Worksheet 02: Introduction to NumPy and its application in image manipulation

Part one: numpy routines:

1 *Import numpy and get documentations*

- Write the instructions to import the NumPy module and alias it as np.
- How to get help on a NumPy function?

2 *Array creation (shapes and types)*

- Create a 1D NumPy array containing integers from 0 to 9.
- Create a 2D NumPy array containing integers from 0 to 15 arranged in a 4x4 grid.
- Create a 1D NumPy array containing the values 2, 4, 6, 8, and 10 using the `np.arange()` function.
- Create a 3D NumPy array containing random floating-point numbers between 0 and 1 with shape (2, 3, 4).
- Create a 1D NumPy array of length 5 containing all zeros.
- Create a 1D NumPy array of length 8 containing all ones.
- Create a 2D NumPy array with shape (3, 2) containing all zeros.
- Create a 2D NumPy array with shape (4, 3) containing all ones.
- Create a 1D NumPy array containing 30 equally spaced values between 0 and 1.
- Create a 4x4 2D array with ones on the diagonal and zeroes elsewhere.
- Create a 5x2 array of flaot numbers, filled with 7.
- Let `x = np.arange(7, dtype=np.int64)`. Create an array of `8` with the same shape and type as X.
- Show the documentation of the following NumPy functions then try to use it, `np.asscalar`,`np.asarray`, `np.asmatrix`.
- Let `x = np.array([1, 2, 3])`. Create an array copy of x, which has a different id from x.

- Create a 1-D array of 50 element spaced evenly on a log scale between 3. and 10. .
- Use the function `np.diagflat` to create a matrix with the diagonal `[1,2,3,4,5,6,7]`.
- Use the function `np.tri`; `np.tril` and `np.triu` to create the following arrays.

  -
    ```
    array([[ 0., 0., 0., 0., 0.],
           [ 1., 0., 0., 0., 0.],
           [ 1., 1., 0., 0., 0.]])
    ```

  -
    ```
    array([[ 0,  0,  0],
           [ 4,  0,  0],
           [ 7,  8,  0],
           [10, 11, 12]])
    ```

  -
    ```
      array([[ 1,  2,  3],
             [ 4,  5,  6],
             [ 0,  8,  9],
             [0, 0, 12]])
    ```

3 *Array manipulation*

- Let x be a `ndarray [10, 10, 3]` with all elements set to one. Reshape x so that the size of the second dimension equals 150.
- Execute the following instruction then guess what the negative index do:
  - `np.reshape(x, [-1,3, 25])`
  - `np.reshape(x, [-1, 50])`
- Let x be array `[[ 0,  1], [ 2,  3], [ 4,  5], [ 6,  7], [ 8,  9], [10, 11]]`. Convert it to `[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`.
- Execute the following instructions then guess what the `np.squeeze` do

```
x=np.ones((3,1,2))
print(x)
np.squeeze(x)
```

- What is the instruction that transpose a matrix?

Solutions

1 *Import numpy and get documentations*

To import the NumPy module and alias it as np, you can use the following statement:

```
import numpy as np
```

This will allow you to access all the functions and classes in NumPy by prefixing them with np.

To get help on a NumPy function, you can use the `np.info()` function or the ? operator in the Python console. For example, to get help on the np.arange() function, you can type:

```
np.info(np.arange)
```

or

```
np.arange?
```

Both of these commands will display the documentation for the np.arange() function, including its parameters, return value, and usage examples. - ## *Array creation (shapes and types)*

2 *Array creation (shapes and types)*

```
import numpy as np

# Create a 1D NumPy array containing integers from 0 to 9.
arr1d = np.arange(10)
print(arr1d)

# Create a 2D NumPy array containing integers from 0 to 15 arranged in a 4x4 grid.
arr2d = np.arange(16).reshape((4, 4))
print(arr2d)

# Create a 1D NumPy array containing the values 2, 4, 6, 8, and 10 using
 the np.arange() function.
arr1d_custom = np.arange(2, 11, 2)
print(arr1d_custom)

# Create a 3D NumPy array containing random floating-point numbers between 0
  and 1 with shape (2, 3, 4).
arr3d = np.random.rand(2, 3, 4)
print(arr3d)

