{M} = \mathbf{A}^\top \mathbf{A} \tag{2.10}$$

The test for convergence was implemented by taking dot product

$$\mathbf{b_{p+1}}^\top \mathbf{b_p} \approx 1 \tag{2.11}$$

The final eigenvector can be represented as

$$\mathbf{v_1} = \mathbf{b_{p+1}} \tag{2.12}$$

The eigenvalue corresponding to this eigenvector is usually the highest ((((((TODO))))). We keep repeating this power iteration k times.

We update $\mathbf{A} : \mathbf{A} - \sigma \mathbf{u_1} \mathbf{v_1}^\top$, where $\mathbf{Av_1} = \sigma_1 \mathbf{u_1}$
We then repeat the process from **??**

Then, the rank k approximation $\mathbf{A_k}$ can be computed by summing the previously obtained k singular values and vectors

$$\mathbf{A_k} = \sum_{i=1}^{k} \sigma_i \mathbf{u_1} \mathbf{v_1}^\top \tag{2.13}$$

$$\sigma_1 > \sigma_2 > \sigma_3 \ldots \sigma_k \tag{2.14}$$

2.4 Pseudocode

```
FUNCTION FINDEIG(MATRIX A)
    B = transpose(A)
    C = multiply(B,A)
    rand_vector = generate_random_vector()
    current_vector = rand_vector/norm(rand_vector)
    WHILE True
        prev_vector = current_vector
        current_vector = multiply(C,prev)
        current_vector = current_vector/norm(rand_vector)
        dot = dot_product(prev_vector, current_vector)
        IF ABSOLUTE(dot) > 0.999999999 THEN
            return current_vector
        ENDIF
    ENDWHILE
ENDFUNCTION

FUNCTION SVD(MATRIX A, INT k)
    B = A
    FOR i = 1 to k
```

```
                v = findEig(B)
                u = multiply(A, v)
                sigma = norm(u)
                u = u/sigma
                svd_one = sigma*multiply(u, transpose(v))
                B = B − svd_one
                RES += svd_one
        ENDFOR
        RETURN RES
ENDFUNCTION

A = INPUT_IMAGE_MATRIX()
B = SVD(A, k)
OUTPUT_IMAGE(B)
```

3 Comparisons

The algorithm implemented is lightweight, short, easy to understand and implement. The reason this algorithm was chosen is to demonstrate the power of SVD without the implementation of more heavy (although more efficient) algorithms.

This algorithm could be improved further by using RSVD (Randomized SVD). Which involves taking a random matrix and performing a series of computations, until we can do the SVD on a smaller matrix. This can drastically improve the speed if coupled with the current algorithm.

Other algorithms include:

