iitgn-ss-week1-questions

December 14, 2023

0.1 Getting Started

0.1.1 Instructions

1. Make a Copy of this Colab Notebook in your Drive. To do so: > * Go to the "File" tab in the top navigation bar. > * Click on "Save a copy in Drive". > * Once done, open the copied Colab for editing.
2. Read the questions for each exercise by **adhering to nomenclature used**.
3. Write your code solutions only in the designated cells (**# CODE HERE**).
4. Run any other cells in the sequence they appear.

0.1.2 Importing Libraries

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

0.2 NumPy Operations - *Into the World of Dimensions*

In this section, we will explore different operations that we can perform using numpy efficiently:

For often, while venturing into the world of High-Dimensional Data(one involving multi-dimensional arrays), we often need to summarise its entirety within few dimensions. This process is called Principal Component Analysis (PCA). We walk through the various steps involved from scratch.

Let's create a 2D Numpy Array which we'll represent as a Matrix

Suppose we have this 9 x 5 (N x M) Martix X, indicating that there are 5 points/vectors in 9-dimensional space:

$$X_{NxM} = \begin{bmatrix} 6 & 3 & 1 & 6 & 7 \\ 3 & 9 & 2 & 2 & 5 \\ 2 & 3 & 4 & 8 & 5 \\ 4 & 2 & 5 & 1 & 2 \\ 6 & 4 & 1 & 4 & 1 \\ 3 & 0 & 5 & 0 & 0 \\ 2 & 5 & 0 & 7 & 2 \\ 6 & 7 & 5 & 0 & 8 \\ 8 & 2 & 6 & 5 & 2 \end{bmatrix} \quad (1)$$

[ ]:
```
X = np.array([[6, 3, 2, 4, 6, 3, 2, 6, 8], [3, 9, 3, 2, 4, 0, 5, 7, 2], [1, 2,␣
↪4, 5, 1, 5, 0, 5, 6], [6, 2, 8, 1, 4, 0, 7, 0, 5], [7, 5, 5, 2, 1, 0, 2, 8,␣
↪2]])
print(X)
```

Notice each vector (Xi) was a **column** when represented in matrix form viz:

$$X_i = \begin{bmatrix} 6 \\ 3 \\ 2 \\ 4 \\ 6 \\ 3 \\ 2 \\ 6 \\ 8 \end{bmatrix} \tag{2}$$

But NumPy arrays always have Row as a Vector. So the matrix representation would be the **transpose** of the array **X**:

[ ]:
```
print(X.T)
```

The first step in the PCA is to Standardise the given matrix. To standardise a matrix, we replace each column vector by subtracting each column vector of the matrix from **the mean vector** :

$$\mu = \frac{\sum_{i=1}^{M} X_i}{M} \tag{3}$$

So,

$$\mu = \frac{\begin{bmatrix} 6 \\ 3 \\ 2 \\ 4 \\ 6 \\ 3 \\ 2 \\ 6 \\ 8 \end{bmatrix} + \begin{bmatrix} 3 \\ 9 \\ 3 \\ 2 \\ 4 \\ 0 \\ 5 \\ 7 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 4 \\ 5 \\ 1 \\ 5 \\ 0 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 6 \\ 2 \\ 8 \\ 1 \\ 4 \\ 0 \\ 7 \\ 0 \\ 5 \end{bmatrix} + \begin{bmatrix} 7 \\ 5 \\ 5 \\ 2 \\ 1 \\ 0 \\ 2 \\ 8 \\ 2 \end{bmatrix}}{5} \tag{4}$$

Hence, each new column of X is now:

$$A_i = X_i - \mu \tag{5}$$

0.2.1 EXERCISE 1

Compute the Mean Vector using np.mean() and name it **mean**

[ ]:
```
# CODE HERE
```

0.2.2 EXERCISE 2

Compute the standardised matrix X after subtracting **mean** from **X** and name it **A** and also find the shape of the A array. Notice that A is N x M matrix while in NumPy array it has the shape M x N.

