Lab 10

The SVD and Image Compression

Lab Objective: *Learn how to compute the compact SVD. Explore the SVD as a method of matrix approximation, and use it to perform image compression.*

The *Singular Value Decomposition* or *SVD* is a matrix decomposition that is widely used in both theoretical and applied mathematics. Originally discovered by theoretical mathematicians, it is a canonical way to decompose a matrix. Its practical use became apparent later on when Erhard Schmidt showed that the SVD could be a computational tool for providing low-rank matrix approximations. Modern developments continue to confirm the importance of the SVD in both computational and theoretical applications.

Computing the SVD

The Singular Value Decomposition decomposes an $m \times n$ matrix A into the form

$$A = U\Sigma V^{\mathsf{H}}$$

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 along the diagonal of Σ are called the *singular values* of A. These are also the square roots of the eigenvalues of $A^{\mathsf{H}}A$. Commonly the singular values are listed in decreasing order. Thus we have

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

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

If A is of rank r, then A has exactly r nonzero singular values. The first r columns of U span the range of A, and the last $n - r$ columns span the null space of A^{H}. Likewise, the first r columns of V span the range of A^{H} and the last $m - r$ span the null space of A.

For a more in-depth definition and proof that the SVD exists for every matrix, refer to the section on the SVD in the text. Here we will focus on computing the SVD.

First, we define two modifications of the regular SVD. In the *compact SVD*, we only keep the r nonzero singular values. Only r column vectors of U and r row

vectors of V^H, corresponding to the r singular values, are calculated. The compact SVD takes the form $A = U_r \Sigma_r V_r^\mathsf{H}$ where U_r is $m \times r$, Σ_r is $r \times r$ and diagonal, and V_r^H is $r \times n$. Although we drop the decompositions of the nullspaces, by calculating $U_1 \Sigma_1 V_1^\mathsf{H}$ we can still recover the full matrix A.

The *truncated SVD* is similar to the compact SVD, but instead of keeping all the nonzero singular values, we only keep the k largest. While this saves space, it means that we cannot recover the whole matrix. Instead we end up with $\widehat{A}_k = U_k \Sigma_k V_k^\mathsf{H}$ where $\widehat{A}_k$ is a rank k 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 components of the compact or truncated SVD can be calculated as follows:

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

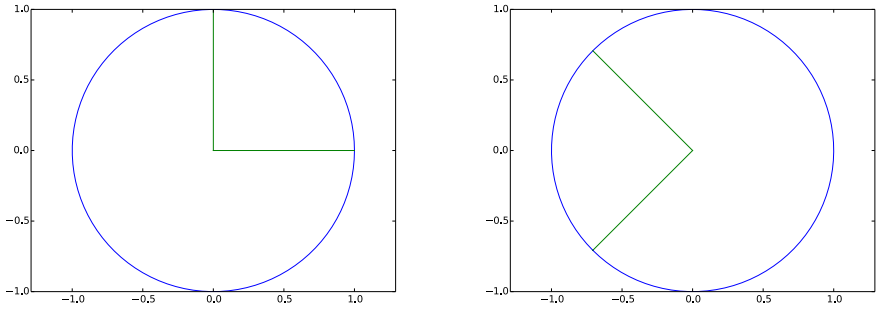
Problem 1. Write a function `truncated_svd` that accepts a matrix A and an optional integer `k = None`. If `k` is `None`, calculate the compact SVD. If `k` is an integer, calculate the truncated SVD, keeping only the `k` largest singular values. (Note: if there are fewer than `k` nonzero singular values, the truncated SVD will come out the same as the compact SVD.) Since the only difference between these two processes is the number of singular values we keep, we only need to write one function.

Here's an outline to follow:

1. Find the eigenvalues and eigenvectors of $A^\mathsf{H}A$.
2. Find the singular values of A. Keep only the greatest `k`, and discard any that are equal to zero.
3. Calculate V.
4. Calculate U.