- One-Sided Jacobi Method
- Two-Sided Jacobi Method

4 Results

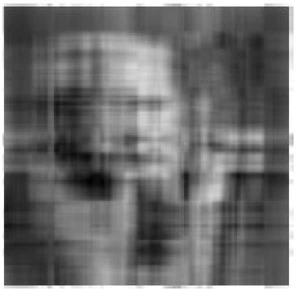

Fig. 2.1: Einstein 5

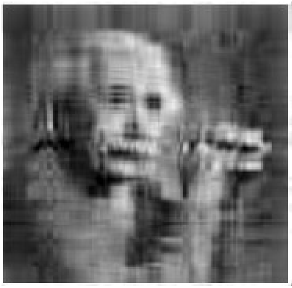

Fig. 2.2: Einstein 10

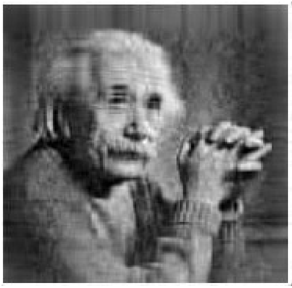

Fig. 2.3: Einstein 20

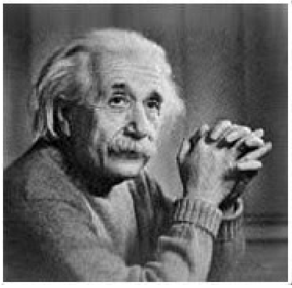

Fig. 2.4: Einstein 50

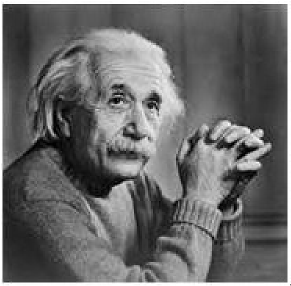

Fig. 2.5: Einstein 100

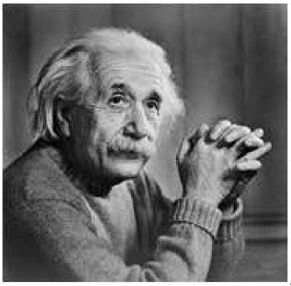

Fig. 2.6: Einstein 150

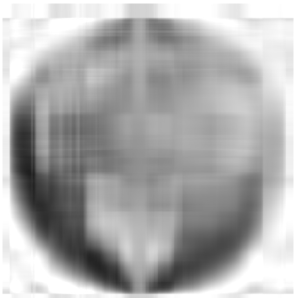

Fig. 2.7: Globe 5

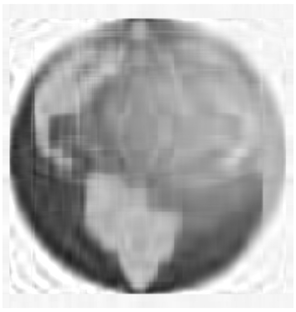

Fig. 2.8: Globe 10

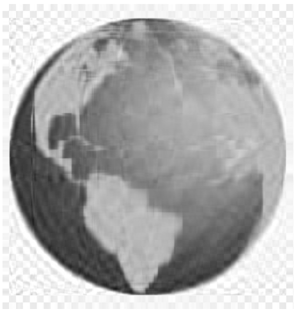

Fig. 2.9: Globe 20

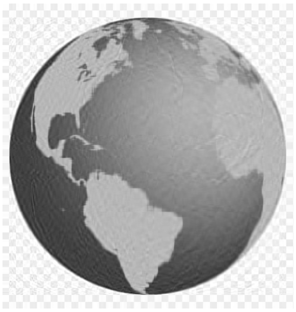

Fig. 2.10: Globe 50

Fig. 2.11: Globe 100

Fig. 2.12: Globe 150

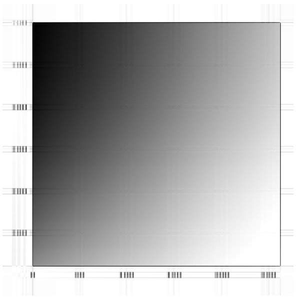
Fig. 2.13: Greyscale 5

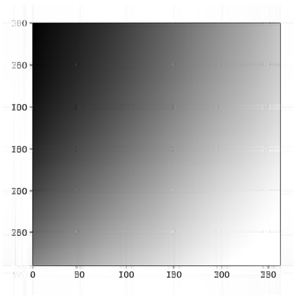
Fig. 2.14: Greyscale 10

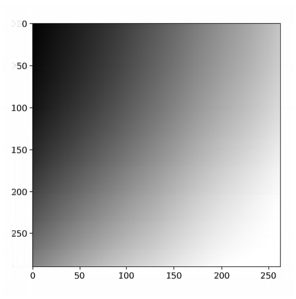
Fig. 2.15: Greyscale 20

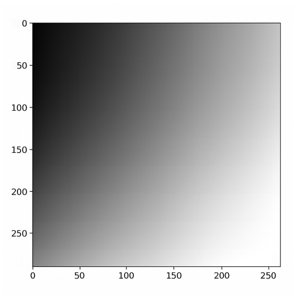
Fig. 2.16: Greyscale 50

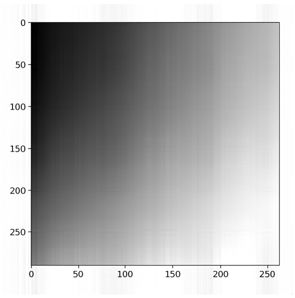

Fig. 2.17: Greyscale 100

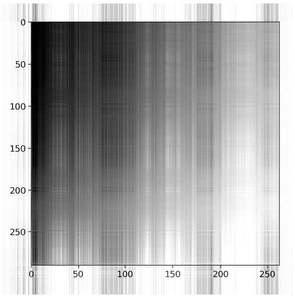

Fig. 2.18: Greyscale 150

5 Error analysis

K	Time taken	Frobenius norm $\|\|A - A_k\|\|$
5	0.176s	155153022976.000000
10	0.198s	3255.545166
20	0.402s	2134.984619
50	1.073s	899.152283
100	2.278s	170.30783
150	3.333s	127.909058

TABLE 2: Results for einstein.jpg

K	Time taken	Frobenius norm $\|\|A - A_k\|\|$
5	7.652s	26229.347656
10	15.322s	15058.915039
20	36.404s	10633.403320
50	80.325s	6188.093262
100	168.151s	3679.777832
150	254.437s	2506.446777

TABLE 2: Results for globe.jpg

K	Time taken	Frobenius norm $\|\|\mathbf{A} - \mathbf{A}_k\|\|$
5	15.122s	13883785.000000
10	26.271s	7185.126953
20	53.067s	3825.678711
50	137.599s	1216.309204
100	295.020s	4136.271484
150	417.317s	26552.162109

TABLE 2: Results for greyscale.jpg

6 Trade-offs and reflections on implementation choice

The implemented algorithm is basic in nature. It is lightweight and easy to understand. It scales linearly according to k, and cubic according to n, where n is image size for square image. Number of computations:- $O\left(kn^3\right)$

While the algorithm implemented is simple, and easy to understand, it clearly under-performs when it comes to efficiency and speed. Industry-standard algorithms process the image almost instantaneously.

There were also a few flaws in the implementation:

- **The rate limiting step is computing $\mathbf{A}^\top\mathbf{A}$**. This step alone takes $\approx 90\%$ of the time. Smarter methods need to be implemented.
- **Lack of parallelism**. It performs the computations on the given matrix sequentially, instead of simultaneously.
- **Errors due to random vector initialization.** The generated random vector might be orthogonal to the eigenvector with highest corresponding eigenvalue.
- **Hardcoding limits for convergence**. The code limits the convergence check upto 2000 iterations. In some cases higher number of iterations are needed. Hence there is a big compromise on accuracy at the cost of efficiency.