Singular Value Decomposition and Image Compression

Can I Compress an Image to Take Up Less Storage on My Laptop?

Laurence Palmer

June 2, 2021

1 Introduction

In this project, we attempt to answer the question: Using the SVD, can I compress one of my pictures to take up less storage on my device while keeping decent quality? Further, does the SVD that I code have better performance than the np.linalg.svd built in function? This question is significant, because it can allow me to save both storage on my computer and, potentially, money in my pocket (by avoiding DropBox premium when I share pictures). To actually compress the image, I utilize the classic Singular Value Decomposition and use a simple method to determine the optimal rank approximation to the original image. First, I begin with analyzing the Olivetti Faces data to get a sense of how to determine the rank of a decent compression. Then, I compress a png file on my computer to see if I can successfully decrease the amount of memory used.

The following sections outline the methods and theory, results, and issues in my project. The couple of references I use are included at the very end of the outline.

2 Methods and Theory

The main method that I utilize is the Singular Value Decomposition. As we know, the singular value decomposition for an $m \times n$ matrix A is given by $A = U\Sigma V^T$. First, its important to note that I only perform the SVD on $m \geq n$ arrays, but the algorithm would work for $m < n$ if the transpose is inputted instead. For my algorithm, the construction of each of these matrices arises from finding eigendata for the matrices AA^T (U) and A^TA (V), which are both symmetric positive semi-definite matrices. The symmetry of these matrices is a huge advantage, because the matrices U and V can be constructed from the eigenvectors of these matrices. By the properties of symmetric matrices, these eigenvectors are immediately orthogonal, and the final U and V are constructed so that they are unitary, i.e. $U^TU = UU^T = I, V^TV = VV^T = I$ and

$$AA^T = UD_{m,m}U^T \qquad\qquad A^TA = VD_{n,n}V^T$$

where D contains the eigenvalues of AA^T and A^TA. Another important characteristic of these matrices is that they have the same nonzero eigenvalues as explained by Theorem 9.26 in Burden and Faires. Σ is constructed as an $m \times n$ matrix with the singular values (positive square roots of the nonzero eigenvalues of A^TA) placed on the main diagonal in descending order. Consequently, the corresponding eigenvectors in U,V are also ordered from greatest to least. Since Σ is sometimes rectangular, situations may arise where there will be an $(m-n) \times n$ block of zeros underneath the $n \times n$ diagonal block of singular values. Diagrammatically, we have

$$\begin{bmatrix} \\ \mathbf{A}_{m,n} \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \mathbf{U}_{m,m} \\ \\ \end{bmatrix} \begin{bmatrix} \\ \mathbf{S}_{m,n} \\ \\ \end{bmatrix} \begin{bmatrix} \\ \mathbf{V}_{n,n} \\ \\ \end{bmatrix}^T$$

Finally, the properties of U, V and $A = U\Sigma V^T$ lead to another equation that I leverage in my algorithm in order to fix a sign mix up that was propagating through my algorithm.

$$\begin{aligned} A = U\Sigma V^T \implies AV &= U\Sigma V^T V \\ &= U\Sigma I \\ &= U\Sigma \\ \implies Av_i &= u_i \Sigma_{i,i} \end{aligned}$$

for $i = 1, \ldots, n$. If this equation does not hold for any i, then I multiply the column u_i by -1, which rectified the sign issue. Also note that the SVD is not a unique decomposition, so my SVD differs slightly from np.linalg.svd, but both decompositions are valid.

An additional method that I use within my SVD algorithm is the built in function np.linalg.qr. This is utilized to orthonormalize vectors in U with multiple 0 eigenvalues. Suppose that AA^T has eigenvalues $\lambda_i, i = 1, \ldots, k$ with eigenvectors v_i, and $\lambda_{k=1} = \lambda_k = 0$. The sets $\{v_1, \ldots, v_{k-2}, v_k\}, \{v_1, \ldots, v_{k-2}, v_{k-1}\}$ are orthogonal, but v_k is not orthogonal to v_{k-1}. Consequently, these vectors must be orthogonalized and normalized in order to include them in U. These situations are where I leverage the QR algorithm, replacing the non-orthogonal vectors with the columns of Q in $[v_{k-1}, v_k] = C = QR$, which form an orthonormal basis for the column space of C. This ensures that the entire set $\{v_1, \ldots, v_k\}$ will be orthogonal/orthonormal.

