Image Compression using Singular Value Decomposition

Fernando Rojas - Nicolás Vega
$frrm04@gmail.com - nicovporta.luppi@gmail.com$
Information Engineering, Universidad Politécnica Taiwán Paraguay, Asunción

November 27, 2023

Abstract

Nowadays it is common to encounter with high quality pictures, and higher quality means more data to store which leads us to the problem of having a full storage in our electronic devices, such as smartphones or Personal Computers. Into this research we are able to use a fundamental theory of linear algebra and apply it within a programming script in order to compress a given file. The program aims to resize an image provided by the user so they can choose whether to compress the image with or without colours, and later on, download the preferred version of the image.

1 Introduction

Throughout the years it was possible to witness how technology advances and improves in different areas, in the photography context is no exception, there has been many changes in the past 200 years. Society went from having to pose for several minutes, to quick snapshots that takes milliseconds to capture and process a picture. All of those improvements also represents an increase in every aspect of photography, such as the increment in the focal length, camera lenses, pixels proportions, picture quality and also the size of the image file. Bear in mind the concept of "size of the image file" which refers to how much bytes an image "carries".

Considering how much space the pictures occupy, it is normal to talk about methods to compress the images or set of images in order to have more storage in our devices. Image compression is a crucial technique in computer graphics, image processing, and data storage. It involves reducing the size of an image while maintaining an optimal visual quality. Linear algebra, particularly Singular Value Decomposition (SVD), can be used for efficient image compression. In this research it will be explored the mathematical concepts of SVD and how they can be applied to compress images while implementing a compression algorithm using Python. It also offers a real-world understanding of image compression, which has wide-ranging applications in fields like computer vision, data storage, and multimedia processing.

In the upcoming sections it will also be discuss the efficiency and functionality of the compression process and how the experimental implementations results are compared to the images without SVD compression.

2 Singular Value Decomposition Theory

2.1 The key idea

Any matrix $A \in \mathbb{R}^{m \times n}$ describes some linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ by specifying in its columns where the basis vectors should land relative to the initial basis. Basically, linear transformations are able to stretch, rotate and flip space and, of course, do any combination of these actions at once. So the researchers might be interested in creating some kind of normal form for linear transformations consisting of only geometrically simple operations, so that it is possible to convert any linear transformation to this normal form. This is the key idea behind SVD, it is a normal form that consists of 2 rotations (U and V^T) and 1 scaling (Σ).

Therefore, any matrix $A^{m \times n}$ can be expressed as: $A = U\Sigma V^T$ where:

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix that rotates space, column vectors of this matrix are often referred to as left singular vectors.
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix (all elements that are not on the diagonal are zero, diagonal elements are called singular values), which scales space.
- $V^T \in \mathbb{R}^{n \times n}$ is an orthogonal matrix, which only rotates space. Row vectors of V^T are also known as right singular vectors.
- Singular values (on the diagonal) of Σ are sorted in descending order and are different. They will be denoted with $\sigma_1 > \sigma_2 > \cdots > \sigma_N > 0$ where N is the amount of non-zero entries on the diagonal (it also follows, that $N = rank(\Sigma) = rank(A)$

The entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ on the diagonal of Σ are called singular values of A. Orthogonality of U and V means that they correspond to rotations (possibly followed by a reflection) of m-dimensional and n-dimensional space respectively. U represents the left singular vectors and it can be found by computing AA^T and then taking the eigenvectors from the computed matrix and normalize them in case they have not been normalized yet, those eigenvectors will be the columns of U, it is important to assign their order from higher to lower. V represents the right singular vectors and it follows the same process as U but instead of computing AA^T, it has to be compute A^TA. It is relevant to notice that only Σ changes the length of vectors.

The singular values of A (defined as σ)are the square roots of the positive eigenvalues of A^TA and AA^T, those values are the entries of the diagonal matrix Σ and they have to be ordered from higher to lower.

The relationship between the rank and the quality of the image approximation follows a trade-off: higher rank approximations preserve more details but require more storage, while lower rank approximations reduce storage but sacrifice image quality.

In the image compression using SVD, altering the rank (n) allows you to observe how different levels of information retention affect the quality and fidelity of the reconstructed image. Smaller n values lead to more pronounced loss of details, while larger n values preserve more information but might still differ from the original however it is not as notable as with the lower ranks.

