Project 5 - Image Compression

Thales Mateus Rodrigues Oliveira (RA 148051)

***Abstract*—In this project, it was given the task of implementing an image compression algorithm, using mathematical formulations. To fulfill the requirements, the solution uses the Principal Component Analysis (PCA) approach. It was able to generate the compressed images for some cases, with some degree of information loss. The details of the implementation are explained in this report.**

I. Introduction

Principal Component Analysis (PCA) is a technique used for reducing the dimensionality of data, by applying Single Value Decomposition (SVD) to the image matrix, and selecting a reduced number of axis components. In this work, the implementation of the technique was done, and tests performed, changing the number of selected components and evaluating by quantitative indicators. The next sections explains the implemented algorithm, the experiments realized and the output analysis.
The code, along with the input files is delivered in the compressed file THALES_MATEUS_RODRIGUES_OLIVEIRA_148051.zip, in the Google Classroom. The report is sent via e-mail.

II. The Program

The program was implemented with Python 3.7.3. The libraries used and their respective versions are OpenCV 4.1.0 and Numpy 1.16.4.

A. How to execute it

The project has a Makefile available to help performing some actions on it. The Makefile has 3 basic instructions: clean, build and exec. Clean instruction removes generated images stored in the **output** folder, the execution code in **bin** folder and the folders itself. The Build instruction creates the **output** and **bin** folders, and moves the source code to **bin**. The Exec instruction executes the code with images in the **input** folder. Listing 1 provides examples of how to execute the three instructions in a terminal.

```
#clean environment, deletes output and bin folders and their content
make clean

#prepare the environment for code execution
make build

#executes code
make exec
```

Listing 1. Makefile usage example

B. Input

The program does not have an input argument by default, the input images are listed in code, and they are expected to be stored in the **input** folder. Listing 2 shows how images are listed to be executed in code. The $images$ tuple is implemented in $src/main.py$.

```
# for inserting other images, add tem to /input folder and list them here
images = (
    'baboon',
    'monalisa',
    'peppers',
    'watch'
)
```

Listing 2. Input images inside code

C. Output

The output of the program is a series of compressed images based on the input ones, changing the number of selected components from the SVD. The output images are stored in the **output** folder, and they are labeled by concatenating the image name and the number of selected components (*e.g.: baboon_10.png*)

D. Implementation

The function which implements the compression operation is defined in the *src/compress.py* file, and it is named *apply_svd()*. As explained in the project description[1], as the result of the SVD gives the matrix of eigenvalues in the decreasing order, the first elements of the eigenvalues and eigenvectors matrices contributes more to the final result of the image (has higher contribution in the image formation). Therefore, keeping only few of these elements may result in images with lower quality, which requires less capacity of storage. The function receives the original image, and the number of components to be kept. Listing 3 shows keyparts of implementation. It was inspired by Algorithm I from the project description. As we can see, it applies the SVD for every single color channel, selects the desired number of components from each matrix, and then builds the image back.

```
def apply_svd(img, n_components):
  """
  It applies image compression method using SVD decomposition.

  Keyword arguments:
  img -- the image to be compressed
  n_components -- the number of svd components to be considered
  """

  # initialize result array
  result = np.zeros_like(img.astype(np.double))
```

ra148051@students.ic.unicamp.br

```
    # do the SVD for every color channel separately
    w_b, u_b, vt_b = np.linalg.svd(img[:, :, 0].astype
      (np.double))
    w_g, u_g, vt_g = np.linalg.svd(img[:, :, 1].astype
      (np.double))
    w_r, u_r, vt_r = np.linalg.svd(img[:, :, 2].astype
      (np.double))

    # transforms U to matrix form
    u_b = np.diag(u_b)
    u_g = np.diag(u_g)
    u_r = np.diag(u_r)

    # separates the desired number of components
    w_result_b = w_b[:, 0:n_components]
    w_result_g = w_g[:, 0:n_components]
    w_result_r = w_r[:, 0:n_components]
    u_result_b = u_b[0:n_components, 0:n_components]
    u_result_g = u_g[0:n_components, 0:n_components]
    u_result_r = u_r[0:n_components, 0:n_components]
    vt_result_b = vt_b[0:n_components, :]
    vt_result_g = vt_g[0:n_components, :]
    vt_result_r = vt_r[0:n_components, :]

    # builds the compressed image by multiplying the
      desired components
    result[:, :, 0] = w_result_b.dot(u_result_b.dot(
      vt_result_b))
    result[:, :, 1] = w_result_g.dot(u_result_g.dot(
      vt_result_g))
    result[:, :, 2] = w_result_r.dot(u_result_r.dot(
      vt_result_r))

    return result
```

