Dynamic SVD Compression

Shane Stevenson

Data Storage Solution Inc.

Email: shanestevenson04@gmail.com

Website: shane-stevenson.github.io

November 14, 2023

Abstract

The modern world is fueled by data. Companies collect data on their customers, governments collect data on their citizens, schools collect data on their students, and engineers collect data on the world around them. We collect all this data because data contains knowledge, and knowledge is what informs decisions. The modern industry is greatly concerned with the processing of our petabytes of data because understanding data means a greater ability to understand the world, and a greater ability to make better decisions. Low dimensional approximation of high dimensional data as well as data compression are some of the most important subjects in both modern data science and computer science. Furthermore, these subjects have been proven to have extremely valuable real-world applications.

This paper delves into dynamic image compression, leveraging Singular Value Decomposition (SVD) as a powerful tool to uncover the compressibility of natural data, especially natural images. The goal of this paper is to create a python program that uses Singular Value Decomposition in order to efficiently and dynamically compress various images. Unlike many mainly theoretical SVD usages, our goal is to create a functional program that can be leveraged to save real storage. Our results will be quantified by examining our compression ratios, how much storage we saved practically, as well as the loss of quality of our data.

In order to leverage the SVD to create a functional compression program, we must first examine the mathematical formulation behind Singular Value Decomposition and understand why the SVD is so useful. Furthermore, this paper aims to explore the natural compressibility and sparsity of real-world data, and show why this natural compressibility allows the SVD to be so useful.

Introduction

Singular Value Decomposition is a critical innovation from linear algebra that has found use in just about all fields like electrical engineering, computer science, statistics, machine learning, and a plethora of other important fields. To give some practical examples, Singular Value Decomposition is pivotal in signal processing, Principal Component Analysis (PCA), data compression, data processing, and plenty more.

Compression is the focus of this paper, and the question we ask is "how can we dynamically compress a set of images using the SVD to fit specific requirements?" (requirements meaning storage limits or quality minimums). In order to begin answering this question we must first understand the mathematical formulation behind Singular Value Decomposition.

According to Steven L. Brunton and J. Nathan Kutz in their book "*DATA DRIVEN SCIENCE & ENGINEERING*", singular value decomposition can be characterized as follows:

$$\forall A \in \mathcal{R}^{nxm},\ A = U\Sigma V^{*}$$

Where U and V are unitary matrices, Σ is a diagonal matrix, and * represents the complex conjugate transpose. One of the most important features of the SVD is that it exists for any sized matrix, including non-square matrices. The figure below is included to help the reader understand the dimensions of the matrices involved in the SVD.

IMAGE HERE

Because of how U and V^{*} are formed, they contain information about the inherent scaling and orientation that the A transformation performs. This is because U, or the left singular matrix,

consists of a set of orthonormal eigenvectors of AA^T, while V, or the right singular matrix, is comprised of a set of orthonormal eigenvectors of A^TA. These orthonormal eigenvectors are then ordered by the magnitude of their corresponding eigenvalues in descending order. The square root of these eigenvalues are the diagonal entries of Σ, in descending order. It has been proven that $\tilde{A}_k = \tilde{U}\Pi\tilde{V}^*$ is the best k-rank approximation of A where $\tilde{U}$ is U with only the first k columns, $\tilde{V}^*$ is V^* with only the first k rows, and Π is Σ with only the first k entries. A diagram of this k-rank approximation is shown below.