2.2 Image Compression with SVD

Any image can be represented as a matrix of pixels, where each pixel (typically) consists of 3 channels: for the red, green and blue components of the color, respectively. Each channel can be represented as a $(m \times n)$ matrix with values ranging from 0 to 255. And hence compress the matrix A representing one of the channels.

To do this, it is necessary to compute an approximation to the matrix A that takes only a fraction of the space to store. Now here is the great thing about SVD: the data in the matrices U, Σ and V is sorted by how much it contributes to the matrix A in the product. That enables the chance to get quite a good approximation by simply using only the most important parts of the matrices.

Now it is time to define a number k of singular values that are going to be used for the approximation, this will also represent N which is a parameter mentioned in the previous section. The higher the number k, the better the quality of the approximation, but also the more data is needed to store it.

The values of k are already defined in the program made by the researchers, they are: 5, 20, 100, 200, 500, 750 and 1000. Those numbers are chosen in order to appropriately show the different image approximations, in the other hand, the singular values σ_N proper from an image all are calculated by a function in the code snippet that will be presented in the upcoming sections.

3 Data Collection and Programming Script

3.1 Data set

The data set employed in this research project is partly composed of photographs captured by the research group and its collaborators, which are enlisted in the *Acknowledgment* section.

3.2 Code snippet

As mentioned before, the program elaborated by the research group, aims to offer the user the option of compressing an image in a gray scale or with red, green and blue scale (RGB); since it is possible to reduce even more the size of the file by scaling it in the gray color spectrum. Finally, different versions of the image will be presented, and the user will be able to edit and download the chosen compressed version of the picture.

Several libraries are used for the project, however, *numpy* is the one which has the linear algebra function of computing the singular values of a matrix and other mathematics functions which are fundamental for the program, notice that the *numpy* library was imported as *np*. It is also important to clarify that since the symbol Σ will be represented as the letter S in the following code lines.

```
grayx = np.mean(x, -1)
```

Here it is computed the average of every RGB column so it is possible to make them all have the same value, since if $RED = GREEN = BLUE$ then the result is the gray color.

```
U, S, VT = np.linalg.svd(grayx, full_matrices=False)
S = np.diag(S)
```

The first line uses the SVD function on the *grayx* matrix, assigning the three fundamental variables of the definition $A = U\Sigma V^T$ to U, Σ and VT respectively. It also computes a compact SVD, meaning it only returns the essential parts of the decomposition. Then it creates a new diagonal matrix using the values from the array Σ, this may seem redundant since the Σ matrix defined with the SVD function is already a diagonal matrix but it is relevant because it is precise to use a different variable for the following procedures.

```
for n in (5, 20, 100, 200, 500, 750, 1000):
    grayxapprox = U[:, :n] @ S[0:n, :n] @ VT[:n, :]
```

A loop is created in order to show the different image approximations with different k for the singular values. In this loop it is defined a new variable which represents the compressed image, taking only the values of U until k columns, of V^T until k rows and Σ with k number of singular values. That is how the approximations are created, the function use a fixed numbers of ranks for each component of the SVD and starts processing the new plot.

For the RGB image compressing procedure it is done a similar process as before, but it is thrice the work since now there are three different channels or columns that the program needs to work with (red, green and blue).

```
red_U, red_S, red_VT = np.linalg.svd(x[:, :, 0], full_matrices=False)
green_U, green_S, green_VT = np.linalg.svd(x[:, :, 1], full_matrices=
    False)
blue_U, blue_S, blue_VT = np.linalg.svd(x[:, :, 2], full_matrices=
    False)

red_S = np.diag(red_S)
green_S = np.diag(green_S)
blue_S = np.diag(blue_S)
```

The functions of SVD and diagonalization are utilized once again but with the difference that now they have to be defined for the columns $0, 1$ and 2 which represents the RGB channels.

```
for n in (5, 20, 100, 200, 500, 750, 1000):
redapprox = red_U[:, :n] @ red_S[0:n, :n] @ red_VT[:n, :]
greenapprox = green_U[:, :n] @ green_S[0:n, :n] @ green_VT[:n, :]
blueapprox = blue_U[:, :n] @ blue_S[0:n, :n] @ blue_VT[:n, :]
final = np.stack([redapprox, greenapprox, blueapprox], axis=-1)
final = np.clip(final, 0, 255)
```