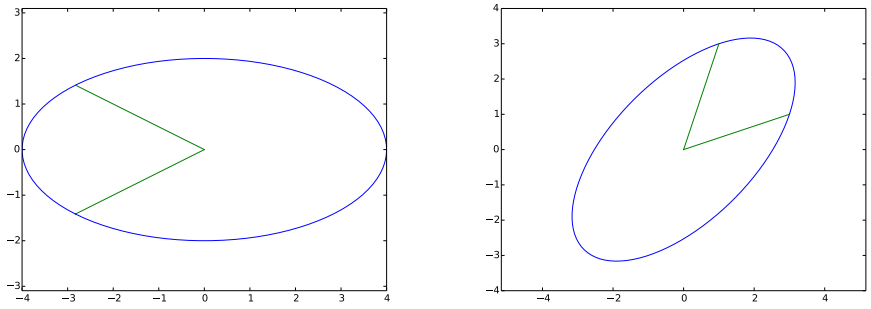
Return U, the diagonal of Σ, and V. Check your function by calculating the compact SVD and seeing if $U\Sigma V^\mathsf{H} = A$ using `np.allclose()`.

Hint: When calculating the SVD, you will need to sort the eigenvalues while keeping track of their associated eigenvectors. Consider using the function `np.argsort` for keeping the eigenvalues and eigenvectors in the same order while sorting.



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

(b) $V^{\mathsf{H}}S$



(c) $\Sigma V^{\mathsf{H}}S$

(d) $U\Sigma V^{\mathsf{H}}S$

Figure 10.1: Each step in transforming the unit circle and two unit vectors using the matrix A.

> **Note**
>
> In practice, calculating $A^{\mathsf{H}}A$ in order to find its eigenvalues is unstable. We use this method here because it is mathematically the simplest. However, industrial SVD solvers use different methods that avoid computing $A^{\mathsf{H}}A$.

Visualizing the SVD

Recall that a matrix is a way to express a linear transformation. An $m \times n$ matrix defines a linear transformation that sends points from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Intuitively, the SVD can be thought of as breaking a linear transformation into more basic steps. The SVD decomposes a given matrix into two rotations and a scaling. V^{H} represents a rotation, Σ represents a rescaling along the principal axes, and U represents another rotation.

Problem 2. In this problem we will use the SVD to visualize how the matrix

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

acts on points in $\mathbb{R}^2$. Given a set of points S in $\mathbb{R}^2$, we can calculate the transformation AS in steps by using the SVD $A = U\Sigma V^{\mathsf{H}}$.

Specifically, let S be a set of points on the unit circle. To generate the x- and y-coordinates of S, recall the equation for the unit circle in polar coordinates:

$$x = \cos(\theta) \qquad\qquad y = \sin(\theta),$$

where $\theta \in [0, 2\pi]$.

Plot four separate subplots to demonstrate each step of the transformation, plotting S, $V^{\mathsf{H}}S$, $\Sigma V^{\mathsf{H}}S$, then $U\Sigma V^{\mathsf{H}}$. Do the same for the standard basis vectors $\mathbf{e}_1 = [1, 0]^{\mathsf{T}}$ and $\mathbf{e}_2 = [0, 1]^{\mathsf{T}}$. Your solution should look similar to Figure 10.1.
(Hint: Force the plot to use the same scale on each of the axes with `plt.axis("equal")`. Otherwise, the circle will appear elliptical.)

The SVD and Data Compression

We now turn to computational uses of the SVD. We will explore how the SVD is useful for matrix approximations, and use it to compress images.

Low-Rank Matrix Approximation

If the rank r of a matrix A is significantly smaller than its dimensions, the compact SVD offers a way to store A with less memory. Storing an $m \times n$ matrix requires storing mn values. By decomposing the original matrix into the compact SVD, U_r, Σ_r and V_r together require $mr + r + nr$ values. This is an efficient storage method if r is much smaller than both m and n. 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.