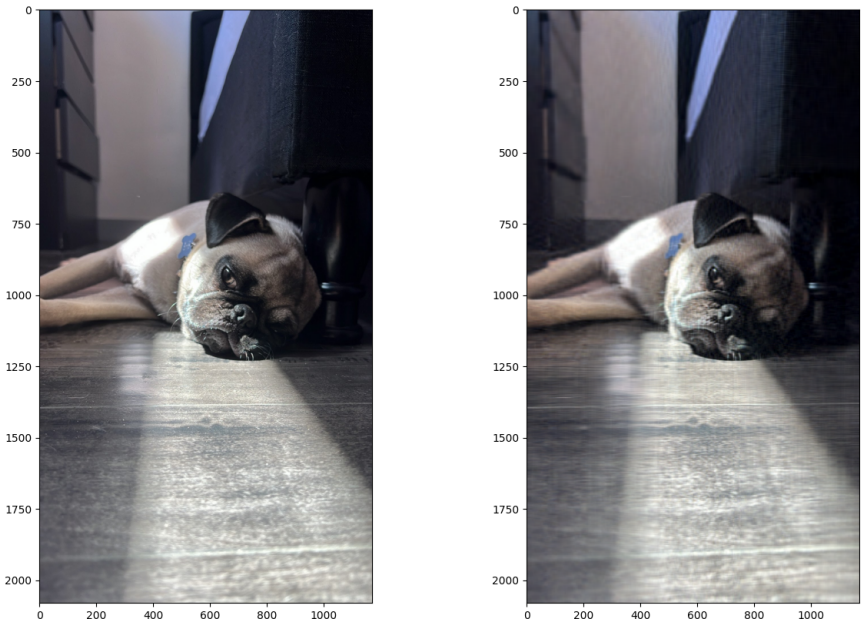
IMAGE HERE

Methodology

Now that the SVD has been sufficiently defined within the context of this research paper, let's examine the SVD in terms of image analysis and image compression. Within this realm, an image is represented as the matrix A with 3 channels, where each channel in A is the red, green, or blue value of a pixel. Singular value decomposition can then be performed on each channel the image matrix A, and then a k-rank approximation can be taken of A. Shown below is an example of singular value decomposition and k-rank approximation of an image in python.

```
def compressWithColor(origFilePath, compressedFilePath, dVals):
    #Read the image
    im = io.imread(origFilePath)
    #Create an array of same shape
    compressedIm = np.full(im.shape, 255)

    #Take the SVD of the red, green, and blue channels
    for i in range(3):
        col = im[ : , : , i]
```

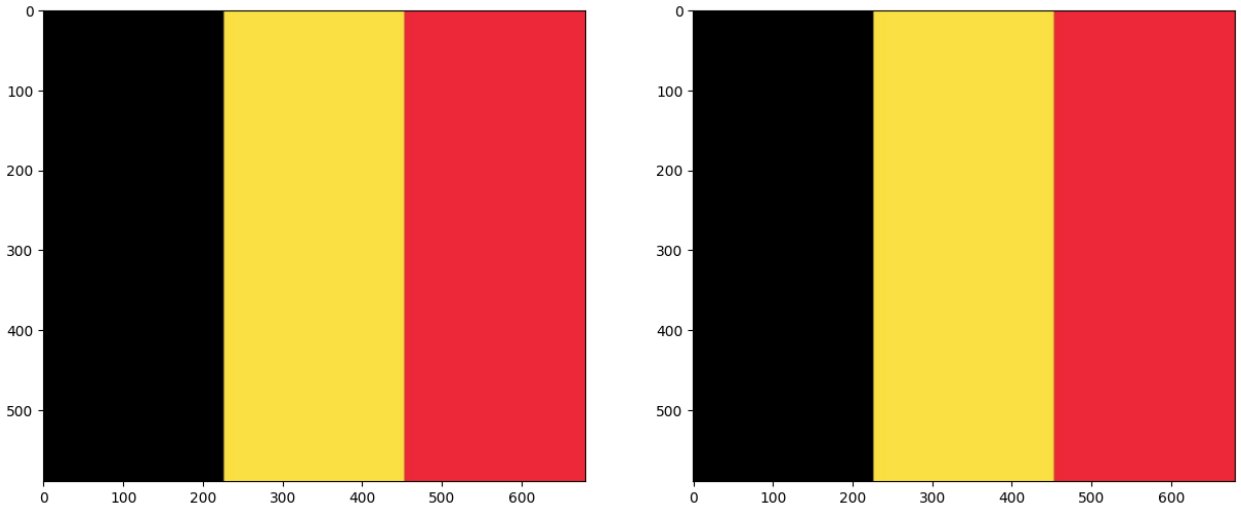
```
        u, d, vt = np.linalg.svd(col, full_matrices=False)

        #Reconstruct each channel with only k singular values
        smallu = u[0:, 0:dVals]
        smalld = np.diag(d[:dVals])
        smallvt = vt[0:dVals, 0:]
        #Place the compressed channels in the compressed matrix
        compressedIm[ : , : , i] = smallu @ smalld @ smallvt


    #Make sure values are from 0-255 for all channels
    compressedIm = np.clip(compressedIm, 0, 255)
    #Change dtype to uint
    compressedIm = compressedIm.astype(np.uint8)

    #Display results
    fig, axes = plt.subplots(1, 2, figsize=(15, 10))
    ax = axes.ravel()

    ax[0].imshow(im)
    #ax[1].plot(np.ma.log10(d), color='k')
    ax[1].imshow(compressedIm)
    # ax[3].imshow(im-compressedIm, cmap=plt.cm.gray)

    #save the image
    matplotlib.image.imsave(compressedFilePath, compressedIm)

compressWithColor('static/dog1.jpg', 'compressions/compressed3.jpg', 50)
```



In this example we take an image's color channels, decompose them into the $U\Sigma V^*$ matrices of the SVD, and then find the rank-50 approximation by truncating the three matrices for each color channel as previously explained. Then to recreate the image the reconstructed matrices are stacked on top of each other to return to the format of the original image. This particular example results in a compression ratio of .04, which means only 4% of the original data is used to recreate the compressed image. Furthermore, the peak signal-to-noise ratio is about .001. Peak signal-to-noise ratio is a measurement of how good an approximation of an image is. It is calculated as follows:

$$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

Clearly, 50 singular values is all it took to fairly faithfully recreate this image. However, some images might require far less, or far more. For example, simple images like nations flags might only need anywhere from 1 to 20 singular values to faithfully reconstruct.