In this loop it is defined three variables which represents the compressed image, taking only the values of U until k columns, of V^T until k rows and Σ with k number of singular values for each

channel. *np.stack* is a *numpy* function that stacks arrays along a new axis. For instance, if each channel has a shape of (height, width), the resulting shape would be (height, width, 3) because of the stacking along the last axis, in simple words, it is used for mixing the RGB values in a proper manner.

np.clip is used to limit the values in an array within a specified range. In this case, it ensures that all values in the final array are between 0 and 255; values less than 0 will be set to 0, and values greater than 255 will be set to 255 to ensure the pixel values remain within the valid range.

4 Comparison and Analysis

4.1 Image and compression comparison

Some sample images were chosen for displaying the practical application and the results of the program repercussions on images. Initially a given image is shown and then the compressed version with different ranks (n).

Figure 1: This is the original picture. (2890 Kilobytes)

Figure 2: This is the original picture with the gray scale. (179 KB)

Only from changing the original image to a gray scaled version of it, it has a notorious 93.8% decrease in file size. Now it is time for approximations...

Figure 3: First approximation done with $rank = 5$. (81 KB)

Figure 4: $Rank = 20$. (111 KB)

Figure 5: $Rank = 100$. (142 KB)

Figure 6: $Rank = 500$. (156 KB)

As shown, the approximated versions of the original pictures have been compressed, bear in mind that the larger the number of *rankornork* as previously defined, the sharper the picture approximation to the original image. In order to show this proportion, the following graph was plotted:

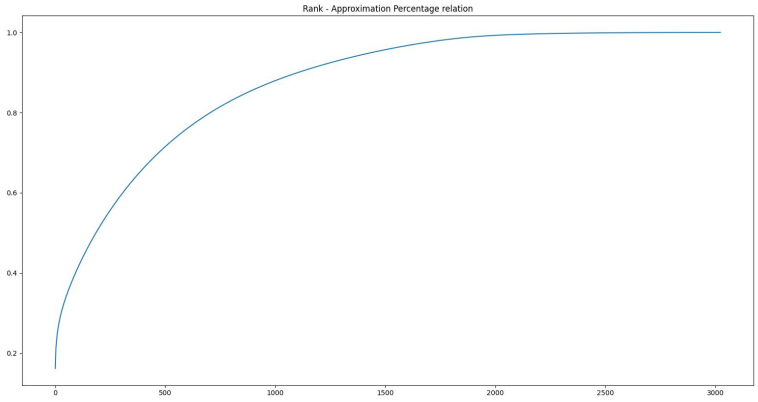

Figure 7: The x-axis represents n rank and the y-axis represents the percentage of approximation($1.0 = 100\%$).

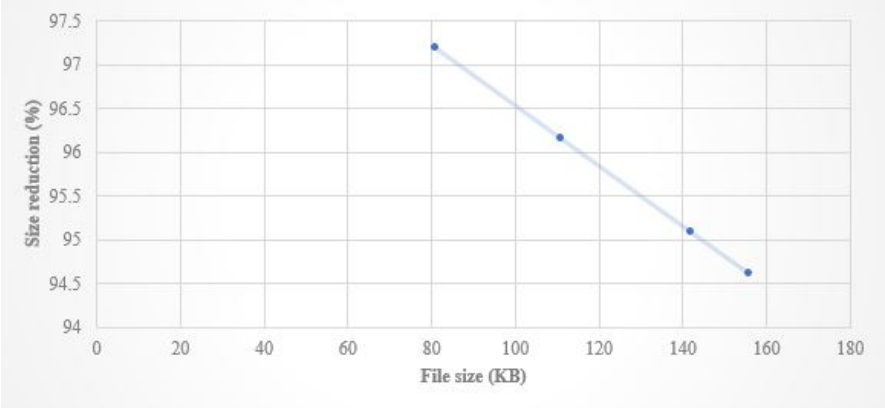

Figure 8: the table shows how much of reduction is done with each approximation file size

For the record, it is provided the same process with a different image but now the compression will be done keeping the same RGB scale as the original photo.

