Lab 1

Image Compression (SVD)

Lab Objective: *Learn how to compute the compact SVD and explore the SVD as a method of image compression.*

The theoretical use of the *Singular Value Decomposition* or *SVD* has long been appreciated. In fact, the idea of a canonical way of decomposing a matrix was so alluring that the SVD was independently discovered by at least four people through use of both integral equations and systems of linear equations.

However, it wasn't until Erhard Schmidt showed how the SVD could be a computational tool for providing low-rank approximations that the practical applications became apparent. Since Schmidt's work, further devolpments have confirmed the importance of the SVD in both computational and theoretical applications.

Computing the SVD

The Singular Value Decomposition separates a $m \times n$ matrix A into two unitary matrices and a diagonal matrix. This takes the form of

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

where U and V are square and unitary of sizes m and n respectively, and Σ is diagonal and of size $m \times n$. The values of Σ are the *singular values* of A, and are the square root of the eigenvalues of $A^H A$. Commonly the singular values are listed in decreasing order. Thus we get

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ are the singular values of A.

The *compact SVD* is where the r column vectors of U and the r row vectors of V are calculated, cooresponding to the r nonzero singular values. We lose the decomposition of the nullspace of A, but can still regain the full matrix A. It takes the form $A = U_1\Sigma_1 V_1^H$ where U_1 is $m \times r$, Σ_1 is $r \times r$ and diagonal, and V_1^H is $r \times n$.

The *truncated SVD* is similar to the compact SVD, but instead of taking all the nonzero singular values, we only take the k largest. While this saves space, it means

that we cannot recover the whole matrix. Instead we have an approximation that shares the largest k singular values. We end up with $\widehat{A} = U_k \Sigma_k V_k^H$ where $\widehat{A}$ is an approximation of A, U_k is $m \times k$, Σ is $k \times k$ and diagonal, and V_k^H is $k \times n$.

The only difference in computing the compact SVD verses the truncated SVD is the number of singular values we keep. If we keep all the nonzero singular values, it is the compact SVD. If we only keep some of the nonzero singular values, then it is the truncated SVD.

To compute the compact or truncated SVD:

- The singular values of A are the square root of the eigenvalues of $A^H A$ and are sorted in descending order. These are the diagonal of Σ. For the compact SVD, keep all nonzero singular values. For the truncated SVD, keep the first k.
- The columns of V are the eigenvectors of $A^H A$, where the ith column matches the ith singular value.
- The columns of U are $U_i = \frac{1}{\sigma_i} A V_i$.

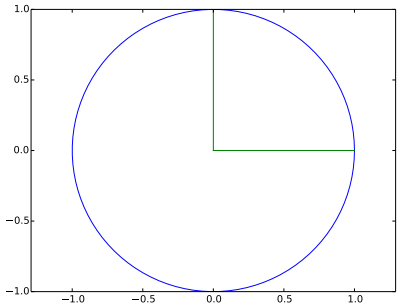
Problem 1. Write a function `truncated_svd` that accepts a matrix A and an optional integer `k = None`. If k is `None`, calculate the compact SVD. If k is an integer, calculate the truncated SVD. Since the only difference between these two processes is the number of singular values we keep, we only need to write one function.

1. Find the eigenvalues and eigenvectors of $A^H A$.
2. Find the singular values of A.
3. Sort the singular values and only keep the greatest k.
4. Calculate V.
5. Calculate U.

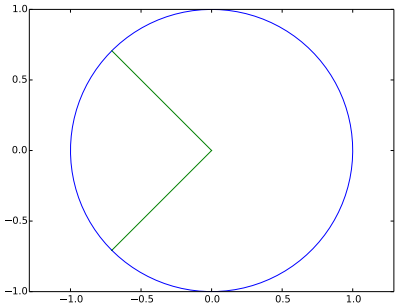
Check your function by calculating the compact SVD and seeing if $U_1 \Sigma_1 V_1^H = A$ using `np.allclose()`.

Hint: While calculating the SVD, you will need to sort the eigenvalues while keeping track of their associated eigenvectors. To accomplish this, use `np.argsort` to generate a mask that can be used to order the eigenvalues. The same mask can be used to order the columns of a matrix. Also, an efficient and concise way to keep the nonzero eigenvalues is to use fancy indexing. See the NumPy and SciPy lab for more information on `np.argsort` and fancy indexing.