Belgium's flag can be completely represented with only 1 singular value, one left singular vector, and one right singular vector. This particular example results in a compression ratio of

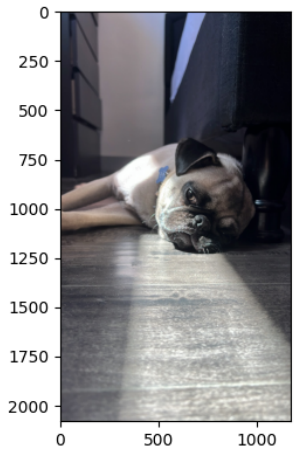
.0017, meaning that only .17% of the original image is actually required to reconstruct the image. Furthermore, the peak signal-to-noise ratio is 0, meaning that this is an exact reconstruction.

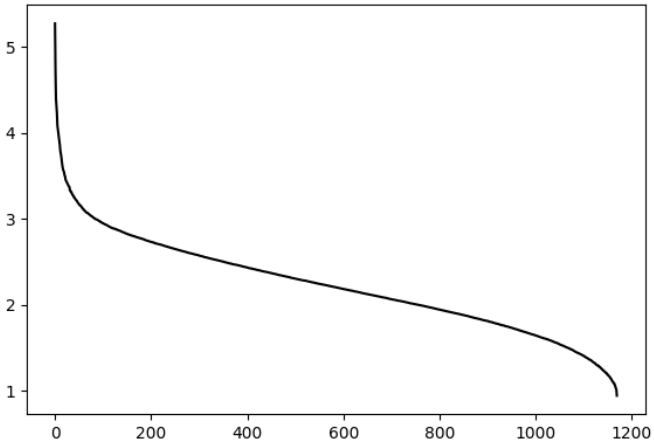
Because different images might require different compression, our goal is to dynamically decompose and reconstruct images based on storage or quality needs. Thus, the following Python code will decompose and reconstruct an image with the optimal amount of singular values in order to reach a specified peak signal-to-noise ratio.

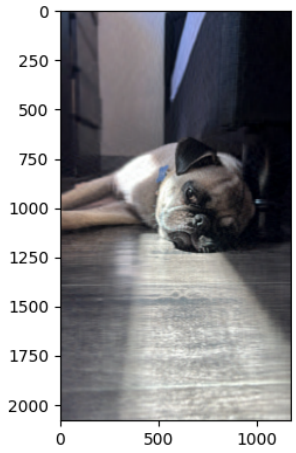
```
def compressWithPSTN(origFilePath, compressedFilePath, desiredpstn):
    #Read file
    im = io.imread(origFilePath)

    #Initialize the different channels
    red = im[ : , : , 0]
    green = im[ : , : , 1]
    blue = im[ : , : , 2]

    #Take the SVD of red, green, and blue channels
    red_u, red_d, red_vt = np.linalg.svd(red, full_matrices=False)
    green_u, green_d, green_vt = np.linalg.svd(green, full_matrices=False)
    blue_u, blue_d, blue_vt = np.linalg.svd(blue, full_matrices=False)

    #Place these SVDs in a list
    svds = [[red_u, red_d, red_vt], [green_u, green_d, green_vt], [blue_u,
blue_d, blue_vt]]

    #Begin iteration at the maximum amount of dvalues
    dVals = int(len(red_d))
    #For binary search, increment begins at half the number of dvals
    increment = int(np.ceil(dVals/2))
    pstn = 0
    #Create an array of same shape as the original image
    compressedIm = np.full(im.shape, 255)

    #If increment is 1, that means binary search has converged
    while increment != 1:
        #Here we create the compressed image by taking the economy svd of
each color channel.
```

