Exercise 2 + Tutorial

(Total: 30 points)

In this exercise, you will practice your NumPy skills, and also learn about Matplotlib – which is Python's fundamental plotting library. This exercise will therefore also serve as a short tutorial for Matplotlib.

NumPy for Image Manipulation

There are several uses for high-dimensional arrays in data analysis. They can be used to store:

- matrices, solve systems of linear equations, find eigenvalues/vectors, find matrix decompositions, and solve other problems familiar from linear algebra
- multi-dimensional measurement data. For example, an element an 2D array a[i,j] might store the temperature t_{ij}, where i might be location and j is date
- images and videos can be represented as NumPy arrays:
    - a gray-scale image can be represented as a 2D array
    - a color-image can be represented as a 3D image, the third dimension contains the color components red, green, and blue
    - a color video can be represented as a 4D array

In this exercise we will work with color image data, and manipulate the data to produce manipulated image data.

Why would we want to do this?

One ML application is of image manipulation is for **data augmentation**. It is a technique of altering the existing data to create some more data for the model training process. In other words, it is the process of artificially expanding the available dataset for training a deep learning model in order to improve the performance and ability of the model to generalize.

We augment our available dataset by augmenting it. It is not always possible for us to obtain more training data in some cases (it could be very time-consuming or very expensive), and yet we know complex ML models will perform better with more data. First, training with small data can lead to overfitting and subsequently poor results on test data, and low generalizability. The data augmentation technique has been found to produce good external generalization.

Caveat: Of course, nothing beats the ability to have more data, but augmentation can be a second-best solution. The final goal of designing any ML system is for the model to be good at production. So excessive usage of data augmentation can be challenging in the production environment (can cause inefficiencies). If your dataset is really small you should first contemplate on the type of model you are going to use. It is meaningless to apply classical heavy neural network to the few samples as its optimizer was not designed for low data setup. In such cases, it is recommended to try simpler ML models or use neural network approaches designed specifically to solve low-data problems (zero-shot, one-shot, few-shot learning and meta learning models).

Let's carry on to see how we can perform data augmentation using NumPy and Matplotlib.

A. What is an image made of?

Take a 2-D image. While there are many formats, I will only discuss a jpeg/jpg or png file below. Each component in an image – i.e. a pixel - can be represented by 3 dimensions of data:

- the position on the row axis,

- the position on the column axis and

- the colors (described using 3 channels – RGB).

The color is described by a combination of 3 channel values –levels of red, green, blue, or RGB. A color value of 0 is absence of color, whereas a maximum color value of (1.0 or 255) represents the highest level of intensity. If all 3 RGB are 0.0, we get black; if all 3 RGB values are 1.0, we get white. Sometimes images contain a fourth piece of data on the 3rd axis – the opacity or depth of color. The fourth piece of information, if present, is known as an *alpha channel* and enables the use of partial transparency by specifying a level of opacity. We will refer to an RGB image with an alpha channel as an RGBA image. An alpha value of 0 is fully transparent, and the maximum value for the pixel depth (1.0 or 255) is completely opaque.

PNG images use these 4 pieces of data - RGBA, whereas JPG format uses the first 3- RGB - to represent the 3rd dimension.

Each level of color is either a float (0.0 to 1.0) for PNG images, or a 1-byte unsigned integer for JPG images – which means 8 bits are used to represent each color. The total number of values that can be represented in 1 byte is 2^8 - 1 or 255 (since we start counting from 0).

Now let's think about how we can represent an image in a programming language such as Python. In NumPy, this is just a 3-D ndarray. The first dimension gives us the values of rows (y axis), and the second dimension gives us values of the columns (x axis). The third dimension provides the color (and opacity) of each pixel as a (3,) vector for JPG or (4,) vector for PNG.

Manipulating an image now is now a matter of using NumPy functions to manipulate the ndarray.

Today many image packages are available that are built on top of and extend NumPy functionalities, including Scikit-image, OpenCV, Pillow, among others. In this exercise, we will only use NumPy to manipulate images to help us practice our NumPy (array manipulation) skills.

An example to understand some indexing basics

Let's use an example of a 4X4 image called ***MyImage***– each pixel is a cell in the table below.

MyImage	Col=0	Col=1	Col =2
Row=0	[R1, G1, B1, A1]	[R2, G2, B2, A2]	[R3, G3, B3, A3]
Row=1	[R4, G4, B4, A4]	[R5, G5, B5, A5]	[R6, G6, B6, A6]
Row=2	[R7, G7, B7, A7]	[R8, G8, B8, A8]	[R9, G9, B9, A9]

row is the 1st axis, col is the 2nd axis and colors/opacity form the 3rd axis. Dimensions of this array are (3,3,4).

Each pixel is accessed as [row, col], which will give us 4 values [R, G, B, A] for that pixel.

To access specific color values within each pixel's third axis, we can use slicing or fancy indexing.

To access the red value in [R8, G8, B8, A8] we can type MyImage[2, 1, :1] or MyImage[2, 1, [0]]

To access the green and blue in [R1, G1, B1, A1] we can type MyImage[0, 0, 1:3] or MyImage[0, 0, [1,2]]

B. Image Manipulation Tasks

Task 0. Housekeeping (1 point)

Here you will write code corresponding to each task in a Python Notebook named as **E2_yourname.ipynb**.

Make sure to enter your name at the top of the notebook (or you will lose 1 point).

Task 1. Import two libraries

Matplotlib is a popular library in Python for plotting and graphing purposes. Within the Matplotlib module we will specifically use a sub-module called pyplot and import it as plt – that's what we will call it within our program. It is custom to import and name it this way. The other library we need is NumPy.

```
import matplotlib.pyplot as plt
import numpy as np
```

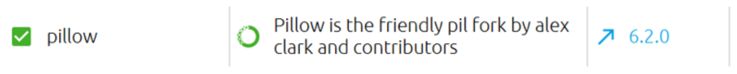
Task 2. Read in an image and view it in the Notebook (1 points)

You can use any colored PNG image that you want. I will use an called `img1.png`. We will mainly use PNG, but for comparison, you will also import the same image (after saving it as img1.jpg) in JPG format. You should be able to open both (as behind the scenes Matplotlib takes care of the dependency mentioned below).

Please store this image file in the same folder as your .ipynb notebook. This is what we will assume when we grade.

Aside: Matplotlib can only read PNGs natively. Further image formats are supported via the optional dependency on Pillow. Python's Imaging Library in its newer version (with Python 3 support) is called Pillow and older version (with Python 2) was called PIL.

You can first check Anaconda Navigator to make sure Pillow is installed (it should be already be installed for you with Anaconda); if not, you can install it there itself by selecting the library and choose Apply.



✓ pillow	Pillow is the friendly pil fork by alex clark and contributors	↗ 6.2.0

Or you can install using Anaconda command line.

```
conda install -c anaconda pillow
```

Read in the image using `plt.imread()` function.

```
img_png = plt.imread("img1.png")
```

You can add a label to the image on the two axes as follows.

```
plt.xlabel('columns')
plt.ylabel('rows')
```

You can view images using `plt.imshow()` function, and give it the variable that stores the image as an argument.

```
plt.imshow(img_png)
```

After calling `plt.imshow()`, plot a colorbar by calling the following command in the same cell as the `plt.imshow()` command.

```
plt.colorbar()
```

Repeat the above steps read in the .jpg version of the same image, with axis labels. Do you notice the difference in scale for the colors across the two image formats?

OPTIONAL

Sometimes, you may wish to resize your image if the one you picked is too small or too big to see well. You can do this as follows:

`plt.figure()` has an option called `figsize` that you can set = `(newrowsize, newcolsize)` in inches

```
plt.figure(figsize=(10, 12))
```

You have to write the above line of code at the *top of the cell* before you read in the image.

TASK 3. What kind of object is the image? (1 point)

Use the type function to obtain the class type of the PNG image object

TASK 4. Print the shape of both the PNG and the JPG images (2 point)

You can get this using the `ndarray.shape` attribute which returns a 3-tuple

Q. what difference do you notice in the size of the 3rd dimension across .jpg and .png images?

<Please answer question in a separate mark down cell by itself >

TASK 5. Use an f-string to print the number of pixels the image contains, for both images (2 points)

This calculation requires you to multiply the length (number of rows) by the width (number of columns) of the image. Use indexing on the tuple returned by the `ndarray.shape` attribute to obtain the length and width, and multiply them to produce the number of pixels.

Recall that values in a tuple can be indexed using []. The printed output should look similar to below (with diff values for the pixels):

```
The PNG image consists of 3666673 pixels
The JPG image consists of 3662820 pixels
```

TASK 6. Print the values of the 3rd axis (i.e. color/opacity) of the pixel at position [15,13] for both images. Also obtain and print the data type of the values in the 3rd axis for both images. (2 points)

Verify that your PNG image produces a (4,) array while the JPG image produces a (3,) array

Note that the datatype of colors are different across PNG and JPG.

From here on forth, we will only use the PNG image.

TASK 7. Obtain the mean values of the 3 colors – RGB – in your PNG image. (3 points)

Use a numpy function to calculate the mean. Don't forget to specify the value for the axis parameter, which can be a scalar or a tuple. If no axis is specified, the function will take the mean of values over all axes – which is not meaningful.

When we have a 2D array, axis is a scalar, i.e. we either want means over rows or columns. Here , we have a 3D array, so we can specify axis as just rows (0), or just columns (1) or both (0,1). Since we want the mean level of each of the 3 colors, we want to take means over both the row and column axes.

You can set axis = (rowaxis, columnaxis) – which will obtain means over all rows and columns for each of the values in the 3rd axis. This will give you 4 mean values for RGBA for the PNG image. Use appropriate indexing to ***only print*** the means of the RGB (but not A).

Next, we will perform data augmentation tasks. To get full points here, please use numpy array commands and not built-in functions from other modules (otherwise will only get half the points).

TASK 8a. Reverse the painting on the second-axis (flip horizontally) using array indexing (2 points)

Since we are flipping horizontally, the rows and the third-axis are not affected. Only the column values have to be flipped (reversed). To do this, you can use indexing with slicing on the original array, and use step size -1 for the second-axis (col). Since we want the whole image, all rows, cols and all values in the 3rd dimension, be sure to indicate that in the start and stop values in those dimensions.

Tip: Recall that a slice has 3 parts – start, stop, step. If you don't provide a start or end value, the default is assumed (so you don't have to type in hardcoded numbers for stop), and all values are used. To use all values in a dimension, you can use : for that dimension.

Step = -1 will read the values in that dimension from last to first, in reverse order.

TASK 8b. Reverse the picture on the first axis (flip vertically) using a Numpy function (1 point)

Perform the same task as in 8a, except use the `np.flip()` function.

TASK 9. Render the image with the lower half of the image set at transparency level = 0.5. The rest of the image (the top half) should be at its original level (4 points)

Usually when we manipulate an image by setting its values using the = symbol, it is good to make a copy if you do not wish to overwrite the original image.

First make a copy of the image. This can be by using the `ndarray.copy()` method of NumPy's ndarray (replace ndarray with your original image array name), and save this copy of the PNG image as a new ndarray object.

```
img_png_transp = img_png.copy()
```

Next, you will need to use indexing (such as slicing) to select only the lower half of the rows, all the columns and then set the fourth value (opacity) in the 3rd axis = 0.5. Write the code to do this by replacing the three values inside [] below with appropriate values.

`img_png_transp[row , col, 3rd-axis] = 0.5`

Tip: You can use the shape attribute to obtain the desired row index to start changing the transparency – which is half of the total row length. Row index has to be an int, so you can use the Python's int() function to cast to an int if half the rowsize is not an int.

Select all the columns and select only the opacity value from the 3rd axis.

Then print the image using `plt.imshow()`.

TASK 10. Obtain a negative of the image (3 points max with bonus, else 2)

Here, we want to reverse the colors for each RGB. Since the max value is 1.0 for PNG, we just want to subtract the value of each color from 1.0 to obtain its negative on that color channel. We will keep the opacity the same.

First create a copy of the original image.

Then select the values that you want to change from this copied image and set those values = new values.

> We want to select only the three colors in index 0, 1 and 2 from the 3rd axis (for all values of row and column) and then subtract the values for those three colors from 1.0

You can perform the subtraction for all three colors in one go using one line of code (using broadcasting along 2 dimensions), or you can write 3 lines of code to subtract each color one by one (using broadcasting along 1 dimension).

Here's how to subtract only the red color.

```
img_png_neg2[:,:, :1] = 1.0 - img_png[:,:,:1]
```

Complete the rest and print the image.

BONUS : Bonus point here for writing one line of code instead of 3 to perform the color subtractions from 1.

TASK 11. Display a cropped image (2 points)

Cropping is akin to selecting contiguous slices along both the row axis and the column axis, and keeping all the values on the 3rd dimension. Focus on any interesting aspect of your picture and obtain only that area of the image.

TASK 12. Display the image as grayscale (2 points)

A grayscale image only uses a single-color channel – from black to white with luminosity to indicate the level of gray. A grayscale image is therefore a 2D array (row, col) and no longer a 3D array – we only need to know the level of luminosity at each (row, col).

There are several ways to obtain this value – one approach is to use the weighted average of the three colors- RGB. What we want is to multiply the (3,) array of RGB colors with some (3,) array that contains desired weights for each color channel (opacity is ignored).

To perform the weighting, we can use a `np.dot()` or `np.matmul()` function which produces a dot product of two arrays. Let us use the weights 0.21, 0.72 and 0.07 - with green weighted the highest because humans are able to see more contrast in green colors.

Tip: To perform this task, we will first obtain the RGB values from the 3rd axis (for all rows and columns) of the image and multiply that array with another array of weights, [redweight, greenweight, blueweight]

Print the new image's shape to verify it is a 2D image. That is, the 3rd axis has been removed.

Now plot this image to see the grayscale image. To view in grayscale, we have to set an optional parameter `cmap` of `plt.imshow()` equal to `'gray'`

TASK 13. Apply Gamma correction to the image (2 points)

Gamma correction is applied to make the picture have more light or less light. This is achieved by raising the color values of each pixel to a power (gamma value). Recall the power operator in Python is **

```
output-image = input-image^ gamma
```

The colors are a float value (0-1.0). A gamma value > 1 makes the image darker (more shadows), whereas a gamma value of < 1, makes the image lighter and a gamma = 1 keeps the image the same.

Create a variable called `gamma = 1/5`.

Obtain an image array where ***only the 3 color values*** are raised to the power gamma (not opacity – leave it the same), for all pixels in the image. Then use `plt.imshow()` to print the images. Can you see the difference as compared to the original? Try changing the value of gamma to 5 and then to 1.

You know how to print each image one by one – call `plt.imshow()` each time in a separate cell. Show the two images with more light (gamma <1) and less light (gamma >1).

OPTIONAL

Wouldn’t it be nice to plot all 3 in one plot with 3 subplots, so we can view them all together as one output? have more control over each image – give it its own subtitle etc.?

You can do this using Matplotlib subplots to print the three images horizontally within one plot. Let us give the title of the plot as ‘Horizontally stacked subplots’. Each sub-image should have its own title (gamma = value), it should not show the yticks (row ticks), but show the xticks (column ticks).

Below is the code.

```
fig, ax = plt.subplots(1, 3)
fig.suptitle('Horizontally stacked subplots')
```

```
ax[0].set_title('gamma = 1/5')
ax[1].set_title('gamma = 1')
ax[2].set_title('gamma = 5')
```

```
ax[0].set_yticks([])
ax[1].set_yticks([])
ax[2].set_yticks([])
```

```
ax[0].imshow(img_png_sat1)
ax[1].imshow(img_png_sat2)
ax[2].imshow(img_png_sat3)
```

`plt.subplots()` allows us to make a plot with sub-plots (more than one plotting area, known as **AxesSubplot**). The dimensions are specified as a 2-tuple. (1,3) means that plots are arranged using 1 row and 3 columns.

`plt.subplots()` function returns two values – which we save in two variables, fig and ax. This is called ***unpacking a tuple*** (separating its values out).

The main overall figure is stored in fig and the axes of the subplot are stored in ax. ax is an ndarray of shape (3,). That is, a 1D vector with 3 elements or 3 sub-plots. Each subplot is accessed as ax[0], ax[1] and ax[2], respectively. We can then set individual titles using the set_title() method of AxesSubplot. By default, x and y ticks will show, or you can use set_xticks and set_yticks method of AxesSubplot (set it to [] to not show ticks).

`fig.suptitle()` accepts the title of the main plot. It has a size keyword parameter that you can set using an int.

Finally, we call imshow() on each subplot given by ax[0], ax[1] and ax[2]

TASK 14. Obtain and print one image formed by horizontally stacking 3 images – redscale, greenscale and bluescale of the original image (3 points).

To do this, we first obtain 3 different images from the original image. The redscale image is where the colors green and blue are set to 0; the greenscale is where the colors red and blue are set to 0, and the bluescale image is where the colors red and green are set to 0.

For each color, first create a copy of the original image and save in a new variable, say img_png_Red. Use the `ndarray.copy()` method. Then change the green and blue colors to zero. You can use the command.

```
img_png_Red[:,:,[1,2]] = 0
```

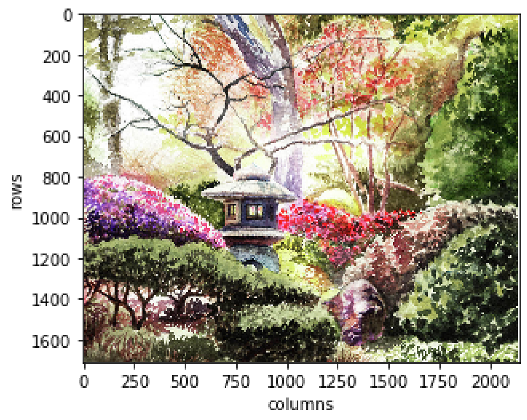
Do this for the other two colors as well, and you will then have 3 new images.