The truncated SVD allows even greater efficiency. By only keeping the first k singular values, we can create an approximation $\widehat{A}_k = U_k\Sigma_k V_k^{\mathsf{H}}$. This requires storing only $mk + k + nk$ values. As we make k small, we eventually require very little storage. This comes at the cost of losing information from the original matrix.

The beauty of the SVD is that it makes it easy to only keep the information that is most important. Larger singular values correspond to columns of U and V that contain more information, so dropping the smallest singular values retains as much information as possible. In mathematical terms, given a matrix A and its rank-k truncated SVD approximation $\widehat{A}_k = U_k\Sigma_k V_k^{\mathsf{H}}$, the matrix $\widehat{A}_k$ is the *best rank k approximation* to A (with respect to the induced 2-norm and Frobenius norm). This

is a very significant concept in applied mathematics, appearing in areas including signal processing, statistics, semantic indexing, and control theory.

Implementation

We can use SciPy's linear algebra module to create low-rank SVD approximations or a given matrix. The code below computes the SVD of `A`.

```
>>> 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 the 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_r and V_r^{H} discussed earlier. The array `s` 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` (here we have a rank 4 matrix).

Next, we calculate a rank 3 approximation. We take the first three singular values, 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.

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 and Frobenius norm). Use `scipy.linalg.svd`.

Error of Low-Rank Approximations

Recall that the error between the best rank s approximation $\widehat{A_s}$ of 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. (See the proof of the Schmidt-Eckard-Young-Mirsky theorem in the text).

This offers a way to approximate a matrix within an error tolerance: choose the truncated SVD approximation such that the largest discarded singular value is less than the error tolerance.

Problem 4. Using `scipy.linalg.svd`, 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). You should only calculate the SVD once.

Application to Image Compression

Sometimes there is not enough available bandwidth to transmit a full resolution photograph. Suppose you need to transmit an image from a remote location. You might aim to reduce the amount of data being sent, while also minimizing the loss of detail in the image.

This can be done using the SVD. An image is just a matrix of pixel values, which means it has a singular value decomposition. Computing and sending a low-rank SVD approximation of the image can considerably reduce the amount of data sent, while retaining a high level of image detail. Additionally, successive levels of detail can be sent after the inital low-rank approximation by sending additional singular values and their corresponding columns of V and U.

Examining the singular values of an image gives us an idea of how low-rank the approximation can be. Figure 10.2 presents an image and a log plot of its singular values from greatest to least. The plot in 10.2b is typical for a photograph—the singular values start out large but drop off rapidly. In this rank 670 image, 624 of the singular values are 50 or more times smaller than the largest singular value. By discarding these relatively small singular values, we can retain all but the finest image details, while storing only a rank 46 image! This is a huge reduction in data size.

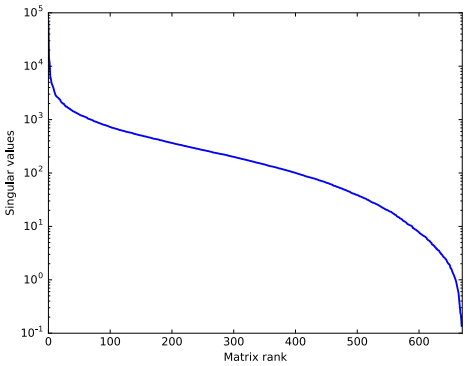
Figure 10.3 shows several low-rank approximations of the image in Figure 10.2a. Even at a low rank the image is recognizable. By rank 40, the approximation visibly differs very little from the original.

The following code demonstrates how to use `plt.imread` and `plt.imshow` to read in an image, convert it to black and white, and show it. The function from Problem 3 can then be used to calculate an approximation of `X`.

