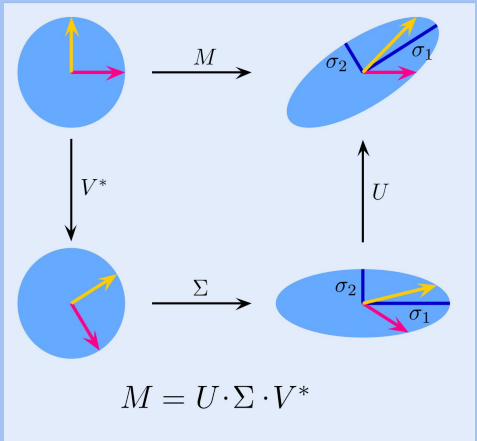
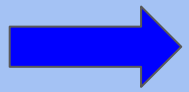
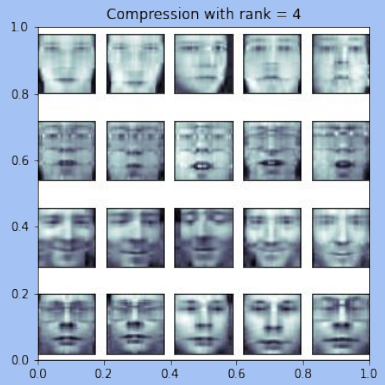
SVD and Image Compression







Can I Compress an Image on my Laptop?

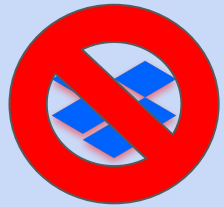
Laurence Palmer, 575802-4

Motivation and Problem

- Image compression is data compression for images to reduce storage or transmission costs.
- Images take up a lot of space in a computer's memory and in applications like DropBox. Can I use my own SVD to factorize and image and convert that into storage savings on my computer? Further, how does my SVD stack up to the np.linalg.svd package?
    - Recently, I tried to upload some photos to DropBox to share with friends, and after a few hundred photos, my free storage began to run out!
    - Images are simply just matrices that store information for the pixels
    - I should be able to reconstruct an image using a lower rank factored form obtained from the SVD I code.
- Mathematically, our goal is the get the SVD factorization of an image, and obtain a lower rank approximation that still gives a good approximation to the original image. If A (m x n, m >= n assumed) represents our image, then mathematically our problem is

$$A = U\Sigma V^T \implies A \approx U_r \Sigma_r V_r^T$$

- At a high level, the SVD essentially creates these matrices using the eigendata from the matrices AA^T and A^TA. U (m x m) stores the eigenvectors of the former and V (n x n) the latter, and since these are symmetric, semi-positive definite matrices, U,V will be unitary when the vectors are normalized. Sigma stores the singular values in descending order which are the square roots of the nonzero eigenvalues of A^TA and AA^T. Note that A^TA and AA^T have the same nonzero eigenvalues
- By finding r, we can get a good resolution approximation to the original image and save the storage units by storing the image in factorized form. The goal is to toe the line between losing too much resolution and not having a significant amount of data storage
- This solution is important because it can save memory on your computer, which has obvious cost savings especially when you use services like DropBox.
    - We will be using a simple percentage threshold method to figure out what rank approximation works best