[ ]:
```
# CODE HERE
```

Now comes the time to compute the **Covariance Matrix** of this new matrix **A**. The covariance matrix (Σ) comes from the Matrix Multiplication of the standardised matrix and its transpose.

$$\Sigma = \frac{A.A^T}{(M-1)} \tag{6}$$

0.2.3 EXERCISE 3

Compute the Covariance matrix of A using numpy matrix multiplication and call it **cov**. Also find its shape. Hint: Take care of the shape of the array and recall the matrix representation thing!

[ ]:
```
# CODE HERE
```

Last few steps involve computing the Eigenvalues and Eigenvectors of the Covariance Matrix. ### **EXERCISE-4** Compute the EigenValues and EigenValues of **cov** using NumPy's Linear Algebra. Store them in **EigenValues** and **EigenVectors** named arrays.

[ ]:
```
# CODE HERE
```

NOTE: The EigenVector array returned actually has the eigenvectors as its columns rather than its rows (unlike the usual NumPy array scenario). So you need to transpose it before proceeding further.

[ ]:
```
EigenVectors = EigenVectors.T
print(EigenVectors)
```

The next task is to arrange the rows of the **EigenVectors** Array based upon the indices of the Eigenvalues in descending order. Look at the following routine:

[ ]:
```
indices = np.array([3, 2, 4, 0, 1])
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
print(f"The Original Array:\n{arr}\n")
print(f"The array with Rows rearranged according to the indices:
 ↪\n{arr[indices]}")
```

0.2.4 EXERCISE-5

Arrange the Eigenvectors(the rows of the **EigenVectors** Array) corresponding to the values of the **EigenValues** Array(in descending order) following a routine similar to the above.

[ ]:
```
# CODE HERE
```

0.2.5 EXERCISE-6

Extract only the first **K** rows of this rearranged array and store them in an array named **W**. Also find the shape of **W**.

[ ]:
```
K = 2
```

[ ]:
```
# CODE HERE
```

The Final Step of Dimensionality Reduction (PCA) involves the multiplication of the standardised matrix **A** with this **W** matrix:

$$X_{new} = A^T_{MxN} . W_{NxK} \tag{7}$$

Notice that the Reduced Version of **X** is of the Dimension **M x K**. Hence we have arrived from an **N**-dimensional space to a smaller **K**-dimensional space while keeping the number of points **M** same still yet extracting the max amount of information from the original data!!

[ ]:
```
X_New = A @ W.T
print(X_New)
print(X_New.shape)
```

In python, we have come from (5, 9) to (5, 2). Now we can easily plot these 5 points in 2D which initially were in 9D. ### **EXERCISE-7** Plot these 5 points using MatPlotLib.

[ ]:
```
# CODE HERE
```

0.3 Image Processing - *Seeing through the eyes of arrays*

We explore the smart use of NumPy when working with images. Such operations would enable one to understand the basic concepts behind Image Processing and Computer Vision.

###**Creating an Image Array**

We first load the image using the MatPlotLib package. Next, this image is converted to a numpy array.

In the code given below, **image** is the name of the array that stores our image.

[ ]:
```
image = plt.imread('/content/LalMinar.jpeg')
image = np.array(image)
image
```

Notice that the **image** array is a three-dimensional one. The first two dimensions offer the height and width of the image. While the mysterious third dimension shows the RGB channels present in the image.

[ ]:
```
image.shape
```

The total number of pixels in the image

[ ]:
```
image.size
```

###**Displaying the Image**

We will define a function **disp()** to display the RGB images, so we can reuse this piece of code.

```
[ ]: def disp(img):
       plt.imshow(img)
       plt.axis("off")
       plt.show()
```

This is what the image looks like

```
[ ]: disp(image)
```

###**The Red, Green, and Blue Channels**

The Red Channel

```
[ ]: image[:, :, 0]
```

0.3.1 EXERCISE-1

Extract the Green and Blue Channels likewise

The Green Channel

```
[ ]: #CODE HERE
```

The Blue Channel

```
[ ]: #CODE HERE
```

0.3.2 Visualizing the Channels

Notice the image in RG, GB, and RB channels

```
[ ]: imgRG, imgGB, imgRB = image.copy(), image.copy(), image.copy()
     imgRG[:, :, 2] = 0
     imgGB[:, :, 0] = 0
     imgRB[:, :, 1] = 0
     print("Image in RG Channel:")
     disp(imgRG)
     print("Image in GB Channel:")
     disp(imgGB)
     print("Image in RB Channel:")
     disp(imgRB)
```

###**EXERCISE-2**

Display the image in Red, Green, and Blue Channels separately

```
[ ]: # CODE HERE
```

###**BONUS: Copy and DeepCopy**

When you copy python arrays by doing:

array2 = array1

This statement does not create a new copy of the array in the memory, instead, it just creates a new pointer array2 that points to array1 in the memory. Any changes in array2 will be reflected in array1 and vice-versa. Try this!

```
[ ]: # CODE HERE
```

To avoid this and create a completely new copy of arrays in the memory, we use .copy() for one dimensional arrays and deepcopy() for higher dimensional arrays.

```
[ ]: # CODE HERE
```

###**Converting to Grayscale** A grayscale image is a two dimensional array, with every value specifing the black content. For obtaining a grayscale pixel channel using 3 RGB channels, we take 29.89% contribution from the Red Channel, 58.70% contribution from the Blue Channel and the remaining 11.40% contribution from the Green Channel.

Hence, we create a grayscale image by taking the dot product with this weighted array for RGB using **np.dot()**.

```
[ ]: weights = [0.2989, 0.5870, 0.1140]
     gray_image = np.dot(image, weights)

     plt.imshow(gray_image, cmap='gray') # We can't use disp() here as we need cmap␣
      ↪= 'gray' for displaying grayscale images
     plt.axis('off')
     plt.show()
```

Also notice the shape of **gray_image**. It no longer has the third dimension!

```
[ ]: gray_image.shape
```

0.3.3 Image Cropping

All cropping involves is the slicing the image array within certain limits aka Regions of Interest (ROI).

```
[ ]: left, right = 300, 500
     top, bottom = 150, 600
     cropped_image = image[top : bottom, left : right]
     disp(cropped_image)
```

0.3.4 Image Resizing

Let's start by creating an array and taking out some specific elements from it, say we create a 3X3 matrix and we want elements in the (Row, Col) = {(0, 1), (0, 2), (2, 1), (2, 2)}:

[ ]:
```
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

rows = np.array([0, 2])
cols = np.array([1, 2])

selection = arr[np.ix_(rows, cols)]
selection
```

###The **np.__ix()** We start by selecting the required number of rows and columns. This is done using **np.linspace()** *Note:* ***np.linspace()*** *generates an array containing elements from the starting to the ending element, the number of elements is decided by the third parameter.*

[ ]:
```
target_width = 500
target_height = 400

row_indices = np.linspace(0, image.shape[0] - 1, target_height, dtype=int)
col_indices = np.linspace(0, image.shape[1] - 1, target_width, dtype=int)

resized_image = image[np.ix_(row_indices, col_indices)]

disp(resized_image)
```

[ ]:
```
resized_image.shape
```

###**Flipping an Image** Flipping the image is equivalent to flipping the image array itself.

[ ]:
```
flipped_image = np.flip(image, axis=0)

disp(flipped_image)
```

0.3.5 Having seen a lot of the work being done already, its now the time to test yourself out!

0.3.6 EXERCISE-3

Calculate the **number of pixels** having **green intensity > blue intensity**.

[ ]:
```
# CODE HERE
```

0.3.7 EXERCISE-4

Calculate the **coordinates**[(height, width)] of the **Brightest Red Pixel** in the image.Hint: Use np.unravel_index() and np.argmax() for a single line code.

[ ]:
```
# CODE HERE
```

0.3.8 EXERCISE-5

Find the **Average value** of the **Red, Green** and **Blue** Channels in the image

[ ]:
```
# CODE HERE
```