```
>>> import matplotlib.pyplot as plt
>>> # Take only one layer (layer 0) of the image
>>> X = plt.imread('hubble_image.jpg')[:,:,0].astype(float)
% >>> X.nbytes     #number of bytes needed to store X  <---- this seems ↩
    superfluous
>>> plt.imshow(X, cmap="gray")
>>> plt.show()
```

Problem 5. Using the `svd_approx` function from Problem 3, write a function `compress_img` that accepts two parameters `filename` and `k`. The function should plot the original image and the best rank k approximation of the original image.

(a) NGC 3603 (Hubble Space Telescope).



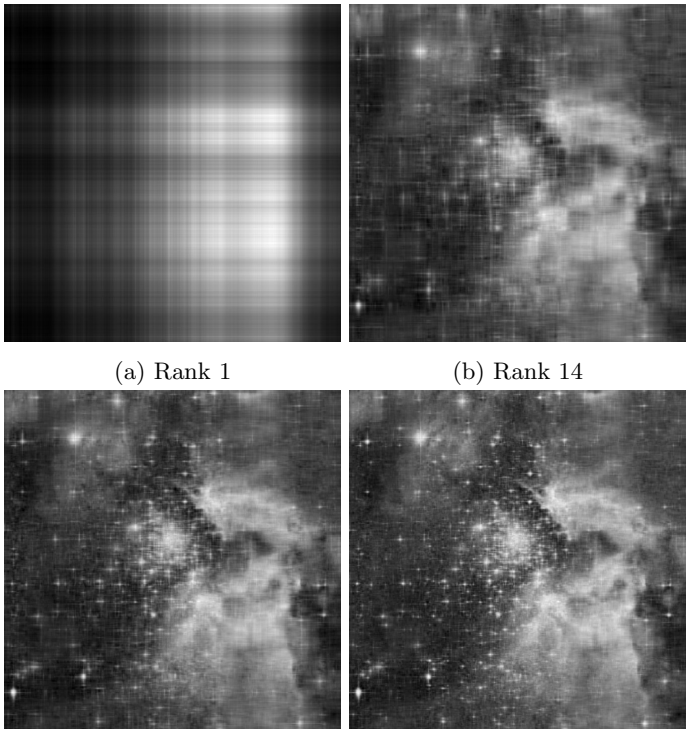
(b) Singular values from greatest to smallest on a log scale

Figure 10.2: An image and its singular values.

While `svd_approx` worked for grayscale images, the `compress_img` function should work on color images. You may split the image into its three RGB layers and approximate each layer separately, then recombine them. Test your function on `hubble_image.jpg` Your output should be similar to Figure 10.4.

Hints:

- Sometimes `plt.imshow` does not behave as expected when being passed RGB values between 0 and 255. It behaves much better when being passed values between 0 and 1.
- Since the SVD provides an approximation, it is possible that the SVD will generate values slightly outside the valid range of RGB values. To fix this, use fancy indexing (as discussed in the NumPy and SciPy lab) to set values greater than 1 to 1 and values less than 0 to 0.



(a) Rank 1 (b) Rank 14

(c) Rank 27 (d) Rank 40

Figure 10.3: Different rank approximations for SVD-based compression. Notice that higher rank is needed to resolve finer detail.

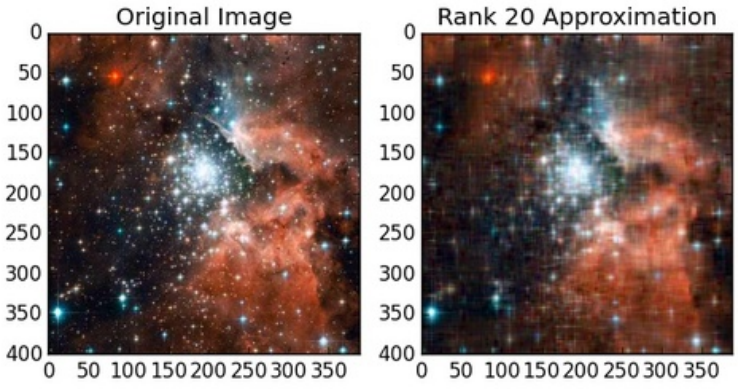


Figure 10.4: Correct output for the best rank 20 approximation.