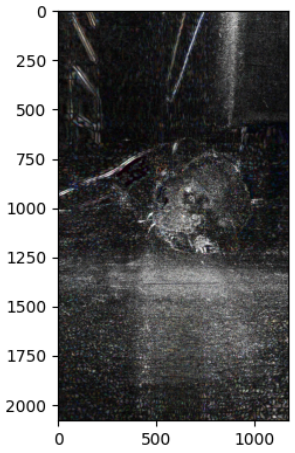
```
        for i in range(3):
            compressedIm[ : , : , i] = svds[i][0][0:, 0:dVals] @
np.diag(svds[i][1][0:dVals]) @ svds[i][2][0:dVals, 0:]
            compressedIm = np.clip(compressedIm, 0, 255)

        #In order to determine peak signal-to-noise, all values must be
between 0-1. Thus we divide by 255, take the pstn, and then revert
        im = im.astype(float)
        im/=255
        compressedIm = compressedIm.astype(float)
        compressedIm/=255
        pstn = np.mean((im-compressedIm)**2)
        im*=255
        np.round(im)
        im = im.astype(int)
        compressedIm*=255
        np.round(compressedIm)
        compressedIm = compressedIm.astype(int)

        #If we are less accurate than we want to be, increase the number of
dvals
        if pstn > desiredpstn:
            dVals += increment
        #If we are more accurate than we need to be, decrease the number of
dvals
        else:
            dVals -= increment
        increment = int(np.ceil(increment/2))
        if dVals < 1:
            dVals = 1
            break

    #Create the final compressed image with the converged values of dvals
    for i in range(3):
        compressedIm[ : , : , i] = svds[i][0][0:, 0:dVals] @
np.diag(svds[i][1][0:dVals]) @ svds[i][2][0:dVals, 0:]
    #Ensure all values are between 0 and 255
    compressedIm = np.clip(compressedIm, 0, 255)
    im = np.clip(im, 0, 255)
```

```
    #Show images
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))
    ax = axes.ravel()

    ax[0].imshow(im)
    ax[1].plot(np.ma.log10(red_d), color='k')
    ax[2].imshow(compressedIm)
    ax[3].imshow(np.abs(im-compressedIm)*10, cmap=plt.cm.gray)

    #Save image
    #Convert to np.uint8 for saving
    compressedIm = compressedIm.astype(np.uint8)
    matplotlib.image.imsave(compressedFilePath, compressedIm)

    #Take the final PSTN
    im = im.astype(float)
    im/=255
    compressedIm = compressedIm.astype(float)
    compressedIm/=255
    pstn = np.mean((im-compressedIm)**2)


    ########
    # Print info
    print('compression Ratio: ', dVals/len(red_d))
    print('peak signal-to-noise: ', pstn)

    #########

compressWithPSTN('static/dog1.jpg', 'compressions/compressed3.jpg', .001)
```









Let's consider the "`compressWithPSTN()`" function. Given an input image, a save location, and a desired peak signal-to-noise ratio, this function will converge at the optimal k-rank approximation to reconstruct the image closest to the desired PSTN. It functions by checking the acquired PSTN ratio from using all of the singular values, and then uses binary search to converge at the optimal number of singular values for the desired PSTN. Finally, the compressed image is saved to the provided save location.

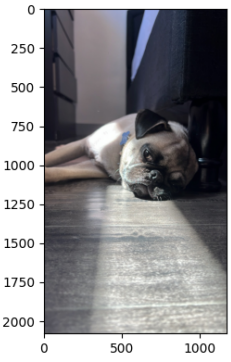
In the above example, "`compressWithPSTN()`" was called with a desired PSTN of .001, and this resulted in a practical PSTN of .000995, and a compression ratio of .04188. Furthermore, a scree plot is provided that indicates (on a log 10 scale) the relative importance of each singular value and by extension, each singular vector. Additionally, a residual image is

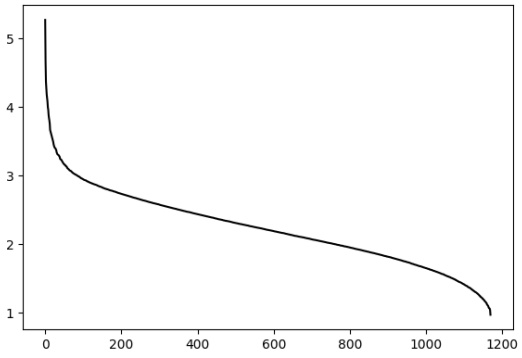
shown to help explore where the SVD failed to capture the image. This is obtained by taking the absolute value of the difference between the original image and the compressed image. The residual is multiplied by 10 to enhance visibility. It is important to note that the peak signal-to-noise ratio is the quantification of how accurate an approximation was, and should be relied upon more than the residual image.

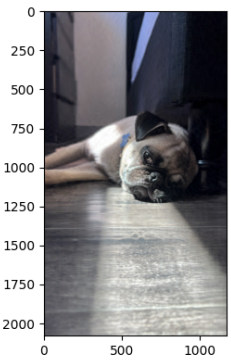
In addition to dynamic compression based on quality, the below example will dynamically compress an image based on a desired compression ratio:

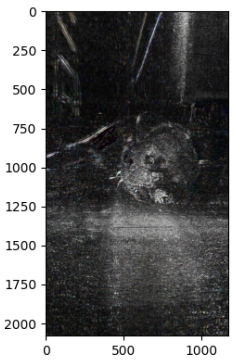
```
def compressWithCompressionRatio(origFilePath, compressedFilePath,
desiredCompressionRatio):
   #Load the original file
   im = io.imread(origFilePath)
   #Create numpy array of same shape
   compressedIm = np.full(im.shape, 255)

   #Take the SVD of each color channel
   for i in range(3):
       col = im[ : , : , i]
       u, d, vt = np.linalg.svd(col, full_matrices=False)
       #We use dvals proportional to what our desired compression ratio
is.
       dVals = np.uint8(len(d) * desiredCompressionRatio)

       smallu = u[0:, 0:dVals]
       smalld = np.diag(d[:dVals])
       smallvt = vt[0:dVals, 0:]
       compressedIm[ : , : , i] = smallu @ smalld @ smallvt


   #Put a ceiling and floor on all the values
   compressedIm = np.clip(compressedIm, 0, 255)
   #Store as an int for residual
   compressedIm = compressedIm.astype(int)
   im = im.astype(int)

   #Show results
```

```
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))
    ax = axes.ravel()

    ax[0].imshow(im)
    ax[1].plot(np.ma.log10(d), color='k')
    ax[2].imshow(compressedIm)
    ax[3].imshow(np.abs(im-compressedIm)*10, cmap=plt.cm.gray)

    #Save image
    compressedIm = compressedIm.astype(np.uint8)
    matplotlib.image.imsave(compressedFilePath, compressedIm)

    im = im.astype(float)
    im/=255
    compressedIm = compressedIm.astype(float)
    compressedIm/=255

    # Print info
    print('compression Ratio: ', dVals/len(d))
    print('peak signal-to-noise: ', np.mean((im-compressedIm)**2))

compressWithCompressionRatio('static/dog1.jpg',
'compressions/compressed3.jpg', .05)
```





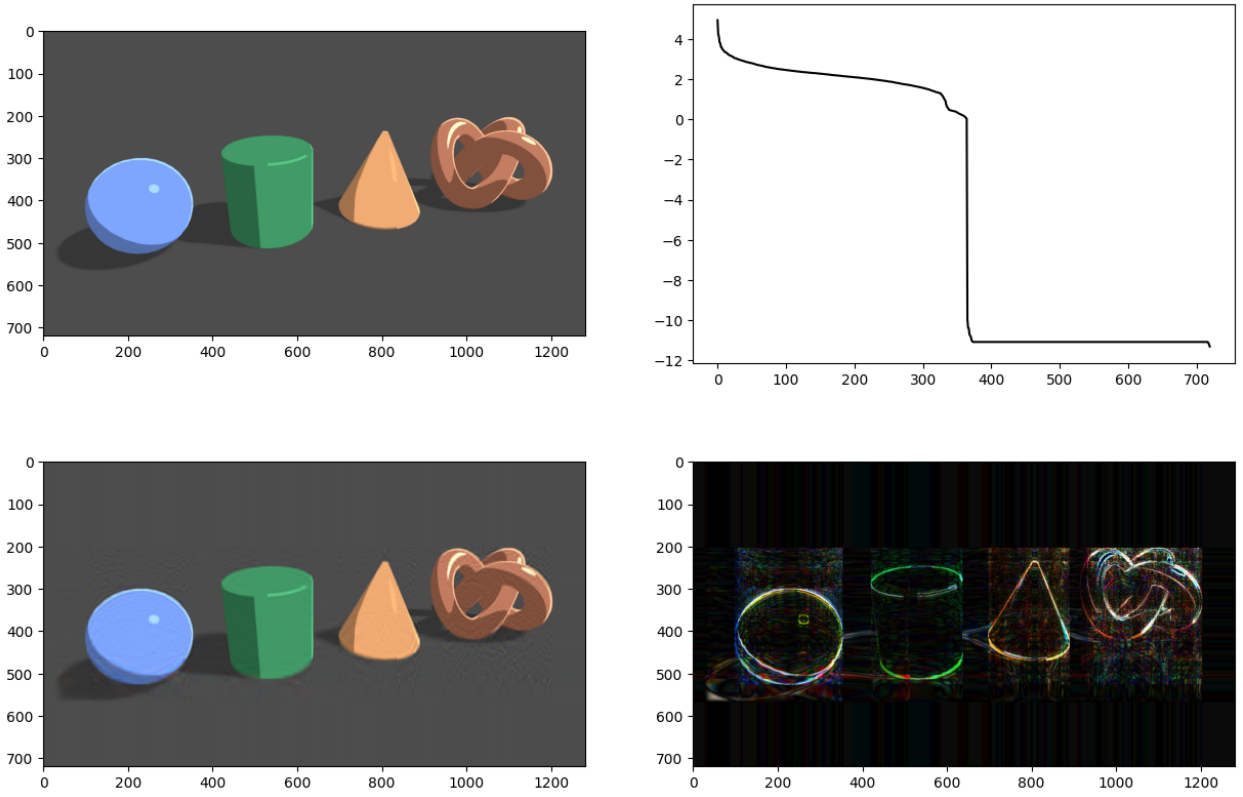




In this example, the image is compressed with a desired ratio of .05. Upon completion, this image was reduced to .49% of its original size, with a peak signal-to-noise ratio of .00882. Once more, a scree graph and a residual image are displayed alongside the original and compressed images. For reference, all of this code can be found ON MY GITHUB!!!