Listing 3. Implementation of Compression with PCA

1) Evaluating the Compression: With the objective of evaluating the compression, two quantitative indicators were implemented: the compression rate, and the root mean-squared error. The formulations of both indicators are given in the project description[1]. Listing 4 shows the implementation. After calculating the values, their are printed in the standard output.

```
def evaluate_compression(original, compressed, name,
     n_components):
    """
    It quantitatively evaluates a compression, by
      calculating the compression rate and the Root
      Mean-Squared Error (RMSE)

    Keyword arguments:
    original -- the original image (numpy array)
    compressed -- the compressed image (numpy array)
    name -- the image name
    n_components -- the number of components
      considered for the compression
    """
    original_size = os.path.getsize('input/' + name +
      '.png')
    compressed_size = os.path.getsize('output/' + name
      + '_' + str(n_components) + '.png')
    # calculates the compression rate by dividing the
      storage sizes
    compression_rate = compressed_size/original_size
    # Calculates RMSE
    rmse = math.sqrt((((original - compressed)**2).sum
      ())/(original.shape[0]*original.shape[1]))
    print("\t\t" + "Compression rate: " + str(
      compression_rate) + " RMSE:" + str(rmse))


```

Listing 4. Evaluation of Compressed Images

III. Experiments

The *src/main.py* file executes the test pipeline. The idea is the following: for each input image, for number of components, generate the compressed versions and evaluate the result. The input images are stored in **input** folder as mentioned before, and their names and dimensions are listed in table I. The input images are shown in figure 1

Image Names	Dimensions (width x height)
baboon.png	512 x 512
monalisa.png	256 x 256
peppers.png	512 x 512
watch.png	1024 x 768

TABLE I
INPUT IMAGES USED IN EXPERIMENTS

Fig. 1. Input images used in experiments. a) Baboon. b) Monalisa c) Peppers d) Watch

The number of components used were taken from the example given by the project description[1]. Table II lists the used values.

Number of components used in tests
1
5
10
20
30
40
50

TABLE II
NUMBER OF COMPONENTS USED IN TESTS

As we have 4 input images and 7 different component values, we have 28 images of output. The output images are stored in the **output** folder.

IV. Discussion

This section is organized in two parts. The first part analyses the output images qualitatively. The second one checks the implemented indicators.

A. Checking the output images

In order to analyze the compressed images, the comparison between input and output images is needed. The figures 2, 3, 4, 5 places input and output images together, for all the input images used in the experiments, showing their versions with the selected components.

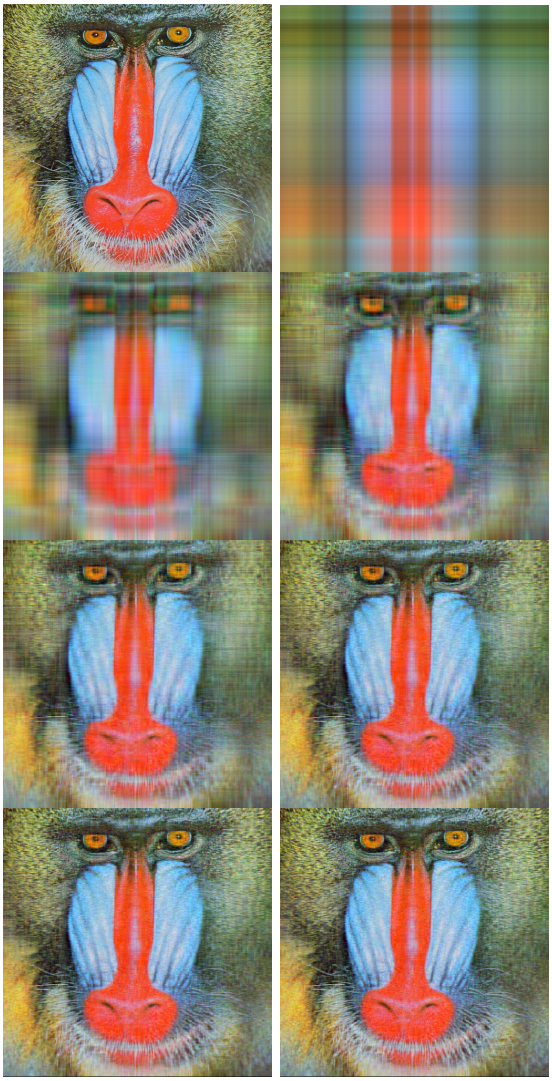

Fig. 2. Baboon images manipulated in experiments. a) Input. b) Compressed with 1 component. c) Compressed with 5 components. d) Compressed with 10 components e) Compressed with 20 components f) Compressed with 30 components g) Compressed with 40 components h) Compressed with 50 components

From the generated images, it is observable the loss of quality created by the process. Even though the greater amount of information is stored in the first component, the image details (high frequency information) is lost from the less significant ones. The higher the dimension and the entropy of the image, the greater is the amount of information loss, because it has more components and more high frequency information. Therefore, the least impacted image is the monalisa one, in which in the last case (50 components), almost half of the components is kept. The most impacted one is the baboon image.

