Singular value decomposition method for image compression

Abdelwahid Benslimane

wahid.benslimane@gmail.com

The singular value decomposition (SVD) of a $m \times n$ matrix A is $A = U\Sigma V^T$, where U is an orthogonal matrix of size $m \times m$, V is an orthogonal matrix of size $n \times n$ and orthogonal, and Σ is a diagonal matrix of size $m \times n$ with nonnegative diagonal entries $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p$, p = min$\{m, n\}$, known as the singular values of A.

$$A = U \begin{pmatrix} \Sigma_p & 0 \\ 0 & 0 \end{pmatrix} V^T.$$

Let U and V have column partitions:

U = $[u_1 \cdots u_m]$

V = $[v_1 \cdots v_n]$.

From the relations $Av_j = \sigma_j u_j, A^T u_j = \sigma_j v_j, j = 1, \ldots, p$, it follows that $A^T A v_j = \sigma_j^2 v_j$ (and $AA^T u_j = \sigma_j^2 u_j$).

That is, the squares of the singular values are the eigenvalues of $A^T A$, which is a symmetric matrix.

- Source: https://web.stanford.edu/class/cme335/lecture6.pdf.

It is known that if the singular values of a matrix A decrease sufficiently rapidly, then we can hope to determine a good approximation of A with a very low rank.

If the rank is equal to k (with $k \leq \min(m, n)$), then in this case the matrix A is approximated as follows:

$$A \sim U \begin{pmatrix} \Sigma_k & 0 \\ 0 & 0 \end{pmatrix} V^T.$$

The matrix A can thus be obtained from only:

- k vectors u_k each having m coefficients
- k vectors v_k each having n coefficients
- and k singular values σ_k.

For further information on SVD, you can consult the following document: https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/book-chapter-4.pdf.

My implementation of the SVD method relies on the deflation method that I already explained, and which itself relies on the power method that I also explained.

The choice of the deflation method for the SVD algorithm is legitimate, despite its accumulation of errors, because it has a particular impact on the weakest singular values, as well as on the singular vectors associated with them, and thus on the least significant components, carrying the least information.

The SVD method can be used for image compression as I will show.

An image can be viewed as a pixel matrix. Each coefficient of this matrix corresponds to a pixel and its numerical value corresponds, in a defined color coding, to a plain coloring of this pixel. The matrix associated with an image is therefore a real-valued matrix $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ of size $m \times n$, consequantly, it accepts a singular value decomposition.

First, find below my implementation of the deflation and power methods:

In [1]:
```python
import numpy as np
from math import *

def powerMethodAlgo(A, x) :

    # x is a vector from which the initial vector used in the first iteration of the algorithm is built on
    #A is the input matrix of which we want to calculate the highest absolute eigenvalue and the related eigenvector

    vOld = x.copy()/np.linalg.norm(x,2) #vOld is the eigenvector calcultaed at iteration i. At the beginning vOld is the
                                        #vector x divided by its L2 norm
    vNew = A.dot(vOld.copy()) #vNew is the eigenvector calculated at iteration i+1. At the beginning VNew is the product
                              #of the input matrix A and vOld
    eigValOld = float('nan') #eigValOld is the eigenvalue calculated at iteration i
    eigValNew = vOld.copy().T.dot(vNew.copy()) #eigValNew is the eigenvalue calculated at iteration i+1. At the beginning
                                               #it is the product of cOld and vNew

    for i in range(100000): # 100000 is the maximum number of iterations arbitrarily chosen



        if np.isclose(eigValOld, eigValNew): #the shutoff parameter of the algorithm is eigValNew and eighValOld
                                             #to be close enough to each other, which would mean that the algorithm
                                             #has converged to the highest absolute eigenvalue of the matrix
                                             #we return the highest absolute eigenvalue eighValNew and the related
                                             #eigenvector vOld
```

```python
            return eigValNew, vOld

        else:

            #if we enter a new iteration eigValOld is replaced by eigValNew, vOld is replaced by vNew divided by
            #its L2 norm, vNew is replaced by the product of A and the (new) vOld, and eigValNew is replaced by the
            #product of the (new) vOld and the (new) vNew

            eigValOld = eigValNew.copy()
            vOld = vNew.copy()/ np.linalg.norm(vNew.copy(),2)
            vNew = A.dot(vOld)
            eigValNew = vOld.T.dot(vNew)

    #if we don't converge to the highest eigenvalue we return 0
    return  0, 0
```