The value of the Singular Value Decomposition in image compression comes from the fact that it is possible to create a lower rank approximation to the original image while retaining the most important features. The most important features or the "true image" correspond to those singular values which are the greatest in magnitude, as specified in Burden and Faires. Since Σ stores the singular values in descending order, creating a lower rank approximation simply means truncating Σ from $m \times n$ to $r \times r$, where r is the chosen rank. Consequently, both U and V^T must be truncated as well. Essentially, an $m \times n$ matrix A with singular value decomposition $A = U_{m,m}\Sigma_{m,n}V^T_{n,n}$ has rank r approximation

$$A_r = U_{m,r}\Sigma_{r,r}V^T_{r,n}$$

so that U now has m rows and r columns, while V has r rows and n columns. Evidently, the resulting approximation A_r is still an $m \times n$ matrix, but the data storage savings are in the factorized form of A using the lower rank approximation. A originally needs mn amount of storage. With the lower rank SVD, U requires rm, S requires r, and V^T requires rn storage registers which amounts to $r(m + n + 1)$ storage registers total.

To determine a good rank approximation for the pictures I process, I use a simple percentage formula. For the Olivetti Faces data set, I sum the singular values for each of the respective faces, and find the rank r that satisfies

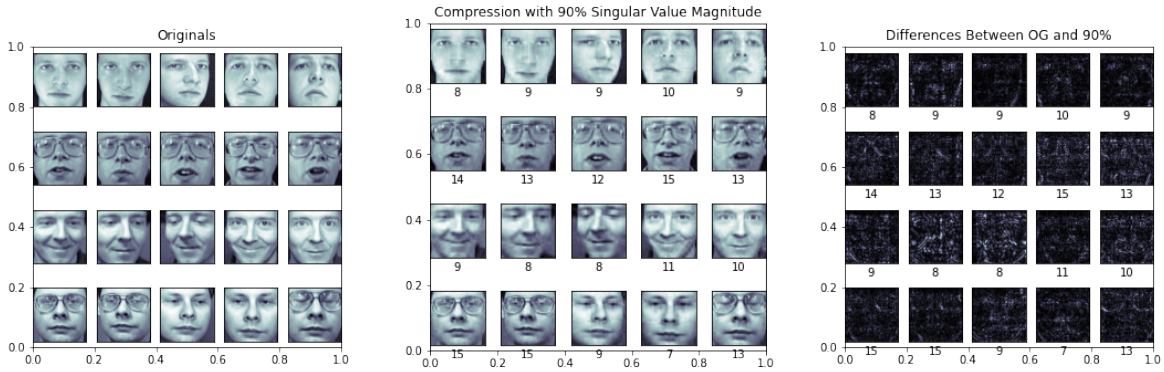
$$\frac{\sum_1^r s_i}{\sum_1^n s_i} < \epsilon \tag{1}$$

for $n = 64$ for the 64 different singular values (Olivetti Faces are 64×64) and where ϵ is the threshold. For our purposes, we will set $\epsilon = 0.9$

Additionally, I calculate the absolute differences between each different rank approximation and the original image to quantify how much information we lose by the compression. To test the performance of my algorithm against a benchmark, I also timed the performance of my algorithm against np.linalg.svd over 10 runs and compared the averages. This will be discussed further in the results section.

3 Results

To test the SVD, I fed a few random matrices into the algorithm to ensure that it was working properly. I checked to make sure that the decomposition that was obtained was able to be multiplied back to form the original matrix. After successfully coding the SVD, the algorithm was applied to 20 of the Olivetti faces, and the faces were reconstructed with the full factorization initially as a baseline check. Then, I began to determine the appropriate rank for the lower order approximation, with equation 1 with $\epsilon = 0.9$. The pictures below are the different rank constructions for each face, with the rank approximations used printed below each face.



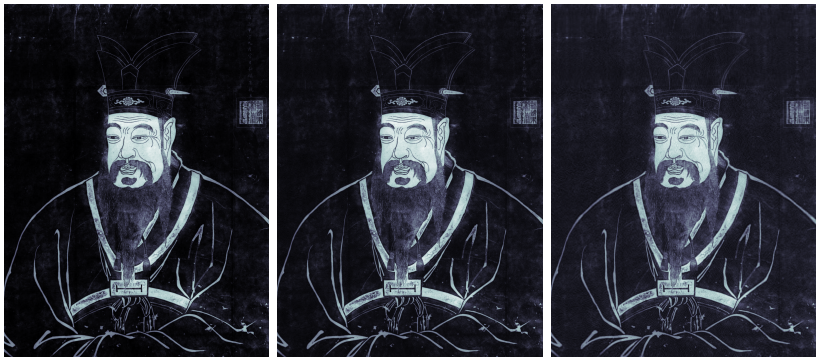
As we can see, there's not a very noticeable difference between the different rank reconstructions and the original images. This is because we have set the threshold relatively high at 90%. Intuitively, this implies that we are retaining approximately 90% of the information within each of the pictures. Below is a few tables of the numerical values of the absolute differences between the original images and their lower rank approximations. The first table is the sum of the absolute differences, and the second table is the sum of the absolute differences divided by the sum of the absolute values of the original picture. This should give a sense of what percentage of information is lost with the compression. The placement of the numbers on the grid corresponds with the appropriate picture.

69.08298839	69.94938629	71.96010573	66.92964003	71.2147339
68.66722216	73.90146147	68.30709066	70.86203547	73.87470562
67.14077926	73.41006735	65.04728133	67.1456491	68.26323379
67.44463651	68.97862078	71.31710154	71.46651785	66.61122453

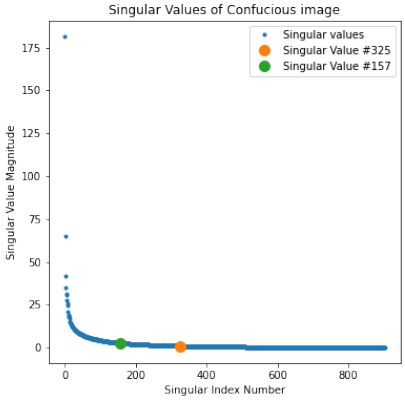
0.0264835	0.027419	0.02883012	0.02557729	0.02674898
0.02982167	0.03228373	0.02903239	0.03278756	0.0325428
0.03182071	0.03433566	0.03104086	0.02848726	0.02836297
0.03007875	0.03136983	0.02924095	0.02845758	0.03057091

As we can see, relative to the original image, we are barely deviating with our much lower level approximations! Granted, the measure is slightly flawed, but given the data savings in factorized form, if we are close to these percentages in loss, then this is excellent performance. Consider the first image in the Olivetti faces that I display. I am reconstructing it using a rank 8 approximation. This means that compared to the 4096 storage units I would need for the original, I can use $8(64 + 64 + 1) = 1,032$. This is a reduction of 74.8%! These measures and the aesthetics of the reconstructions, show that the method is working successfully.

Following the Olivetti faces, I applied my SVD algorithm to an image within my computer. The dimensions are 1200×903, which implies 1,083,600 storage registers are needed for the full picture. After the application of the SVD, the same process for determining a rank approximation was carried out for this picture. This is with the same threshold of $\epsilon = 0.9$ but I also tested the rank finding procedure for $\epsilon = 0.75$. The resulting ranks were 325 and 157 respectively. The original picture, along with the lower rank approximations are below. The far left is the original, the middle is the 90% rank approximation of rank 325, and the far right is the 75% rank approximation of rank 157.