Already we have achieved dynamic image compression with regard to both quality and compression ratios, but it's time we move this formula into the real world, and see how our method fairs at compression. Firstly, let's look at how much memory we can save on a single image.

```
compressWithPSTN('static/shader.png', 'compressions/shader.png', .00025)
```



Now, according to our code, we have reconstructed the original image using only 5.7% of the original data. However, when we examine the original and the compressed file, my compressed image is almost 10 times the size of the original image! My “compressed” image is

actually taking up more space than the original, How is this possible? Essentially, it comes down to understanding compression schemes that are already in place. This image is a .png file, which means many rows of data can be represented by two values, value, and the number of pixels that this color continues for. When we compress our image, many of the pixels are slightly discolored, so when the image is reconstructed, the .png compression scheme will not be nearly as effective as it was on the original image.

If we stored image data simply as a matrix of pixel values, the SVD would be phenomenal at compression. If this were the case, a .057 compression ratio would truly mean we saved 94% of the storage the original image took up. However, this is where the concept of regularity in images saves our original goal. Jpg compression is fundamentally different from png compression. Firstly, jpg compression is lossy much like singular value decomposition, and much like singular value decomposition, jpg compression uses regularity in an image to store significantly less data.

Specifically, jpg compression works by performing a Discrete Cosine Transform on the original image. The DCT transforms spatial data to frequency data much like the fourier transform. Then, these frequencies are saved as DCT coefficients, and the unimportant ones are thrown away, much like the SVD's small singular values. Finally, when a jpg file is loaded, the inverse DCT is performed on the remaining DCT coefficients, and an image approximation is formed.

Clearly, jpg compression and singular value decomposition are very similar as they both leverage the inherent regularity of natural data to throw away unimportant data. So, how can we use this knowledge to create a practical use for our code? Well, both jpg compression and singular value decomposition remove unimportant inherent regularity in the image. So, if we

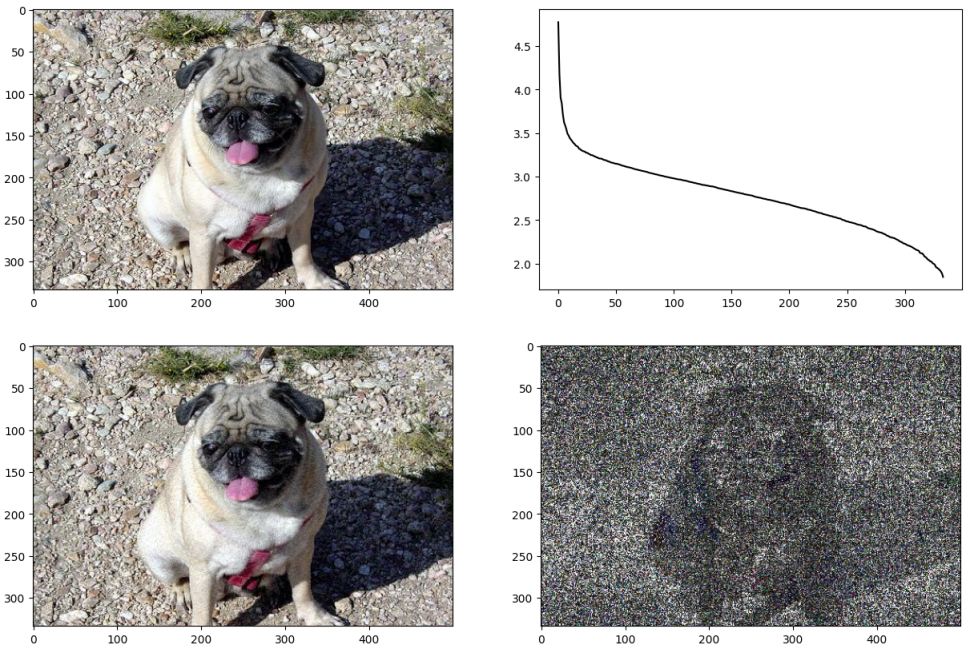
have a jpg image, decompress the jpg image to get a matrix of pixel values, perform singular value decomposition on it, remove unimportant singular values, and then perform perform jpg compression on the compressed image, there will be less DCT coefficients than before because singular value decomposition regularized our image. So, all that's left is to test this hypothesis.