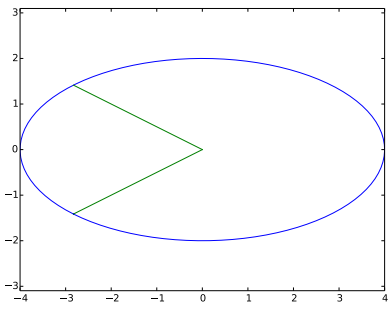
If A is full rank and square, it has n nonzero singular values and this process gives us the full SVD. If A is not full rank or not square, we can compute the first r



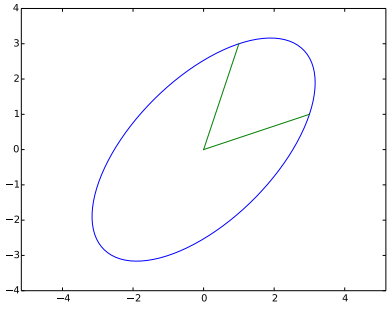
(a) The unit circle, S, with two unit vectors.



(b) $V^H S$



(c) $\Sigma V^H S$



(d) $U\Sigma V^H S$

Figure 1.1: Each step in transforming the unit circle and two unit vectors using the matrix A.

columns of U and V, where r is the number of nonzero singular values, in this way. To find the rest of the SVD, we can use Gram-Schmidt orthonormalization.

More specifically, let r be the number of nonzero singular values. If $r < n$ then we only have r eigenvectors to fill V, which is a $n \times n$ matrix. We find the remaining columns of V by using Gram-Schmidt orthonormalization to finish the basis for n-space. Similarly, if $r < m$ then we only have r columns of U so we use Gram-Schmidt orthonormalization to finish the basis for m-space. The compact SVD does not have this problem because U_1 and V_1 only have r columns, the remaining columns are the decomposition of the null spcae of the matrix, and are not included. In this lab we calculate the compact SVD for simplicity.

Visualizing the SVD

Recall that a matrix is one way to express a linear transformation. A $m \times n$ matrix sends points from $\mathbb{R}^n$ to $\mathbb{R}^m$. These transformations are a mix of rotations and rescalings. The SVD decomposes these transformations into their individual parts. V^H is a rotation, Σ a rescaling along the principal axes, and U is another rotation.

Problem 2. If S are the points on the unit circle, $U\Sigma V^H S$ is equivalent to AS where $U\Sigma V^H$ is the SVD of A. Using either your SVD from problem 1 or from scipy.linalg.svd, plot each part of the transformation of the matrix

$$A = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}. \tag{1.1}$$

Your solution should show S, $V^H S$, $\Sigma V^H S$, and $U\Sigma V^H S$. Since A is square and full rank, the compact SVD is equal to the full SVD. You solution should look like Figure 1.1

The `circle.npz` dataset contains a set of points on the unit circle and a set of unit vectors. You can access the circle by loading the `"circle"` file. You can access the unit vectors by loading the `"unit_vectors"` file. These points are stored as a numpy array of points in $\mathbb{R}^2$ with the first row being all the x-coordinates and the second row being all the y-coordinates.

Image Data Compression

In this lab, we explore how the SVD can be used to compress image data. Recall that an image is simply a matrix where each position is the color value for the pixel in that position. The SVD lets us choose how much information to keep, and what information is most important. Larger eigenvalues correspond to columns of U and V that contain more information, while smaller eigenvalues correspond to less important columns. This idea is used in many areas of applied mathematics including signal processing, statistics, semantic indexing (search engines), and control theory.

Low rank data storage

If the rank of a given matrix is significantly smaller than its dimensions, the compact SVD offers a way to store A with less memory. Without the SVD, an $m \times n$ matrix requires storing mn values. By decomposing the original matrix into the compact SVD, U_1, Σ_1 and V_1 together require $mr + r + nr$ values. Thus if r is much smaller than both m and n, we can obtain considerable efficiency. For example, suppose $m = 100$, $n = 200$ and $r = 20$. Then the original matrix would require storing $20,000$ values whereas the compact SVD only requires storing 6020 values.