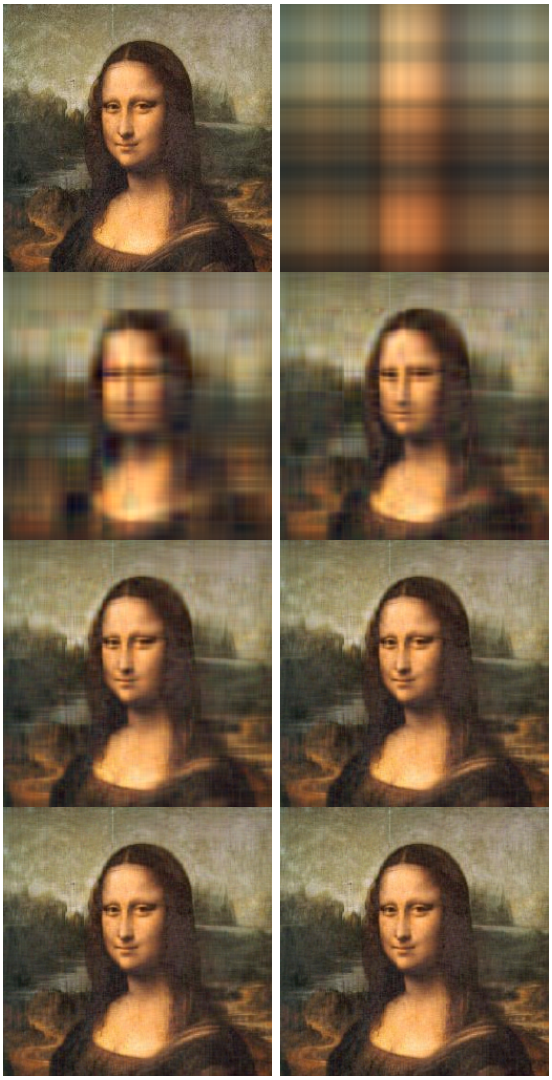

Fig. 3. Monalisa images manipulated in experiments. a) Input. b) Compressed with 1 component. c) Compressed with 5 components. d) Compressed with 10 components e) Compressed with 20 components f) Compressed with 30 components g) Compressed with 40 components h) Compressed with 50 components

B. Indicators Results

In order to analyze the effectiveness of the compression method, the compression rate and RMSE indicators were implemented. The result for all input images are shown in the listings 5, 6, 7 and 8.

```
baboon image:
  1 component(s):
    Compression rate: 0.4606994438097151 RMSE
    :75.38358915100255
  5 component(s):
    Compression rate: 0.605770317267009 RMSE
    :53.71495530921579
  10 component(s):
    Compression rate: 0.6823673241346407 RMSE
    :48.29679156656673
  20 component(s):
    Compression rate: 0.7680416577734811 RMSE
    :43.11258403464472
  30 component(s):
    Compression rate: 0.8153649135582368 RMSE
    :39.44550826386878
```

Fig. 4. Peppers images manipulated in experiments. a) Input. b) Compressed with 1 component. c) Compressed with 5 components. d) Compressed with 10 components e) Compressed with 20 components f) Compressed with 30 components g) Compressed with 40 components h) Compressed with 50 components

```
  40 component(s):
    Compression rate: 0.8499196474532009 RMSE
    :36.36220234788848
  50 component(s):
    Compression rate: 0.8746861228640661 RMSE
    :33.656196118816396
```

Listing 5. Indicators Results for Baboon Image

For the baboon image, we can see that the quality of the image improves as the number of components increases. In the last case, the compression rate is of 87%, which shows considerable storage economy, but with a high decrease of quality, shown by the value of the RMSE.

```
monalisa image:
  1 component(s):
    Compression rate: 0.6481713867848727 RMSE
    :58.78760860070345
  5 component(s):
    Compression rate: 0.851562288407197 RMSE
    :26.41430931357357
  10 component(s):
```