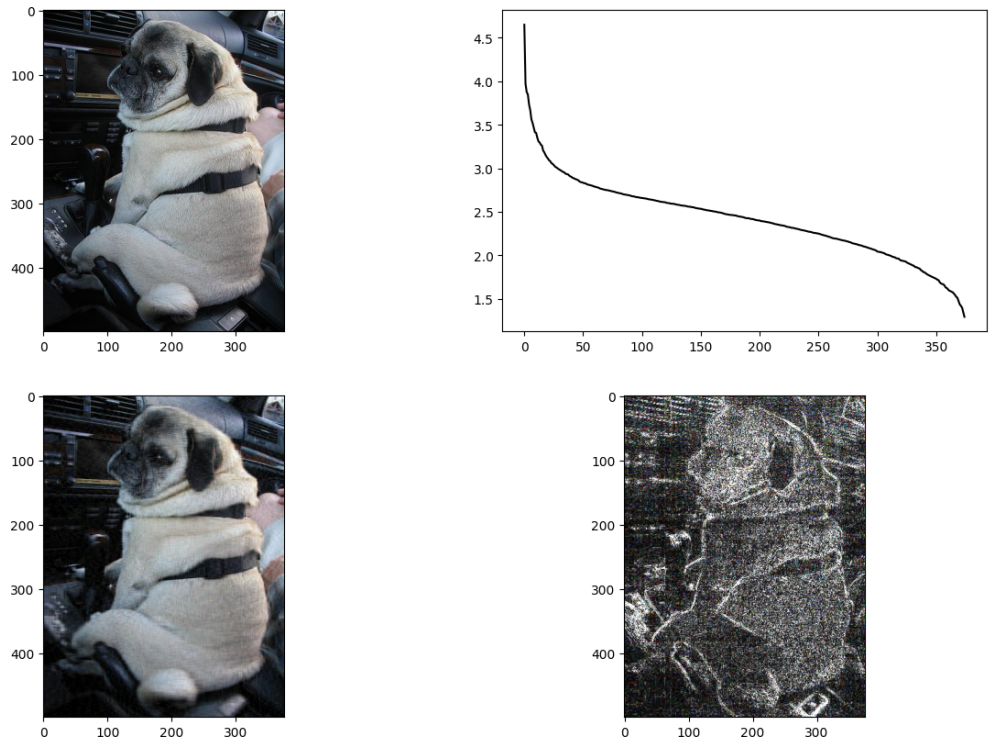
Results

In order to test our hypothesis, we need data. To see how much compression we can get with the combination of jpg compression and singular value decomposition, I used the stanford dataset "*Stanford Dogs Dataset*", specifically their 200 images of pugs. Using the below code, I reference the previously explored "`compressWithPSTN()`" algorithm to iterate over each image, reconstruct the image from jpg compression, perform singular value decomposition on the reconstructed image, and then compress it once more with jpg compression.

```
for filename in tqdm(os.listdir('pugImages/')):
   if filename[0] != '.':
       compressWithPSTN(f'pugImages/{filename}',
f'compressedPugs2/{filename}', .0025)
```

Here are some results:



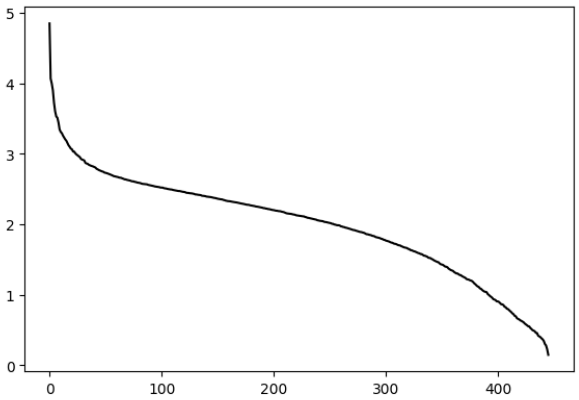
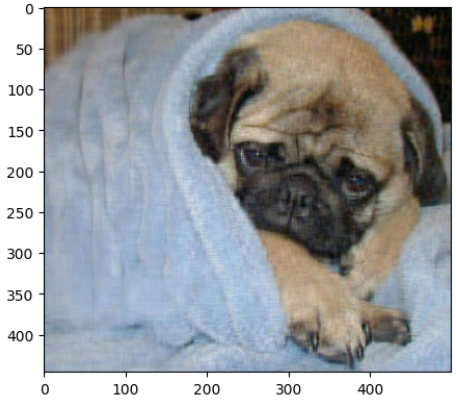
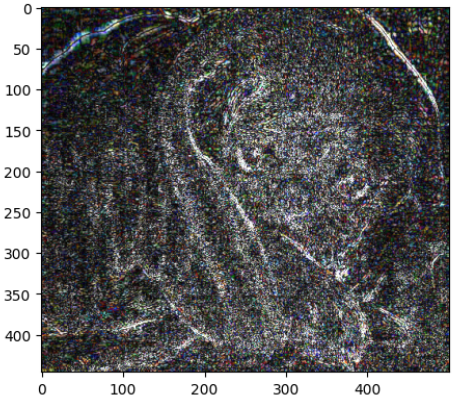
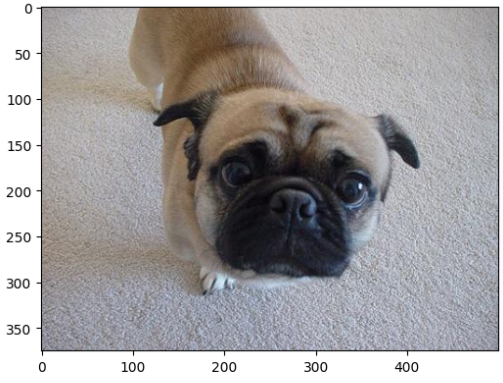
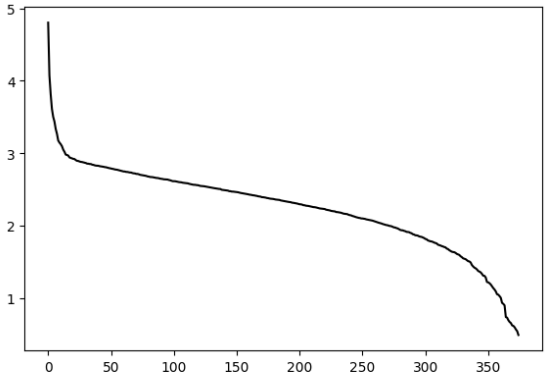
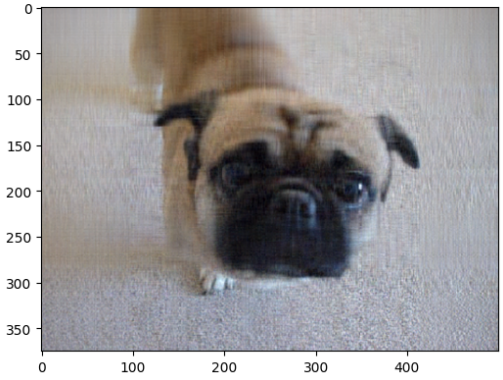
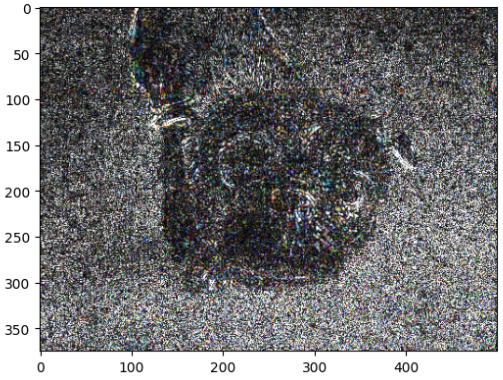


The folder of original images was 7.4 MB in size, upon compression, the folder of new images was 6.0 MB in size. This is a compression ratio of about .81, which may seem small, but when compressing jpg images that have already been compressed extremely efficiently, this is no small accomplishment.

Parallely, compressing based on a desired compression ratio found similar results. Much like the previous code, the below code iterates through all of the original images, and dynamically compresses each image based on a desired compression ratio.

```
for filename in tqdm(os.listdir('pugImages/')):
   if filename[0] != '.':
       compressWithCompressionRatio(f'pugImages/{filename}',
f'compressedPugs/{filename}', .08)
```

Below are some results:

Once again, the original image was 7.4 MB in size, and the folder of new images was 5.5MB in size. This is a compression ratio of .74, but as we can see, the dropoff in quality becomes apparent at this compression. Still though, these images could easily be used for AI model training, or adjacent tasks.

The below table summarizes our numerical results

TABLE HERE

Conclusion

Through the course of this paper we examined four things. Firstly, we explored the fundamental concepts and mathematical foundations behind singular value decomposition. Secondly, we stated our problem, which was how we could achieve dynamic image compression with the SVD, and looked at some primitive examples of SVD compression with color. Thirdly, we employed dynamic singular value decomposition to compress images based on specified requirements. Lastly, we examined the adjacency between behind SVD and jpg compression, and explored the results of our dynamic SVD compression on a set of jpg images.

Overall, while our results were not shocking, and were primitive compared to the modern image compression algorithms applied on every modern day computer, we still created a program that could be used to compress a set of jpg images, leveraging singular value decomposition.

In her paper “Deep Learning-Based Vehicle Classification for Low Quality Images”, Sumeyra Tras realized accuracies of over 90% on classification of low quality images. Perhaps if the size of a dataset is more damaging than a reduction of quality of a dataset, the techniques provided in this paper could be used to create a new dataset for machine vision tasks.

References

Brunton, S. L., & Kutz, J. N. (2021). *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems and Control*. Cambridge University Press.

Strang, G. (2006). *Linear algebra and its applications*. Thomson.

Tas, S., Sari, O., Dalveren, Y., Pazar, S., Kara, A., & Derawi, M. (2022). Deep learning-based vehicle classification for low quality images. *Sensors*, *22*(13), 4740. https://doi.org/10.3390/s22134740