Figure 9: Picture of a temple in Yilan, a wide variety of colors are present

Figure 10: Approximations with 5, 100, 500 and 1000 of rank

4.2 Efficiency analysis

The programming language utilized for the code implementation is the high-level language Python, a choice made due to its versatility and extensive repository of libraries. The graphical user interface (GUI) was crafted using the *CustomTkinter* library, while the functional aspect leveraged the capabilities of *matplotlib* as well as *numpy* which were essential for the plotting and image processing of the program.

Remarkably, the code exhibits commendable efficiency. The GUI interface is resource-efficient and operates with noteworthy speed. Notably, the compression time for images varies based on image quality, but in the majority of instances, it remains remarkably swift, often completing within a minute. The image compression achieved through the code is notably effective, frequently resulting in a reduction of the image size by half without significant compromise to quality and also supports various picture formats such as $jpg, jfif, png, jpeg$. In general terms, data suggests that the code

is efficient, user-friendly, and does not demand a high-performance computer for seamless execution, with an optimistic processing time.

5 Discussion

During the course of the study and code development, a discernible trade-off emerged between the quality and size of images. As the compression of an image increased, it naturally resulted in a reduced file weight compared to the original photo. Furthermore, the research group encountered with the problem of having two relevant changes to the images, the file format and the "white frame" that appears after the compression process, however, both attributes do not have a high impact in the image size attribute, thus they have been considered just as minor problems. In terms of efficiency, and after many trials with different computers, the experimental results demonstrated a satisfactory perform in this regard. Admittedly, the compression time does exhibit an increase in correlation with higher image quality but, it is worth emphasizing that, the compression time typically remains below a minute in the majority of cases therefore it still works with efficiency.

6 Conclusion

In the process of this research, it was acquired the theory of the Singular Value Decomposition. Also, it was learned the practical applications that SVD offers in several fields. In this research the field chosen was the Image compression, how matrices can be applied to the images in order to compress them and make them more suitable for storing without losing so much quality in the process.

The SVD application in image compression has proven to be a valuable and versatile approach. Through the research and implementation of SVD-based compression techniques, a discernible trade-off between image quality and file size has been observed. This trade-off, while inherent in compression processes, has been effectively navigated to strike a balance that meets the specific needs of diverse applications.

The code developed for SVD image compression has demonstrated commendable efficiency, providing a notable reduction in file sizes without compromising overall image quality excessively. The equilibrium achieved between spatial occupation and image fidelity underscores the adaptability of SVD-based compression in addressing real-world challenges across various domains.

Furthermore, the user-friendly nature of the program, coupled with its ability to operate optimally on standard computing resources, enhances its accessibility and applicability. As users interact with the interface, they can appreciate the nuanced considerations made to optimize the compression process, allowing for a seamless and efficient user experience.

Looking ahead, the ongoing exploration of user feedback, potential enhancements, and future considerations will contribute to the continuous refinement of SVD image compression applications. The versatility and effectiveness of this approach places it as a valuable tool in addressing the evolving demands of image processing and storage across diverse fields and industries. Overall, the application of SVD in image compression stands as a promising and practical solution, offering a nuanced balance between image quality, file size reduction, and computational efficiency.

ACKNOWLEDGEMENT

The research group thanks the Professor Yaoting Tseng for all the counseling done during the writing of this research paper. Also give thanks to the person behind the data set material, Kay Caparroso who provided the photos shown in the figures. The authors would like to acknowledge the use of Gilbert Strang's books which were used for thoroughly understand certain Linear Algebra concepts as well as all the people who tested the program.

REFERENCES

Gilbert Strang, Introduction to Linear Algebra, 6th edition, Wellesley-Cambridge Press, 2009, ISBN: 978-0980232714

Steven L. Brunton, J. Nathan Kutz, Data-Driven Science and Engineering, Cambridge University Press, 2019, ISBN: 978-1108422093

Python Software Foundation. (2023). Python Language Reference, version 3.11. Retrieved from Python official website

NumPy: Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

Matplotlib: J. D. Hunter. Matplotlib: A 2D Graphics Environment. Computing in Science and Engineering 9, 90-95 (2007). DOI: 10.1109/MCSE.2007.55

Overleaf: Collaborative LaTeX Editor, https://www.overleaf.com

For a thoroughly examination of the whole code file, reach the repository of the whole program.