Next, use a numpy function to join the three images together horizontally (left to right with the red scale first, followed by green scale and then blue scale).

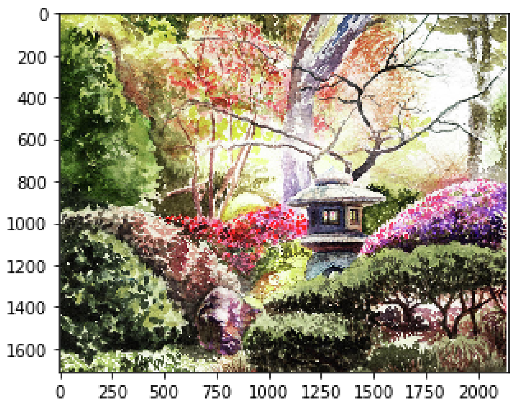
Finally call `plt.imshow()` with this new image horizontally stacked image.

C. Sample outputs of tasks that produce an image

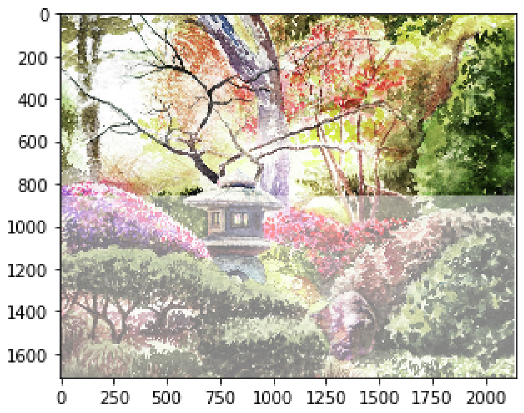
Task 2



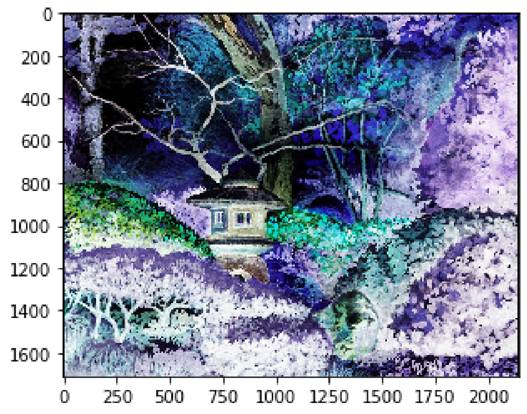
Task 8



Task 9



Task 10



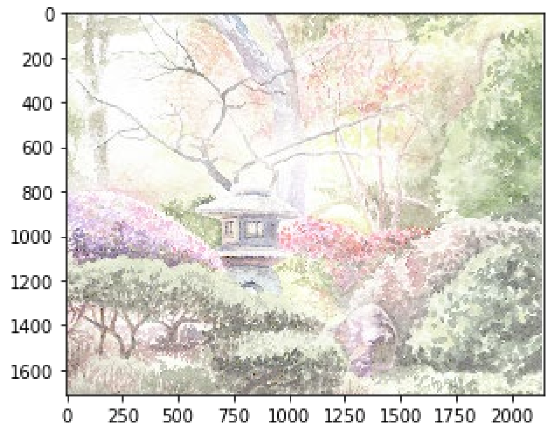
Task 11

0
100
200
300
400
0
100
200
300
400
500

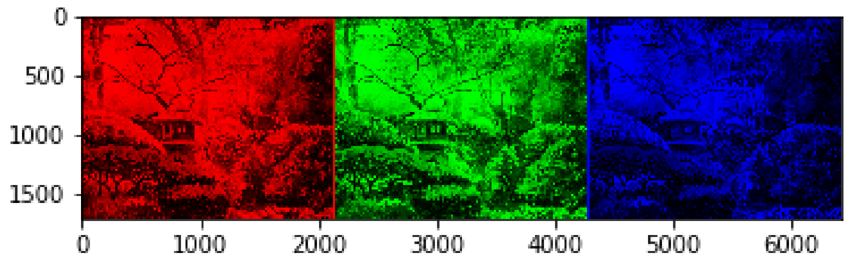
Task 12

0
200
400
600
800
1000
1200
1400
1600
0
250
500
750
1000
1250
1500
1750
2000

Task 13


0
200
400
600
800
1000
1200
1400
1600
0
250
500
750
1000
1250
1500
1750
2000

Task 14


0
500
1000
1500
0
1000
2000
3000
4000
5000
6000