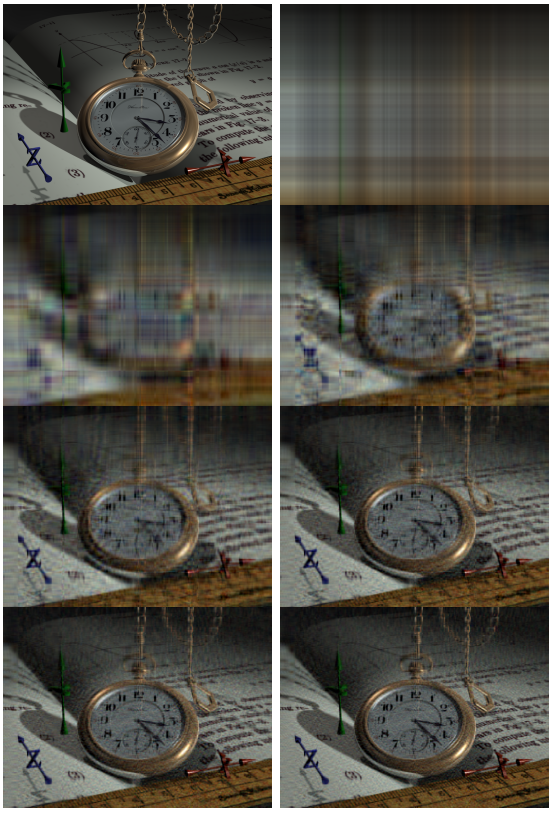

Fig. 5. Watch images manipulated in experiments. a) Input. b) Compressed with 1 component. c) Compressed with 5 components. d) Compressed with 10 components e) Compressed with 20 components f) Compressed with 30 components g) Compressed with 40 components h) Compressed with 50 components

```
    Compression rate: 0.9292930204844403 RMSE
    :18.18466943405659
  20 component(s):
    Compression rate: 1.0013361380194463 RMSE
    :13.001416726457146
  30 component(s):
    Compression rate: 1.0442279740355882 RMSE
    :10.61092785784305
  40 component(s):
    Compression rate: 1.075094567876714 RMSE
    :9.060191683551615
  50 component(s):
    Compression rate: 1.0990186607924743 RMSE
    :7.8451398862882264
```

Listing 6. Indicators Results for Monalisa Image

In the case of the monalisa image, we can see an example of how the method does not necessarily work for all images. With more than 20 components, the storage needed for the compressed image is higher than the original input, which shows some failure in the methodology. Also, even though it requires more space, the quality is affected, as some details of the image are lost. With 10 components, the objective is achieved, even though the economy is not as considerable as the quality loss.

```
peppers image:
  1 component(s):
    Compression rate: 0.4759099320834006 RMSE
    :89.7798662152033
  5 component(s):
```

```
    Compression rate: 0.6391102350073967 RMSE
    :51.023487322642914
  10 component(s):
    Compression rate: 0.7124208119179803 RMSE
    :37.19033543015182
  20 component(s):
    Compression rate: 0.780881263816731 RMSE
    :25.239017029794333
  30 component(s):
    Compression rate: 0.815639564992232 RMSE
    :19.7246540299187
  40 component(s):
    Compression rate: 0.8768201889933902 RMSE
    :16.34744383844143
  50 component(s):
    Compression rate: 0.8874021111872133 RMSE
    :14.066020558078767
```

Listing 7. Indicators Results for Peppers Image

For the pappers image, we also have storage economy, with measurable quality loss. The method works better here, as there are less high-frequency information in this case. Still, the RMSE value could be a problem for some applications.

```
watch image:
  1 component(s):
    Compression rate: 0.7172106688702199 RMSE
    :57.18398114099044
  5 component(s):
    Compression rate: 1.014637274480099 RMSE
    :40.89070190048771
  10 component(s):
    Compression rate: 1.1777633360877748 RMSE
    :34.91354161920759
  20 component(s):
    Compression rate: 1.3540734173438003 RMSE
    :28.577257735650306
  30 component(s):
    Compression rate: 1.4670528623238306 RMSE
    :24.913312768720537
  40 component(s):
    Compression rate: 1.569728974429601 RMSE
    :22.31593967903772
  50 component(s):
    Compression rate: 1.6414333998990038 RMSE
    :20.192015693023528
```

Listing 8. Indicators Results for Watch Image

For the watch image, the method fails to achieve its objective. The storage required dramatically increases when more components are selected, to the point it surpasses the original image size in the case of only 5, with poor image quality, as shown by the high RMSE.

V. Conclusion

The application of the PCA technique to generate compressed versions of images did not succeed for all types of images. For images with high-frequency components, the loss of quality in the output images might be more than expected. The method does not guarantee to decrease the storage requirements for all images.

References

[1] H. Pedrini, “Introduction to digital image processing, project 5 description,” 2019.