Low rank approximation

The truncated SVD is useful in approxiamating a matrix with one of lower rank. By only keeping the first k singular values, we can create an approximation $\widehat{A} = U_k \Sigma_k V_k^H$.

The `scipy.linalg` module has a convenient method to calculate the SVD of a given matrix. We can use this method to create a lower-rank approximation of a given matrix. Execute the following code.

```
>>> import numpy as np
>>> import scipy.linalg as la
>>> A = np.array([[1,1,3,4], [5,4,3,7], [9,10,10,12], [13,14,15,16], ↩
    [17,18,19,20]])
>>> U,s,Vh = la.svd(A, full_matrices=False)
```

In that last line of code, we included the keyword argument `full_matrices=False` to calculate the compact SVD rather than the full SVD. The arrays `U` and `Vh` correspond to the matrices U_1 and V_1^H discussed earlier in the lab. The array `s` simply gives the nonzero singular values of the matrix `A`, and we can find the rank of `A` by inspecting the number of entries in `s` (in this example, we have a rank 4 matrix).

Next, we calculate a rank 3 approximation. We take the first three singular values and the first three columns of `U` and first three rows of `Vh`.

We omit the last singular value from the calculation along with the last column of `U` and last row of `Vh`.

```
>>> S = np.diag(s[:3])
>>> Ahat = U[:,:3].dot(S).dot(Vh[:3,:])
>>> la.norm(A-Ahat)
```

Note that $\widehat{A}$ is "close" to the original matrix A, but that its rank is 3 instead of 4. More precisely, $\widehat{A}$ is the best rank 3 approximation of A with respect to both the induced 2-norm and the Frobenius norm.

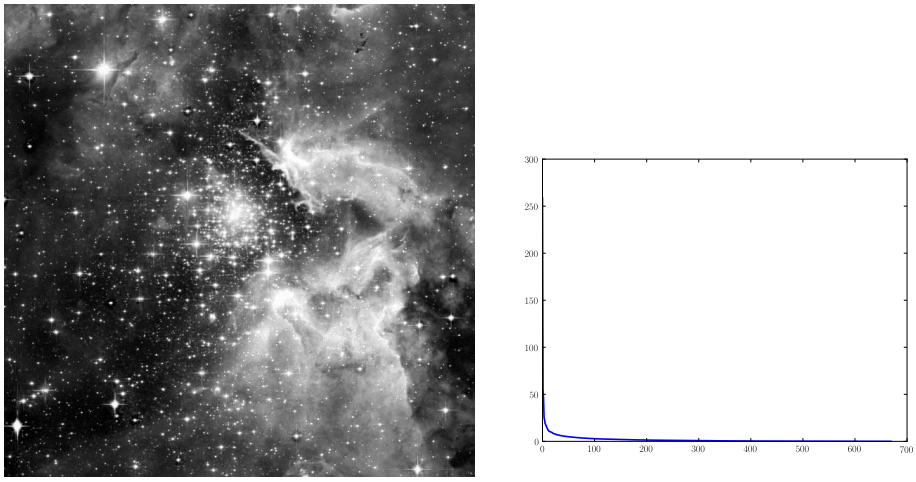
Problem 3. Write a function `svd_approx` that takes as input a matrix A and a positive integer k and returns the best rank k approximation to A with respect to the induced 2-norm.

Application to Imaging

Sometimes there is not enough available bandwidth to transmit a full resolution photograph. You aim to reduce the amount of data that needs to be transmitted from a remote location such that loss of image detail is minimal, but the amount of data that needs to be sent has reduced as much as possible. In Figure 1.2 we present an image and a plot of its singular values. Matrix rank is on the x-axis and the singular values are on the y-axis. Note that the SVD orders the singular values from greatest to least. The more singular values we keep, the closer the approximation but the more data we have to store. By looking at the graph in Figure 1.2b we can have a rough idea of how many singular values we need to preserve to have a good approximation of A. The matrix rank of the image is 670. However, we could easily approximate the image using only the first half of the singular values as the plot shows that all the values after are miniscule.