# Create a 1D NumPy array of length 5 containing all zeros.
arr_zeros_1d = np.zeros(5)
print(arr_zeros_1d)

# Create a 1D NumPy array of length 8 containing all ones.
arr_ones_1d = np.ones(8)
print(arr_ones_1d)

# Create a 2D NumPy array with shape (3, 2) containing all zeros.
arr_zeros_2d = np.zeros((3, 2))
print(arr_zeros_2d)

# Create a 2D NumPy array with shape (4, 3) containing all ones.
arr_ones_2d = np.ones((4, 3))
print(arr_ones_2d)

# Create a 1D NumPy array containing 30 equally spaced values between 0 and 1.
arr_linspace = np.linspace(0, 1, 30)
print(arr_linspace)
```

```
# Create a 4x4 2D array with ones on the diagonal and zeroes elsewhere.
arr_eye = np.eye(4)
print(arr_eye)

# Create a 5x2 array of float numbers, filled with 7.
arr_custom = np.full((5, 2), 7.0)
print(arr_custom)

# Let x = np.arange(7, dtype=np.int64). Create an array of 8 with the
 same shape and type as X.
x = np.arange(7, dtype=np.int64)
arr_same = np.ones_like(x)*8
print(arr_same)

# Show the documentation of the following NumPy functions then try to
 use it, np.asscalar,np.asarray, np.asmatrix.
print(np.info(np.asscalar))
print(np.info(np.asarray))
print(np.info(np.asmatrix))

# Let x = np.array([1, 2, 3]). Create an array copy of x, which has a different
  id from x.
x = np.array([1, 2, 3])
arr_copy = np.copy(x)
print("id of x is",id(x))
print("id of the copy of x is",id(arr_copy))

# Create a 1-D array of 50 element spaced evenly on a log scale between 3. and 10.
arr_logspace = np.logspace(3, 10, 50)
print(arr_logspace)

# Use the function np.diagflat to create a matrix with the
 diagonal [1,2,3,4,5,6,7].
arr_diag = np.diagflat([1, 2, 3, 4, 5, 6, 7])
print(arr_diag)

# Use the function np.tri, np.tril and np.triu to create the following arrays.
# array([[ 0., 0., 0., 0., 0.],
#        [ 1., 0., 0., 0., 0.],
#        [ 1., 1., 0., 0., 0.]])
arr1 = np.tril(np.ones((3, 3)), -1)

# array([[ 0,  0,  0],
#        [ 4,  0,  0],
#        [ 7,  8,  0],
#        [10, 11, 12]])
arr2 = np.tril(np.arange(1, 13).reshape(4, 3), -1)
```

```
# array([[ 1,  2,  3],
#        [ 4,  5,  6],
#        [ 0,  8,  9],
#        [ 0,  0, 12]])
arr3 = np.triu(np.arange(1,13).reshape(4,3), -1)
print(arr1,"\n",arr2,"\n",arr3)
```

3 *Array manipulation*

- Let x be a ndarray [10, 10, 3] with all elements set to one. Reshape x so that the size of the second dimension equals 150.

```
x = np.ones((10, 10, 3))
x = np.reshape(x, (2, 150))
```

- Execute the following instruction then guess what the negative index do:
  - `np.reshape(x, [-1,3, 25])`: Reshapes x into a new 3D array with the first dimension determined automatically and the second and third dimensions set to 3 and 25, respectively. The negative first dimension argument means that its size is inferred from the input size and the other two dimensions.
  - `np.reshape(x, [-1, 50])`: Reshapes x into a new 2D array with the first dimension determined automatically and the second dimension set to 50. The negative first dimension argument means that its size is inferred from the input size and the second dimension.

- Let x be array `[[ 0, 1], [ 2, 3], [ 4, 5], [ 6, 7], [ 8, 9], [10, 11]]`. Convert it to `[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`.

```
x = np.array([[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11]])
x = np.ravel(x) #Or x.flatten()
print(x)
```

- Execute the following instructions then guess what the `np.squeeze` do.