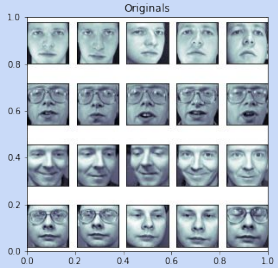
Confucious
Final Image I compress

Methods and Validation

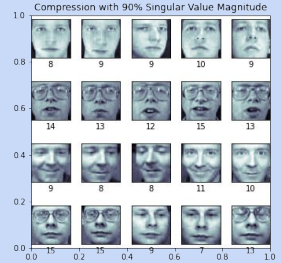
- As mentioned, we solve the image compression problems by using the SVD. The 104 methods that we use are numerical methods for finding eigenvalues and eigenvectors, which are included in the general Singular Value Decomposition. The factorized form of the image with a lower rank approximation that we get from the SVD will reap the storage savings we seek. Note that all images I will be compressing have numbers of rows that are greater than or equal to the number of columns ie (m >=n).
    - I implement the SVD by creating the matrices and finding the eigenvalues and eigenvectors of A^TA and AA^T. Then, I sort the eigenvalues and store them in descending order in the Sigma matrix, and order U and V so that the corresponding eigenvectors are also in descending order. Next, there is a possibility that U (since m>=n, U is m x m) has 0 eigenvalues. These vectors will be orthogonal to the rest of the eigenvectors respectively, but not orthogonal to each other. To rectify this, I apply the QR algorithm to ensure that these vectors are orthogonal to each other. Finally, to fix a sign issue that was propagating through my algorithm I use the relationship, specified in Burden and Faires, that Av_i = u_i s_i, where v_i are the columns of V, u_i are the columns of U, s_i are the singular values, and i = 1,..., n, since m>=n. If this relationship does not hold, then I multiply the u_i vector by -1, and put it back into U. This rectified the sign issue.
- Theoretically, the features of the “true image” correspond to those singular values that have the largest magnitude, while “noise” corresponds to the smaller singular values, as specified in Burden and Faires. The ordering of the singular values in the Sigma Matrix for the SVD makes this easy, since we can just take the first r rows and columns of Sigma as we like, since the highest singular are stored at the top. Then we can take the first r columns of U, and the r rows of V^T, and we will get the lower rank approximation.
    - This is how I construct the lower rank approximations to the images.
    - For an m x n image, the factorized, lower rank approximation takes only r(m+n+1) storage units where r is the chosen rank, compared to mn units (Burden and Faires).
- I validate the SVD by first testing it on random matrices to ensure that it is working properly. Then, I will test its image compression capabilities on 20 of the Olivetti faces and try my rank finding method described below. Finally, I apply the SVD algorithm to a 1200 x 903 image on my computer to see whether the method and the rank finding method works for an image not within a well curated data set.
    - For the Olivetti data set, I use a percentage threshold method to find a lower rank approximation for the images. Essentially, I summed the singular values of each of the 20 faces, and found the rank, r , where $\frac{\sum_1^r s_i}{\sum_1^n s_i} < \epsilon$ and n = 64 (since the Olivetti faces are 64x64) and epsilon = 0.9.
    - For the final image, I did the same procedure with the thresholds 0.9 and 0.75
- I evaluate the performance of the image compression by observing the absolute differences between the original image and the lower rank approximation that I obtain. I also compute the reduction in the amount of storage units needed to store “Confucious” on my computer with the SVD factorization. I will evaluate the performance of my algorithm against the np.linalg.svd package on 10 timed runs, to see whether it runs faster.

Results

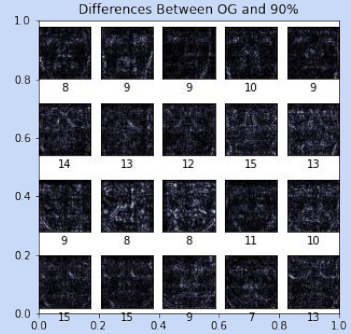
```
array([ 8.,  9.,  9., 10.,  9., 14., 13., 12., 15., 13.,  9.,  8.,  8.,
       11., 10., 15., 15.,  9.,  7., 13.])
```



I was able code successfully code my version of the SVD algorithm. It was able to factorize the example matrix I gave it and reconstruct the images. The pictures on the right are the product of my SVD algorithm applied to the Olivetti faces data set. With the averaged percentage threshold method for finding the rank set at 0.9, I found the ranks in the array above satisfied this requirement for the 20 Olivetti Faces. For each of these respective ranks, I computed the absolute differences between the original face image and the lower rank reconstruction, and the percentage difference of this value divided by the sum of the absolute values of the original image. These values are included in the tables second to the bottom right, with the bottom table being the percentages (their placement corresponds to the image in the grids on the right). Clearly, the algorithm did a good job of approximating as seen aesthetically on the from the images on the right, and the values of the percentages in the table, which roughly quantify how much information we are losing from the approximation.



As for Confucious (1200 x 903), for the percentage thresholds of 0.9 and 0.75, I found that the appropriate rank approximations were 325 and 157 respectively. . The rank 157 approximation was not very accurate so I went with the rank 325 approximation instead. With this rank 325 approximation, using the formula specified previously of r(m+n+1), there was a 36.86% reduction in storage in factorized form. The sum of the absolute differences between the original and the rank 325 approximation was 7957.62287, and the percentage of this absolute difference divided by the sum of the absolute values of the original image was 0.060711492641626924. Thus, I was able to successfully factorize and decrease the storage load of the image. However, these savings were not realized on my computer, since I was unable to export the image in factorized form. For the performance of my algorithm compared to np.linalg.svd, I found that my algorithm performed okay over 10 runs where each algorithm factorized Confucious. Some summary statistics are provided in the bottom left. This was not surprising as the np.linalg.svd package is most assuredly better than what I wrote. However, I still consider the algorithm a success, since it is within an order of magnitude within the performance time of np.linalg.svd.

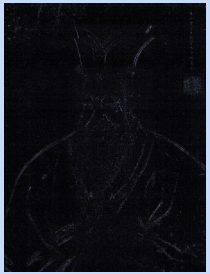


	their_times	my_time
count	10.000000	10.000000
mean	28.752281	0.146327
std	1.055571	0.005234
min	27.643972	0.137578
25%	28.231659	0.142843
50%	28.424527	0.145950
75%	28.880119	0.148746
max	31.243944	0.154283

Confucious

Rank 325 Approx

Abs differences



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