Clearly, there is a significant loss of data between the original and the rank 157 approximation. There is clear degradation in the background and also the face is slightly blurred. Thus, the rank 325 approximation is the better approximation, and 90% is the better threshold. Let's take a look at the absolute differences between the original picture and the rank 325 approximation. Additionally, included is a graph plotting where the rank 325, 157 approximations lie on the range of singular values.



As we can see, this approximation only misses out on slight details in the outline of the image. However, compared to the original picture, the rank 325's storage load in factorized form is $325 \times (1200+903+1) = 683,800$ which is a reduction of $(1,083,000 - 683,800)/1,083,000 = 36.86\%$. Additionally, the sum of the absolute differences for the rank 325 approximation is 7957.62287543793, and this value divided by the sum of the absolute value of the original image is 0.060711492641626924. Clearly, the approximation does a pretty good job, and I would consider this successful. But ultimately, I was unable to store the factorized version of the image, so I was unable to obtain the savings in data storage on my desktop.

For fun, I decided to put my algorithm up against the np.linalg.svd package to see how much time it would take to find the decomposition for the 1200×903 image. I timed each program 10 times and some summary statistics are below

	their_times	my_time
count	10.000000	10.000000
mean	28.752281	0.146327
std	1.055571	0.005234
min	27.643972	0.137578
25%	28.231659	0.142843
50%	28.424527	0.145950
75%	28.880119	0.148746
max	31.243944	0.154283

The performance is actually much better than I expected given how much the np.linalg.svd is optimized. Being within an order of 2 is certainly acceptable, and thus, I consider this to be a success.

4 Conclusions

In conclusion, I successfully created an SVD algorithm that could factorize and reconstruct an image on my computer. I was able to find lower rank approximations to the original image that gave produced an acceptable image. Additionally, the algorithm performed well and was much faster than expected. However, I was unable to store the image in factorized form on my computer, so the savings in data storage was not obtained in the end. Looking back, I would have attempted to utilize more of my self coded algorithms, such as the Power Method, in order to make the SVD algorithm my work as much as possible. However, as outlined below, there were some difficulties with the algorithms I wrote, so I had to go with the built-in packages. But in the end, I still think the project was successful, and I am elated that I was able to code a working SVD.

4.1 Alternative Eigenvalue Method

Originally, I attempted to implement the power and inverse power methods with weilandt deflation to determine the eigenvalues and eigenvectors. However, there was an issue of singular matrices when performing the inverse power method solving the system $(A - qI)y^{(m)} = x^{(m-1)}$. Thus, I used the np.linalg.eig package to compute the eigenvalues and eigenvectors.

4.2 Corner Case

There's an interesting corner case that I encountered with my SVD algorithm. The simple 3×2 matrix

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & -1 \end{bmatrix}$$

failed to be properly factorized using my model. After some investigation it was apparent that np.linalg.eig was producing eigenvectors for U that were different from those I calculated by hand. The vectors computed by the package were $[1, 0, 0]^T, [0, 0, 1]^T, [0, 1, 0]^T$, while I used $[1, 0, 1], [1, 0, -1]^T, [0, 1, 0]^T$ for my manual computation. While, all four vectors are bases for the null space of $A - \lambda I$, the basis constructed by np.linalg.eig produces the wrong SVD factoriztion when coupled with the V^T vectors it also calculated. However, the SVD is not necessarily unique, so an alternative choice of V^T could produce a valid SVD factorization. Unfortunately, my algorithm was unable to compute this using the np.linalg.eig package.

5 References

Burden, Richard L., et al. Numerical Analysis. 10th ed., Cengage Learning, 2016.

Cline and Dhillon, "Computation of the Singular Value Decomposition"
https://www.cs.utexas.edu/users/inderjit/public$_{p}apers/HLA_{S}VD.pdf$