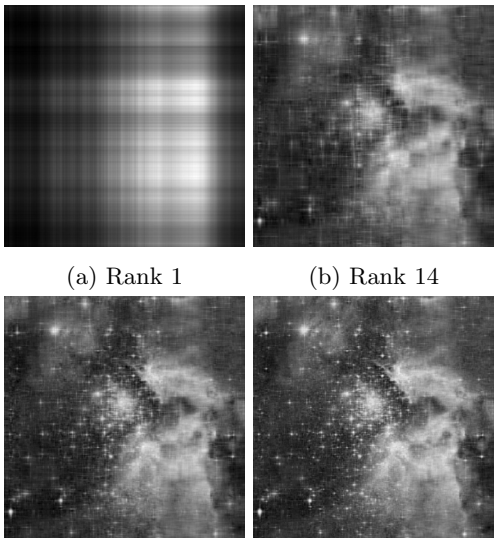
In Figure 1.3, we can see different rank approximations of the image in Figure 1.2.

To read in an image, convert it to black and white, and show it, use the code below. Then use your function from Problem 3 to calculate an approximation.



(a) NGC 3603 (Hubble Space Telescope). (b) Singular values from greatest to smallest.

Figure 1.2: An image and its singular values.



(a) Rank 1 (b) Rank 14

(c) Rank 27 (d) Rank 40

Figure 1.3: Different rank approximations for SVD based compression. Notice that higher rank is needed to resolve finer detail.

```
>>> import matplotlib.pyplot as plt
>>> from matplotlib import cm
>>> X = plt.imread('hubble_image.jpg')[:,:,0].astype(float)
>>> X.nbytes     #number of bytes needed to store X
>>> plt.imshow(X, cmap=cm.gray)
>>> plt.show()
```

Recall that the error between the best rank s approximation $\widehat{A_s}$ to A with respect

to the induced 2-norm is given by

$$\|A - \widehat{A}_s\|_2 = \sigma_{s+1},$$

where σ_{s+1} is the $(s+1)$-th singular value of A.

Problem 4. Using your `svd_approx` function from Problem 3, write a function `lowest_rank_approx` that takes as input a matrix A and a positive number e and returns the lowest rank approximation of A with error less than e (with respect to the induced 2-norm).

Problem 5. Using your `svd_approx` function from Problem 3, write a function `compress_img` that accepts two parameters `filename` and `k`. Your function should plot the original image and the best rank k approximation of the original image. While your `svd_approx` function worked for grayscale images, your `compress_img` function should work on color images. Your output should be similar to Figure 1.4.

At times, `plt.imshow` does not behave as expected when being passed RGB values between 0 and 255. It behaves much better when being passed numbers between 0 and 1. Additionally, since the SVD provides an approximation, it is possible that the SVD will generate values slightly outside the valid range of RGB values. To remedy this, use fancy indexing as discussed in the NumPy and SciPy lab.

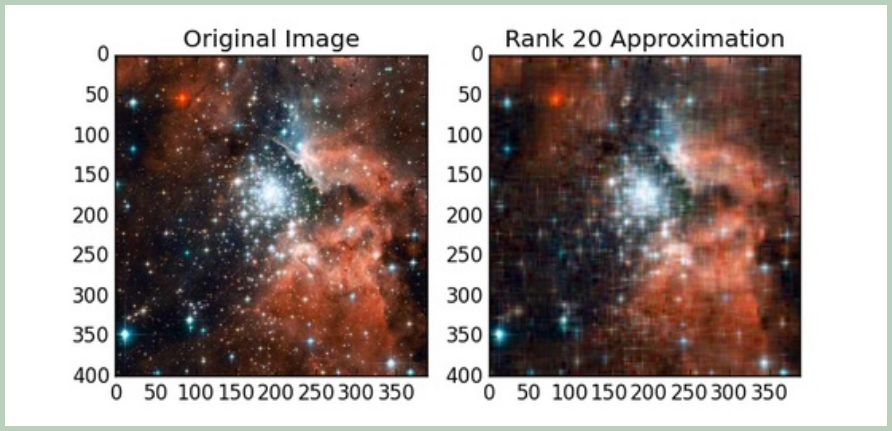


Figure 1.4: Correct output for the best rank 20 approximation.