```
x = np.ones((3, 1, 2))
print(x)
np.squeeze(x)
```

- The `np.squeeze` function removes dimensions of size 1 from an array. In the given example, `x` is a 3D array with dimensions (3, 1, 2). The second dimension has size 1, so `np.squeeze(x)` removes it and returns a 2D array with dimensions (3, 2).
- The NumPy instruction to transpose a matrix is `numpy.transpose` or simply `ndarray.T`. For example, if you have a 2D NumPy array `A`, you can transpose it using `A.T`.

Part Two: Using numpy arrays' operations and image manipulation (application):

1. To begin, let us load the image into numpy. This can be done by using the imread() function from the matplotlib library. This function allows numpy to read graphic files with different extensions. The output is a two-dimensional array with the dimensions equal to the dimensions of the image, and the values corresponding to the colors of the pixels. Here we will work with a grayscale image, so the elements in the array will be integers ranging from 0 to 255 in the numpy integer format uint8. Type in the following code to load the file "fibonacci.jpg" into numpy:

   ```
   from matplotlib import pyplot as plt
   ImJPG = plt.imread('fibonacci.jpg')
   #you can get the image from the link
     https://ydjemmada.github.io/fibonacci.jpg
   ```

   The array ImJPG is a two-dimensional array of the type uint8 which contains values from 0 to 255 corresponding to the color of each individual pixel in the image, where 0 corresponds to black and 255 to white. You can visualize this array by printing it in the console:

   ```
   print(ImJPG) #prints the array values
   plt.imshow(ImJPG,cmap='gray')#shows the image
   ```

2. Use the shape attribute to check the dimensions of the obtained array ImJPG:

   ```
   m, n = ImJPG.shape
   ```

   1. What are the dimensions of the image?

3. Check the type of the array ImJPG by using the dtype attribute: ImJPG.dtype The output of the dtype attribute is a numpy data type.

4. Find the range of colors in the image by using the amin and amax functions and save those elements as maxImJPG and minImJPG:

   ```
   maxImJPG = np.amax(ImJPG)
   minImJPG = np.amin(ImJPG)
   ```

5. Finally, display the image on the screen by using imshow:

   ```
   plt.imshow(ImJPG, cmap='gray')
   ```

   If you did everything correctly, you should see the image displayed on your screen in a separate window.

6. To crop the image in numpy, we can select a subarray from the original array `ImJPG`. The rows and columns we want to keep from the original array can be specified using indexing. The following code will select the central part of the image leaving out 100 pixels from the

top and bottom, and 100 pixels on the left and 70 pixels on the right, and display the result using matplotlib:

```
ImJPG_center = ImJPG[100:m-100, 100:n-70]
import matplotlib.pyplot as plt
plt.imshow(ImJPG_center, cmap='gray')
plt.show()
```

This will create a new figure window displaying the cropped image.

7. We can paste the selected part of the image into another image. To do this, create a zero matrix using the command:

   ```
   ImJPG_border = np.zeros((m, n), dtype=np.uint8)
   ```

   Then paste the preselected matrix ImJPG_center into matrix ImJPG_border and display the image:

   ```
   ImJPG_border[100:m-100, 100:n-70] = ImJPG_center
   plt.figure()
   plt.imshow(ImJPG_border, cmap='gray')
   ```

   Notice the use of the data type np.uint8. It is necessary to use this data type because by default the array will be of the type float, and imshow command does not work correctly with this type of array.

8. To flip the image vertically using NumPy, we can use the flipud function:

   ```
   ImJPG_vertflip = np.flipud(ImJPG)
   plt.imshow(ImJPG_vertflip, cmap='gray')
   ```

   This will create a new array ImJPG_vertflip that is a vertically flipped version of the original array ImJPG.

9. To transpose the matrix using NumPy, we can use the transpose attribute:

   ```
   ImJPG_transpose = ImJPG.transpose()
   plt.imshow(ImJPG_transpose, cmap='gray')
   ```

10. To flip the image horizontally using NumPy, we can combine the transpose attribute and the fliplr function:

    ```
    ImJPG_horflip = np.fliplr(ImJPG_transpose).transpose()
    plt.imshow(ImJPG_horflip, cmap='gray')
    ```

11. To rotate the image by 90 degrees using NumPy, we can use the rot90 function:

    ```
    ImJPG90 = np.rot90(ImJPG)
    plt.imshow(ImJPG90, cmap='gray')
    ```

12. Execute the following numpy commands:

```
ImJPG_inv = 255-ImJPG
plt.imshow(ImJPG_inv)
plt.show()
```

Display the resulting image using matplotlib's imshow function in a new figure window. Note that the constant 255 is subtracted from the array ImJPG, which mathematically does not make sense. However, in numpy, the constant 255 is treated as an array of the same size as ImJPG with all the elements equal to 255. Explain what happened to the image.

13. It is also easy to lighten or darken images using matrix addition. For instance, the following code will create a darker image:

```
ImJPG_dark=np.clip(np.array(ImJPG, dtype='int16') - 50, 0, 255)
plt.imshow(ImJPG_dark,cmap='gray')
plt.show()
```

You can darken the image even more by changing the constant to a number larger than 50. Note that this command can technically make some of the elements of the array to become negative. However, because the ImJPG array type is int16, with the function clip those elements are automatically rounded to zero.

14. Let us create Andy Warhol style art with the image provided. To do so we will arrange four copies of the image into a 2×2 matrix. For the top left corner we will take the unaltered image. For the top right corner we will darken the image by 50 shades of gray. For the bottom left corner, lighten the image by 100 shades of gray, and finally, for the bottom right corner, lighten the image by 50 shades of gray. Then we will arrange the images together in one larger matrix using numpy's concatenation function. Finally, display the resulting block matrix as a single image using matplotlib's imshow function.

```
im1 = ImJPG
im2 = np.clip(np.array(ImJPG, dtype='int16') - 50, 0, 255)
im3 = np.clip(np.array(ImJPG, dtype='int16') + 100, 0, 255)
im4 = np.clip(np.array(ImJPG, dtype='int16') + 50, 0, 255)
row1 = np.concatenate((im1, im2), axis=1)
row2 = np.concatenate((im3, im4), axis=1)
ImJPG_warhol = np.concatenate((row1, row2), axis=0)
plt.imshow(ImJPG_warhol,cmap='gray')
plt.show()
```

15. Numpy has several functions which allow one to round any number to the nearest integer or a decimal fraction with a given number of digits after the decimal point. Those functions include: floor which rounds the number towards negative infinity (to the smaller value), ceil which rounds towards positive infinity (to the larger value), round which rounds towards the nearest decimal or integer, and fix which rounds towards zero.

A naive way to obtain black and white conversion of the image can be accomplished by making all the gray shades which are darker or equal to a medium gray (described by a value 128) to appear as a complete black, and all the shades of gray which are lighter than this medium gray to appear as white. This can be done, for instance, by using the code:

```
ImJPG_bw = np.uint8(255*np.floor(ImJPG/128))
plt.imshow(ImJPG_bw, cmap='gray')
plt.show()
```

Note that this conversion to black and white results in a loss of many details of the image. There are possibilities to create black and white conversions without losing so many details. Also, notice the function np.uint8 used to convert the result back to the integer format.

16. Write code to reduce the number of shades in the image from 256 to 8 using the round function. Save the resulting array as 'ImJPG8' and display it in a separate window.

```
ImJPG8 = np.round(ImJPG / 32)
plt.imshow(ImJPG8.astype('uint8'), cmap='gray')
plt.show()
```

17. Increase the contrast of the image by changing the range of possible shades of gray. One way to do this is to scalar multiply the array by a constant. Use the following code:

```
ImJPG_HighContrast = np.clip((1.25 * ImJPG),0,255)
plt.imshow(ImJPG_HighContrast, cmap='gray')
plt.show()
```

Observe the result by displaying the image. You can manipulate the contrast by increasing or decreasing the constant (we use 1.25 in this case). Note that this operation may cause some elements of the array to become outside the 0-255 range, potentially leading to data loss. Save the resulting array as 'HighContrast'.

18. Apply gamma correction to the image using the following code:

```
ImJPG_Gamma05 = np.clip(ImJPG** 0.95,0,255)
plt.imshow(ImJPG_Gamma05, cmap='gray')
plt.show()
ImJPG_Gamma15 = np.clip(ImJPG ** 1.15,0,255)
plt.imshow(ImJPG_Gamma15, cmap='gray')
plt.show()
```

Observe the results by displaying the images. The above code will produce two images, one with gamma equal to 0.95 (ImJPG_Gamma05) and one with gamma equal to 1.05 (ImJPG_Gamma15). Gamma correction is a nonlinear operation that can be used to adjust the brightness and contrast of an image.