In [2]:

```python
def deflation(A):


    A = A.copy() #we copy the input matrix
    #we assert that the matrix is squared and we print a message if it is not the case
    assert np.shape(A)[0] == np.shape(A)[1], 'the input matrix must be a square matrix'
    N = np.shape(A)[0] #N = number of lines of the input matrix

    eigValv = np.zeros(N) # we create a vector where the eigenvalues will be stored in ascending order
                          #it contains zeros at the biginning
    eigVecta = np.zeros([N, N]) #we create an array where eigenvectors will be stored. It contains zeros at the beginning
    randX = np.random.rand(N) #randX is a random vector of which the length equals the number of lines
                              #(or number of columns) of the matrix A
    solution = np.zeros(2)    #soution is a vector that will store the output of the powerMethodAlgo function

    #we calculate the N eigenvalues and eigenvectors of A
    for i in range(N):
        solution = powerMethodAlgo(A, randX)
        eigValv[i] = solution[0]
        eigVecta[i] = solution[1]
        #in oder to calculate the next highest absolute eigenvalue, we must perform this operation
        A = A - eigValv[i]*np.outer(eigVecta[i], eigVecta[i]).T/np.linalg.norm(eigVecta[i], 2)

        #we return the eigenvalues and the eigenvectors, please note that the eigenvectors must be read vertically
        #not horizontally as eigVecta is transposed before being returned

    return eigValv, eigVecta.T
```

Below, my (basic) implementation of the SVD method with some comments to ease the understanding.

In [3]:

```python
def SVD(A):
    A = A.copy()
    sigmaInv = np.zeros([np.shape(A)[1], np.shape(A)[0]]) #to store the values of the inverse
                                                          #of the singular values matrix Sigma

    if np.shape(A)[0] >= np.shape(A)[1]: #if the input matrix has more lines than columns
        p = np.shape(A)[1]

    else: #if the input matrix has more columns than lines
        p = np.shape(A)[0]

    H = A.dot(A.T) #symetric matrix obtained from the input matrix

    sigma = np.zeros(p) #the matrix of singular values
    sqSingVal, U = deflation(H) #the deflation function will return the squared singular values of the input matrix
                                #as well as the matrix U

    for i in range(p):
        sigma[i] = sqrt(abs(sqSingVal[i]))
        sigmaInv[i][i] = 1/sigma[i]

    #we can now easily build the matrix V
    V = sigmaInv.dot(U.T).dot(A)
    #we return the matrices U, V^T and the vector sigma which contains the singular values of the input matrix
    return U, sigma, V.T
```

Below, the image_compression function which performs a partial singular value decomposition according to the rank passed in parameter and returns the approximation of the (pixel) matrix A.

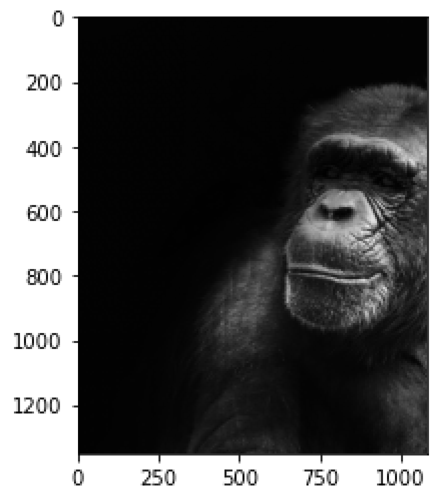
In [4]:

```python
def image_compression(A, rank):

    if rank > min([np.shape(A)[1], np.shape(A)[0]]):
        print("the rank can't be greater than the smallest dimension of the matrix")

        return


    A = A.copy()
    sigmaInv = np.zeros([np.shape(A)[1], np.shape(A)[0]])
    sigma = np.zeros([np.shape(A)[0], np.shape(A)[1]])
    H = A.dot(A.T)


    sqSingVal, U = deflation(H)


    for i in range(rank):

        sigma[i][i] = sqrt(abs(sqSingVal[i]))
        sigmaInv[i][i] = 1/sigma[i][i]


    V = A.T.dot(U).dot(sigmaInv.T)

    return U.dot(sigma).dot(V.T)
```

Let's use it now with an image.

In [5]:
```
import matplotlib.pyplot as plt
img = plt.imread('monkey.png')
monkey = img.T[0].T
plt.imshow(monkey, cmap="gray")

print("Dimensions of the pixel matrix:")
print(str(np.shape(monkey)[0])+"x"+str(np.shape(monkey)[1]))
```

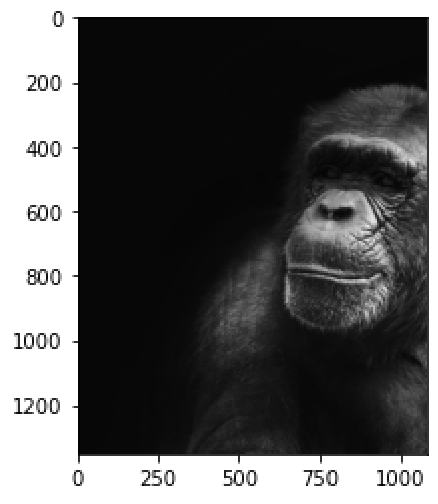
```
Dimensions of the pixel matrix:
1350x1080
```



Let's try with rank = 400.

In [6]:
```
image = image_compression(monkey, 400)
plt.imshow(image, cmap="gray")
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x203f35bf2b0>
```



The compressed image is hardly distinguishable from the original. The number of coefficients needed for its construction is 400 x $1350 + 400$ x $1080 + 400 = 972400$, instead of 1350 x $1080 = 1458000$, so the difference is very significant (486000).

In [ ]: