6CS012 - Artificial Intelligence and Machine Learning.
Week - 01 - Workshop - 01

Image Compression and Decompression with PCA.

" An application of Eigen Value Decomposition".

Siman Giri

Workshop - 01

February - 23 - 2025

Outline

1. Understanding Image as a Matrix.

1.1 What is an Image?

Figure: Lenna!!! The image first appear in 1966 IEEE transactions on Image Proceesing with a note by Edition in chief "image contains a nice mixture of detail, flat regions, shading, and texture that do a good job of testing various image processing algorithms".

Cautions: Lenna Image has been retired i.e. Major scientific publications including Nature and IEEE do not accepts the publication with the use of Lenna Image.

1.2 Understanding Image as a Mathematical Function.

- To simplify the concept of an image as a mathematical function, we can think of an image as a matrix of numbers.
- **Image as a Matrix:**
  - Consider an image with dimensions $m \times n$, where m is the number of rows (height) and n is the number of columns (width) of pixels.
  - Each element in the matrix corresponds to the intensity of a pixel.
  - So, the image can be represented as a **matrix I** where each element $\mathbf{I}[\mathbf{i}, \mathbf{j}]$ represents the intensity of the pixel at row i and column j, and its value lies between 0 and 255.

$$I = \begin{bmatrix} I[1,1] & I[1,2] & \dots & I[1,n] \\ I[2,1] & I[2,2] & \dots & I[2,n] \\ \vdots & \vdots & \ddots & \vdots \\ I[m,1] & I[m,2] & \dots & I[m,n] \end{bmatrix}$$

1.3 Grayscale Image (Matrix Representation).

For a grayscale image with dimensions $m \times n$ (height m, width n), the image can be represented as a **2D matrix** I:

$$I = \begin{bmatrix} I[1,1] & I[1,2] & \dots & I[1,n] \\ I[2,1] & I[2,2] & \dots & I[2,n] \\ \vdots & \vdots & \ddots & \vdots \\ I[m,1] & I[m,2] & \dots & I[m,n] \end{bmatrix}$$

Example:

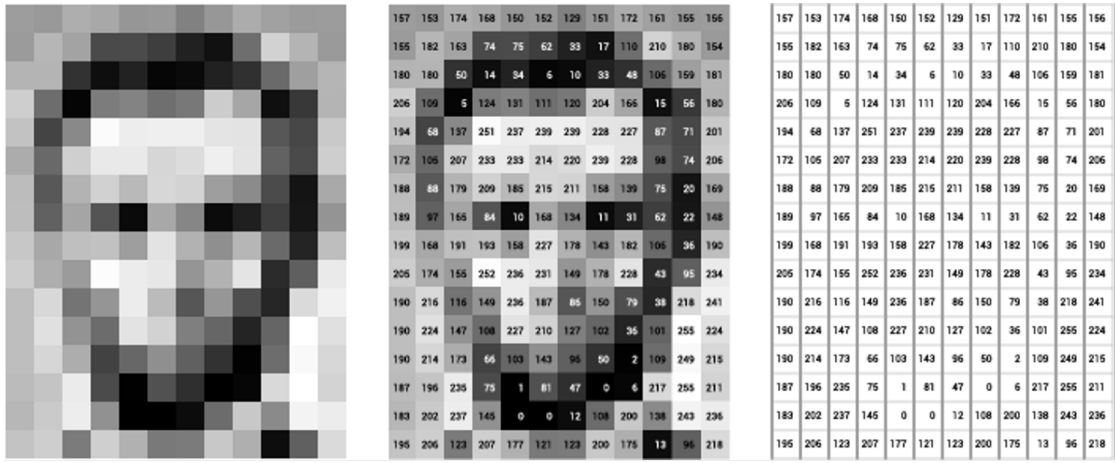


Image from Internet Subject to copyright.

Figure: Grayscale Image as a Matrix.

1.4 Color Image (RGB) as a Matrix

For a color image with dimensions $m \times n$ (height m, width n), the image is represented as a 3D matrix where each element is a vector containing the R, G, and B values for a specific pixel.

$$I = \begin{bmatrix} \begin{bmatrix} R[1,1] \\ R[2,1] \end{bmatrix} & \begin{bmatrix} G[1,1] \\ G[2,1] \end{bmatrix} & \begin{bmatrix} B[1,1] \\ B[2,1] \end{bmatrix} & \begin{bmatrix} R[1,2] \\ R[2,2] \end{bmatrix} & \begin{bmatrix} G[1,2] \\ G[2,2] \end{bmatrix} & \begin{bmatrix} B[1,2] \\ B[2,2] \end{bmatrix} & \cdots & \begin{bmatrix} R[1,n] \\ R[2,n] \end{bmatrix} & \begin{bmatrix} G[1,n] \\ G[2,n] \end{bmatrix} & \begin{bmatrix} B[1,n] \\ B[2,n] \end{bmatrix} \\ \vdots & & & \vdots & & & \ddots & & \vdots & \\ [R[m,1] & G[m,1] & B[m,1]] & [R[m,2] & G[m,2] & B[m,2]] & \cdots & [R[m,n] & G[m,n] & B[m,n]] \end{bmatrix}$$

Here:

- $R[i,j], G[i,j], B[i,j]$ represent the intensity values for the Red, Green, and Blue channels, respectively, at the pixel (i,j).
- Each pixel is represented by a triplet: $[R[i,j], G[i,j], B[i,j]]$, where each component ranges from 0 to 255.

1.4.1 Color Image (RGB) as a Matrix: Example

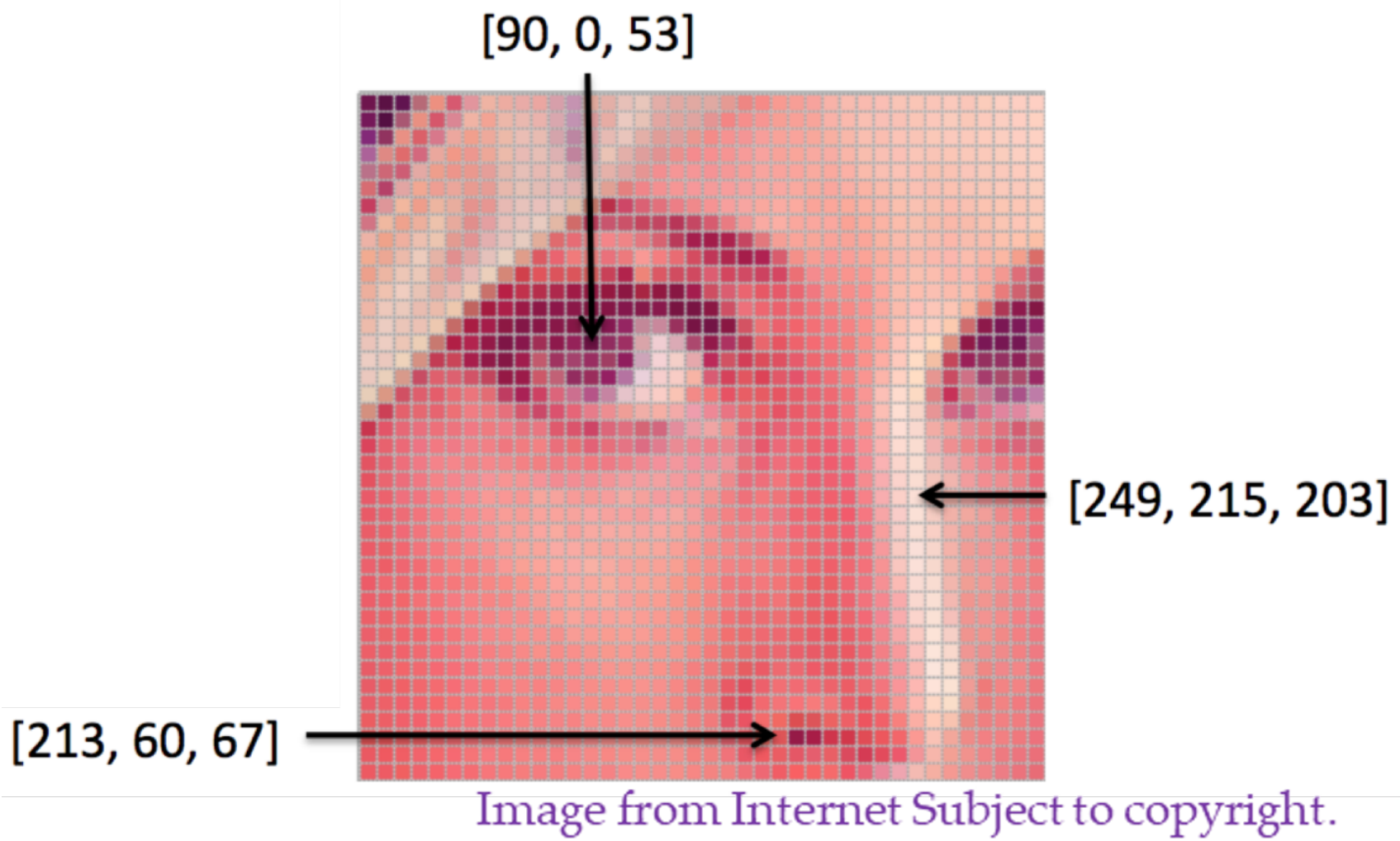
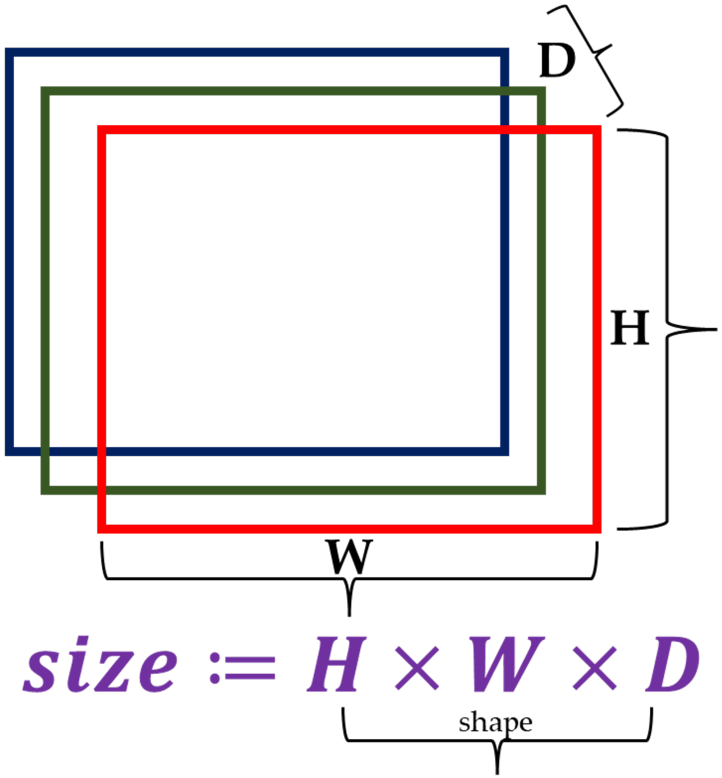


Figure: Color Image as a Matrix.

1.5 Shape and Size of an Image.

- **Shape:** Refers to the dimensions of the image (rows, columns, and channels).
- **Size:** Refers to the total number of pixels in the image.



$size_{color} := H \times W \times 3$ $size_{grayscale} := H \times W$

Figure: Shape and Size of an Image - Matrix.

2. Basic Image Processing with Python. Introduction to Pillow Library.

2.1 Basic Overview of Pillow.

- Pillow is a Python Imaging Library (PIL) fork that provides easy-to-use methods for opening, manipulating, and saving images.
- It is widely used in machine learning and deep learning pipelines for preprocessing images before feeding them into models.
- **Installation:**To install Pillow, use the following command:

```
pip install pillow

```

Installation of pillow

- "Comes in built with Colab environment."

2.2 Loading and Displaying Images.

- In General:

```
image = Image.open("/content/lenna_image.png")
image.show()
```

Opening an Image.

- For Colab:

```
from PIL import Image
# display image in colab
image_colored = Image.open("/content/Lenna_(test_image).
    png")
display(image_colored)
```

Opening an Image.

- **Cautions:**
  - image.show() is not directly supported by Colab Environment, Thus use display() function.

2.3 Getting Image Properties

- **For Gray Scale Image:**

```
print ( " Image Format : " , image_grayed . format )
print ( " Image Size : " , image_grayed . size )
print ( " Image Mode : " , image_grayed . mode )
```

- **Expected Output:**

```
Image Format :  GIF
Image Size :  (512, 512)
Image Mode :  P
```

- **For Colored Image:**

```
print ( " Image Format : " , image_colored . format )
print ( " Image Size : " , image_colored . size )
print ( " Image Mode : " , image_colored . mode )
```

- **Expected Output:**

```
Image Format :  PNG
Image Size :  (512, 512)
Image Mode :  RGB
```

2.3.1 Getting Shape and Size of an Image Using PIL

- **For Gray Scale Image:**

```
# Get the size (width, height) and channels (RGB)
width, height = image_grayed.size
channels = len(image_grayed.getbands())  # For RGB, it will be 3
print(f"Image shape (RGB): ({height}, {width}, {channels})")
image_size_grayed = width*height*1
print(f"Image size (RGB): {image_size_grayed}")
```

- **Expected Output:**

```
Image shape (RGB): (512, 512, 1)
Image size (RGB): 262144
```

2.3.2 Getting Shape and Size of an Image Using PIL

- **For Colored Image:**

```
# Get the size (width, height) and channels (RGB)
width, height = image_colored.size
channels = len(image_colored.getbands())  # For RGB, it will be 3
print(f"Image shape (RGB): ({height}, {width}, {channels})")
image_size_colored = width*height*3
print(f"Image size (RGB): {image_size_colored}")
```

- **Expected Output:**

```
Image shape (RGB): (512, 512, 3)
Image size (RGB): 786432
```

2.4 Converting Image to NumPy Array.

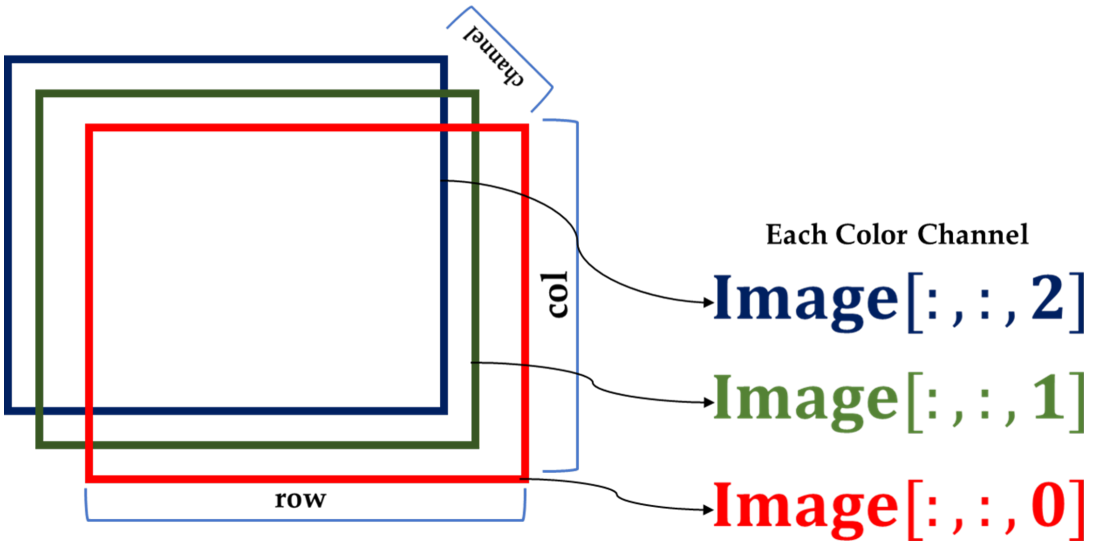
- **For Colored Image:**

```
# Convert the Pillow image to a NumPy array
image_array_colored = np.array(image_colored)
# Display the shape of the NumPy array (height, width, channels)
print("Shape of the image array:", image_array_colored.shape)
#Expected Output
Shape of the image array: (512, 512, 3)
```

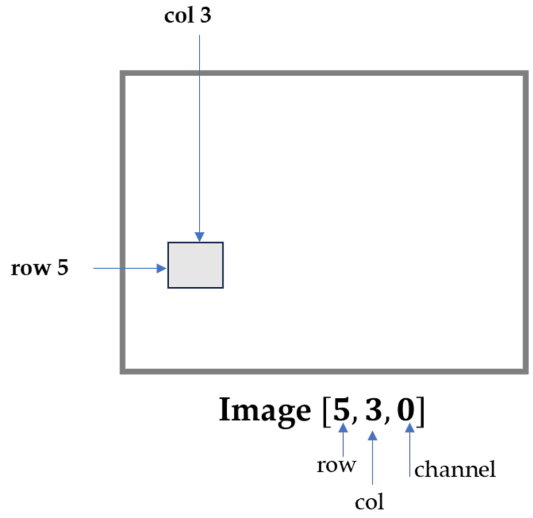
- **For Gray Scale Image:**

```
# Convert the Pillow image to a NumPy array
image_array_grayed = np.array(image_grayed)
# Display the shape of the NumPy array (height, width, channels)
print("Shape of the image array:", image_array_grayed.shape)
#Expected Output
Shape of the image array: (512, 512)
```

2.5 {Python} NumPy Indexing of an Image Array



Extracting Channels from Color image



Extracting Specific values from a channel.

2.5.1 Extracting Colored Channels with NumPy Array.

Accessing individual channels

```
# Convert the Pillow image to a NumPy array
image_array_colored = np.array(image_colored)
# Display the shape of the NumPy array (height, width, channels)
print("Shape of the image array:", image_array_colored.shape)
#For Red Channels
red_channel = image_array_colored[:, :, 0]  # Red channel
print(red_channel)
display(red_channel)
# For Green Channels
green_channel = image_array_colored[:, :, 0]  # Green channel
print(green_channel)
display(green_channel)
```

2.5.1 Extracting Colored Channels with NumPy Array.

Accessing individual channels

```
# For Blue Channel:
blue_channel = image_array_colored[:, :, 2]  # Blue channel
print(blue_channel)
display(blue_channel)
```

Expected Output:

```
[[125 125 133 ... 122 110  90]
 [125 125 133 ... 122 110  90]
 [125 125 133 ... 122 110  90]
 ...
 [ 60  60  58 ...  84  76  79]
 [ 57  57  62 ...  79  81  81]
 [ 57  57  62 ...  79  81  81]]
```

blue output

Figure: Similar Output can be Observed for each color channel.

2.5.2 Extracting Colored Channels with NumPy Array.

Accessing individual channels

```
# For Blue Channel:
blue_channel = image_array_colored[:, :, 2]  # Blue channel
print(blue_channel)
display(blue_channel)
```

Expected Output:

```
[[125 125 133 ... 122 110  90]
 [125 125 133 ... 122 110  90]
 [125 125 133 ... 122 110  90]
 ...
 [ 60  60  58 ...  84  76  79]
 [ 57  57  62 ...  79  81  81]
 [ 57  57  62 ...  79  81  81]]
```

blue output

Figure: Similar Output can be Observed for each color channel.

2.6 Extracting Colored Channels Using PIL.

Accessing individual channels

```
# Assuming image_colored is already defined and loaded above
# Get the R, G, and B channels
r, g, b = image_colored.split()
# Display or further process the individual channels
display(r)
display(g)
display(b)
```

Expected Output:

Figure: Extracted Colored Channels using pillow.

2.7 Extracting Sub Images from an Image Array.

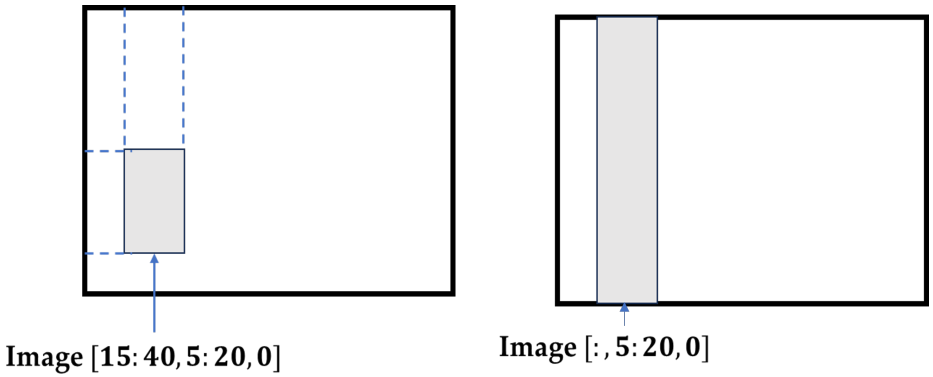


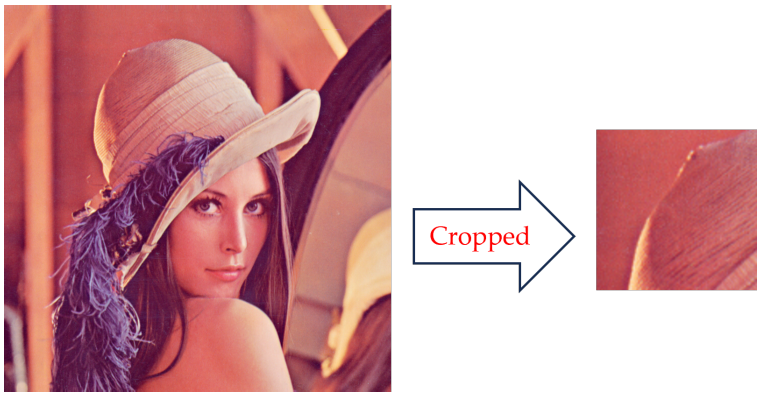
Figure: Similar to Indexing in Multi - dimensional ndarray.

```
# Sample Implementation:
# Accessing a specific row - 100 th row
row_100 = image_array[100, :, :]
# Accessing a specific column - 50 th col
col_50 = image_array[:, 50, :]
# Accessing a specific pixel (row 10, col 20)
pixel = image_array[10, 20, :]  # Gets RGB values at (10,20)
```

2.7.1 Extracting Sub Images(cropping) Using PIL .

```
# Define the cropping box (left, upper, right, lower)
left = 100, upper = 50, right = 200, lower = 150
# Crop the image
cropped_image = image_colored.crop((left, upper, right,
    lower))
# Display the cropped image
display(cropped_image)
```

Expected Output:



2.8 Saving NumPy Array to Image.

```
# Create a NumPy array (Here, we use a dummy array for illustration)
image_array = np.zeros((100, 100, 3), dtype=np.uint8)  # Example: black image
# Convert the NumPy array back to a Pillow Image object
image_from_array = Image.fromarray(image_array)
# Display the image
image_from_array.show()
# Optionally save the image to a file
image_from_array.save("output_image.jpg")
```

3.Understanding PCA for Image Compression. " With Eigen Value Decomposition."

3. What is PCA?

- Principal Component Analysis (PCA) is a dimensionality reduction technique used in machine learning, statistics, and data analysis.
- It transforms high-dimensional data into a lower-dimensional space while preserving as much variance as possible.
- **Key Concepts of PCA:**
  1. Feature Transformation: PCA converts correlated features into a set of linearly uncorrelated variables called principal components.
  2. Variance Preservation: The first principal component captures the highest variance in the data, the second captures the next highest, and so on.
  3. Eigenvalues & Eigenvectors: PCA relies on computing eigenvectors and eigenvalues of the covariance matrix to determine the principal components.
  4. Orthonormal Basis: The transformed features (principal components) form an orthogonal basis, making them independent.
  5. Dimensionality Reduction: By selecting only the top k principal components, we reduce the feature space while retaining most of the variance.

3.2 PCA Applied to Task of Image Compression.

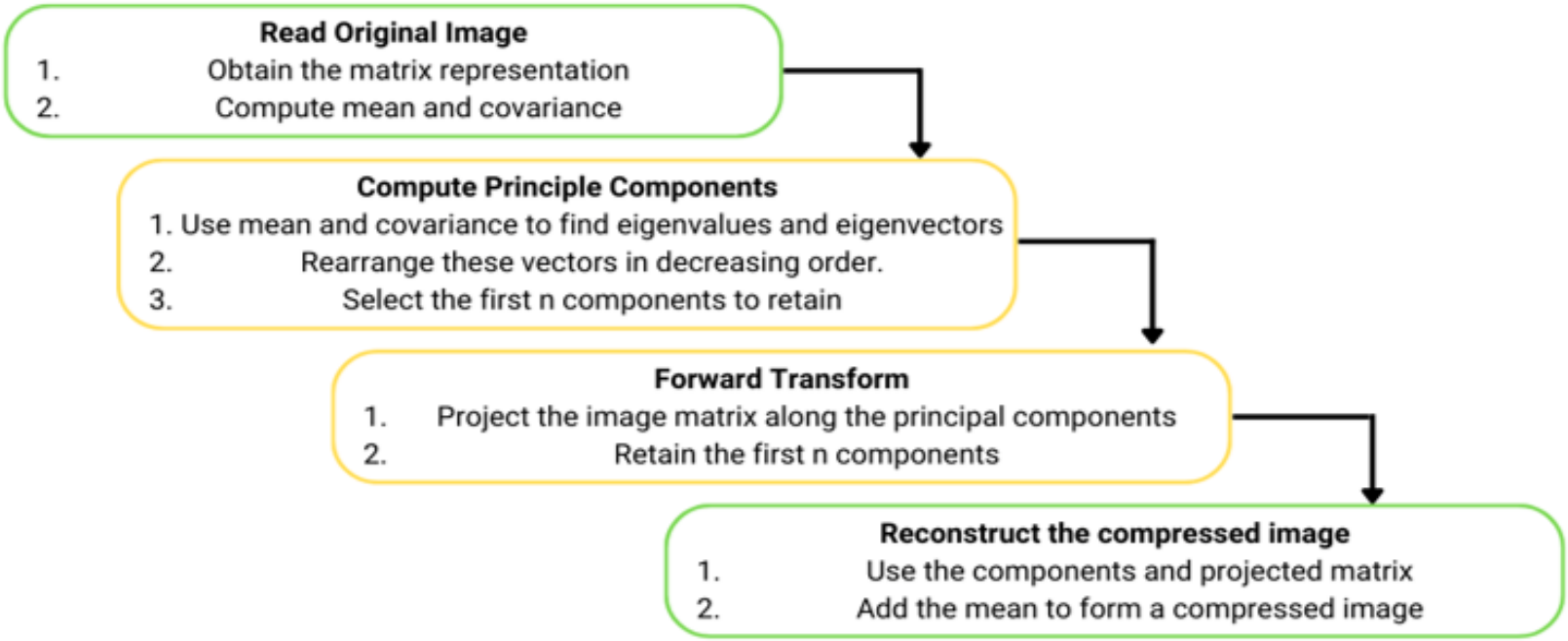


Figure: Steps in PCA Applied to Image Compression Task.

Step - 0 - Load and Prepare Image Data:

- Convert the image to grayscale.{If Image is not Gray Scale}
- Convert it into a NumPy array for processing.

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
# Load image and convert to grayscale
image = Image.open("image_cameraman.jpg").convert("L")
# Convert to NumPy array
image_array = np.array(image)
# Get image dimensions
height, width = image_array.shape
# Reshape the image into a 2D array where each row
    represents a row of pixels
# This allows PCA to process each row as an observation
    and find patterns across pixel intensities
data = image_array.reshape(height, width)
```

- **key insights:**PCA treats each row as a data point, where each pixel is a feature.

Step - 1 - Standardize the Data:

- Why Standardization? PCA is sensitive to different ranges of values.
- Subtract mean from each pixel to center the data.

```
# Compute the mean of each column (feature)
mean = np.mean(data, axis=0)
# Subtract mean to center the data
centered_data = data - mean
```

- **key insights:**Centering ensures PCA focuses on variance, not absolute values.

Step - 2 - Compute Covariance Matrix:

- The covariance matrix captures relationships between pixels.
- Higher covariance means stronger correlation.

```
# Compute covariance matrix
cov_matrix = np.cov(centered_data, rowvar=False)
```

- **key insights:** This step helps identify important patterns in the image.

Step - 3 - Compute Eigenvalues & Eigenvectors:

- Eigenvectors: Principal directions of data variation.
- Eigenvalues: Magnitude of variance in those directions.
- Sorting eigenvalues helps identify the most informative principal components.

```
# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
# Sort eigenvalues and corresponding eigenvectors in
    descending order
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

- **key insights:** Larger eigenvalues correspond to components with more variance.

Step - 3.1 - Picking the Principal Components:

- After sorting the eigenpairs, the next question is "how many principal components are we going to choose for our new feature subspace?" A useful measure is the so-called "explained-variance" which can be calculated from the eigenvalues.
- The explained variance tells us how much information (variance) can be attributed to each of the principal components.

Step - 3.2 - Explained Variance :

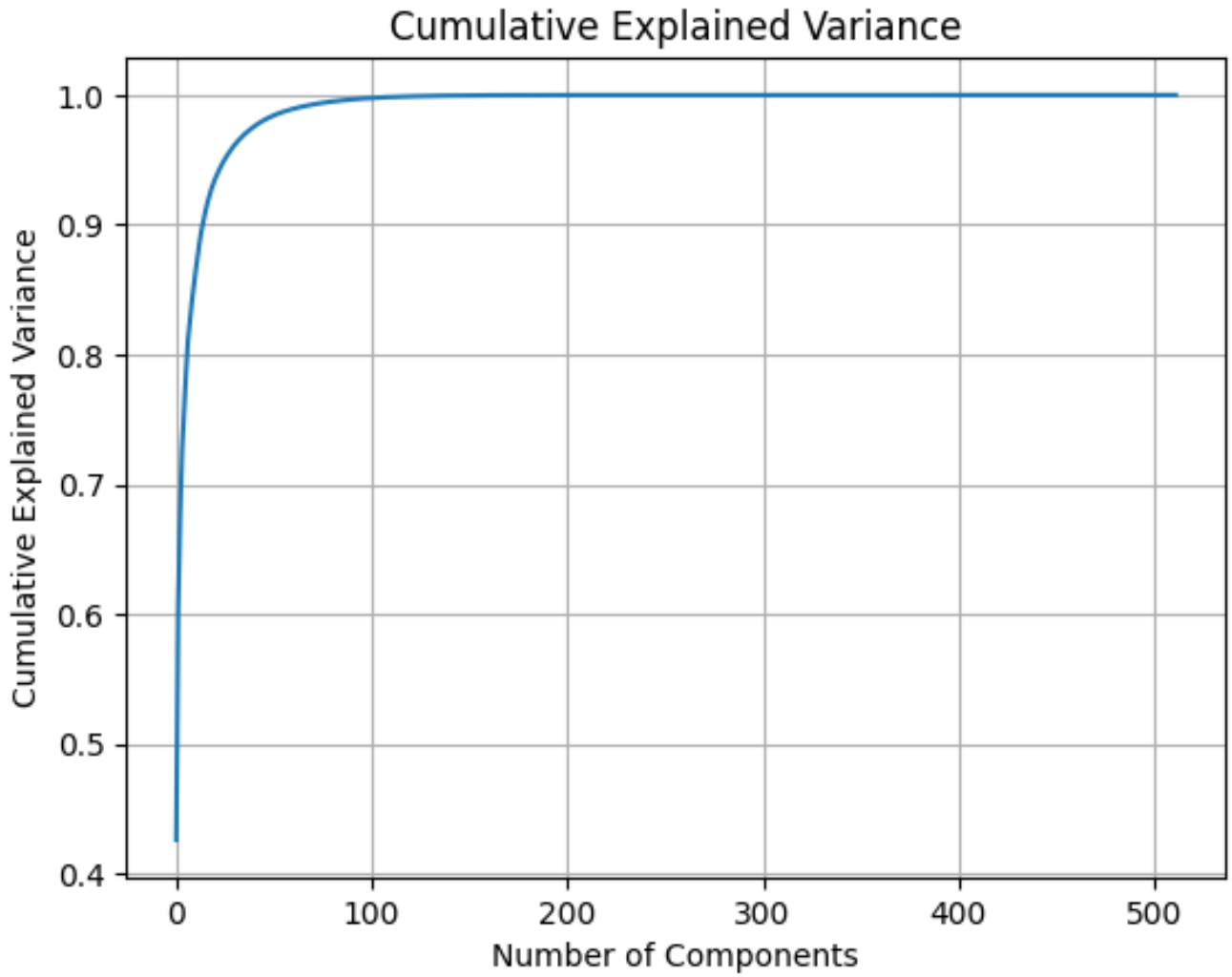
Sample Code and Implementation:

```
# Plot the explained variance ratio
explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
plt.plot(np.cumsum(explained_variance_ratio))
plt.title("Cumulative Explained Variance")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
plt.grid(True)
plt.show()
```

Output:

Step - 3.2 - Explained Variance:

Output:



Step - 4 - Select Top k Principal Component :

- Choose k components that retain most variance.
- The larger k, the more information is retained.
- You can make use of "Explained Variance" to make decisions.

```
k = 50  # Choose k principal components
components = eigenvectors[:, :k]
```

Step - 5 - Compress the Image :

- Transform the image into the lower-dimensional subspace
- The compressed representation has fewer dimensions than the original image.

```
# Project the data onto the principal components
compressed_data = np.dot(centered_data, components)
```

Step - 5.1 - Understanding the Compression:

- **centered_data:** This is the data matrix after subtracting the mean, so it has a shape of $(\text{num\_images}, \text{num\_pixels})$, which corresponds to the height $\times$ width of the image if we are working with a single image. If the image is 512×512, the shape of centered_data will be:

$$\text{centered\_data.shape} = (512, 512)$$

- **components:** The components matrix (also called eigenvectors in PCA) has a shape of $(\text{num\_pixels}, \text{n\_components})$, where n_components is the number of principal components you decide to keep. If you keep 50 principal components, the shape of components will be:

$$\text{components.shape} = (512, 50)$$

Step - 5.2 - Understanding the Compression:

- **Step:** np.dot(centered_data, components): The dot product between centered_data and components will project the centered data onto the new principal component space. The multiplication of a matrix with shape $(512, 512)$ and a matrix with shape $(512, 50)$ will result in a matrix with shape $(512, 50)$. Thus, the shape of compressed_data will be:

$$\text{compressed\_data.shape} = (512, 50)$$

Step - 6 - Reconstruct (Decompress) the Image :

- Reverse transformation to reconstruct the image.
- Add back the mean to restore pixel intensity values.

```
# Reconstruct the image from compressed data
decompressed_data = np.dot(compressed_data, components.T) +
    mean
```

Step - 6.1 - Understanding the Reconstruction:

- **compressed_data:** This is the data that has been projected onto the principal component space, with a shape of $(\text{num\_images}, \text{n\_components})$. For example, if you keep 50 components, the shape of compressed_data will be:

$$\text{compressed\_data.shape} = (512, 50)$$

- **components:** The components matrix is the matrix of eigenvectors, which has a shape of $(\text{num\_pixels}, \text{n\_components})$. If you keep 50 components, the shape of components will be:

$$\text{components.shape} = (512, 50)$$

Step - 6.2 - Understanding the Reconstruction:

- **Step:** np.dot(compressed_data, components.T): The dot product between the compressed data and the transpose of the components matrix will reconstruct the data from the reduced-dimensional space back into the original space. The multiplication of a matrix with shape $(512, 50)$ and a matrix with shape $(50, 512)$ will result in a matrix with shape $(512, 512)$, which is the reconstructed image:

$$\text{reconstructed\_data.shape} = (512, 512)$$

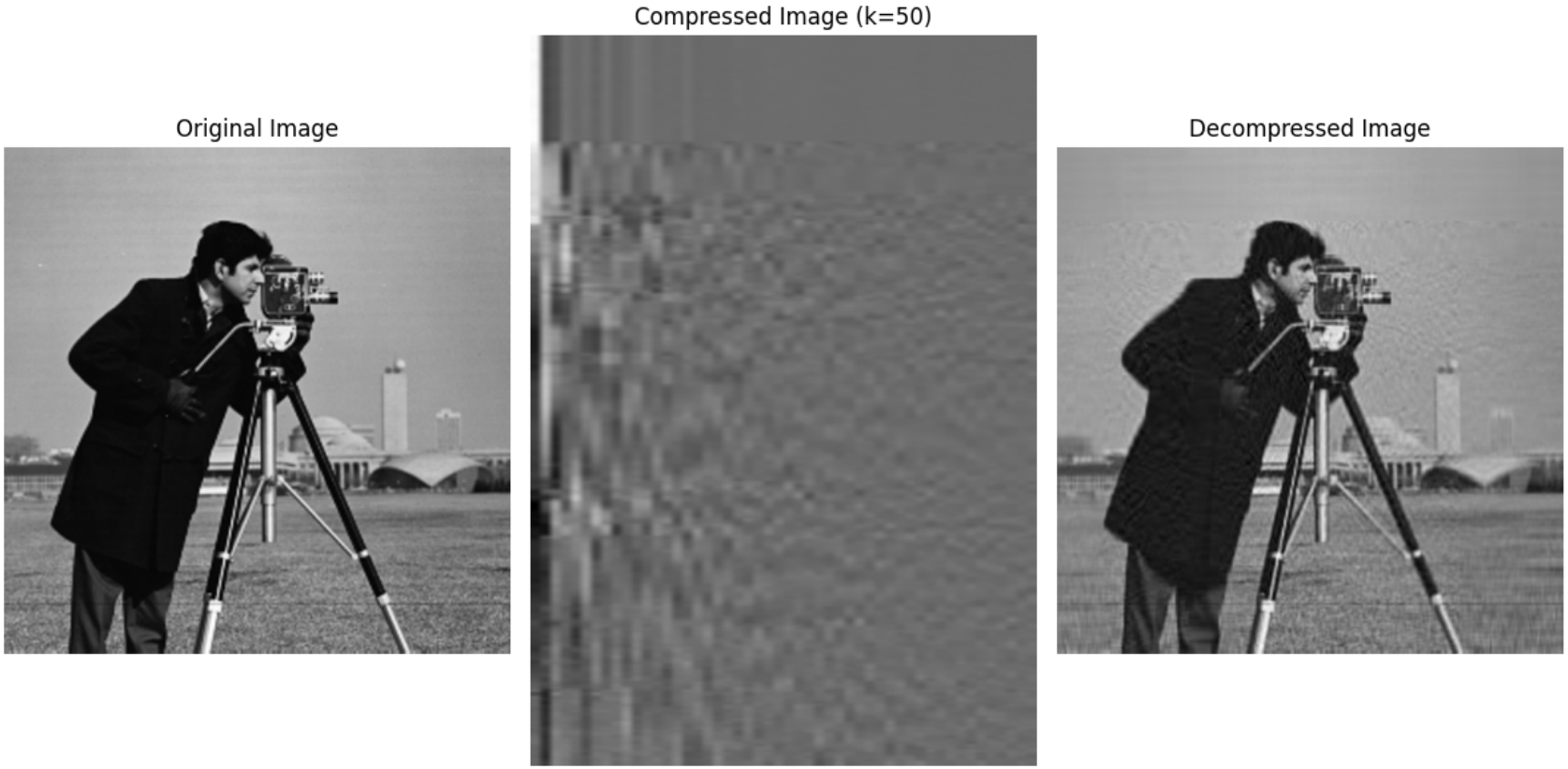
- **Reconstruction:** The reconstructed data is the approximation of the original data in the reduced principal component space. This step returns the image back to its original dimensions, but with some loss due to the dimensionality reduction.

Step - 7 - Final Visualization:

Visualizing Original, Compressed & Reconstructed Images

```
plt.figure(figsize=(12, 6))
# Original Image
plt.subplot(1, 3, 1)
plt.imshow(image_array, cmap="gray")
plt.title("Original Image")
plt.axis("off")
# Compressed Representation
plt.subplot(1, 3, 2)
plt.imshow(compressed_data, cmap="gray", aspect="auto")
plt.title(f"Compressed Image (k={k})")
plt.axis("off")
# Decompressed Image
plt.subplot(1, 3, 3)
plt.imshow(decompressed_data, cmap="gray")
plt.title("Decompressed Image")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Final Output:



Towards Worksheet - 1.

Thank You.