

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG



**Môn học: Xử lý ảnh**

**Bài tập lớn**

**Giảng viên  
Trần Văn Huy**

**: Nguyễn Tất Thắng  
: B21DCCN442**

*Hà Nội – 2024*

## Mục lục

<b>Chương 1: Getting Started with Image Processing .....</b>	<b>3</b>
1. Các thao tác xử lý hình ảnh bằng cách sử dụng slicing trên mảng numpy: .....	3
2. Changing pixel values of an image( Thay đổi giá trị pixel của hình ảnh) .....	6
<b>Chương 2: Sampling, Fourier Transform, and Convolution.....</b>	<b>8</b>
1. The Fast Fourier Transform algorithm to compute the DFT .....	8
a. The FFT with the scipy.fftpack module .....	8
b. The FFT with the numpy.fft module.....	14
<b>Chương 3 Convolution and Frequency Domain Filtering.....</b>	<b>18</b>
1. Application of the convolution theorem( Ứng dụng của định lý tích chập)....	18
<b>Chương 4 Image Enhancement .....</b>	<b>23</b>
1. Contrast stretching and histogram equalization with scikit-image.....	23
<b>Chương 5 Image Enhancement Using Derivatives.....</b>	<b>30</b>
1. Unsharp masking with the SciPy ndimage module.....	30
<b>Chương 6 Morphological Image Processing .....</b>	<b>36</b>
1. The scikit-image morphology module .....	36
2. Skeletonizing .....	39
<b>Chương 7 Extracting Image Features and Descriptors .....</b>	<b>42</b>
1. Determinant of Hessian (DoH).....	42
2. Compute HOG descriptors with scikit-image .....	48
<b>Chương 8 Image Segmentation .....</b>	<b>51</b>
1. Felzenszwalb's efficient graph-based image segmentation.....	51
2. RAG merging .....	55
<b>Chương 9 Classical Machine Learning Methods in Image Processing .....</b>	<b>60</b>
1. K-means clustering for image segmentation with color quantization .....	60
2. Supervised machine learning – image classification .....	67
<b>Chương 10 Deep Learning in Image Processing - Image Classification .....</b>	<b>75</b>
1. Classification with TF .....	75
<b>Chương 11 Deep Learning in Image Processing - Object Detection, and more .....</b>	<b>83</b>
<b>Chương 12 Additional Problems in Image Processing.....</b>	<b>85</b>
1. Content-aware image resizing with seam carving.....	85

# Chương 1: Getting Started with Image Processing

Basic image manipulations( Các thao tác cơ bản trên hình ảnh)

## 1. Các thao tác xử lý hình ảnh bằng cách sử dụng slicing trên mảng numpy:

Thuật toán tạo một mặt nạ hình tròn trên ảnh và chuyển tất cả các pixel bên ngoài hình tròn thành màu đen.

### a. Đọc ảnh từ tệp

- Ảnh được đọc từ tệp bằng thư viện mpimg (Matplotlib Image), được lưu dưới dạng một mảng NumPy (numpy ndarray).
- Mỗi pixel được lưu dưới dạng giá trị màu RGB nếu ảnh là ảnh màu.
  - o Ví dụ: lena[0, 40] trả về giá trị màu RGB của pixel tại hàng 0 và cột 40.

### b. Tạo lưới tọa độ (X, Y)

- Hình ảnh được biểu diễn dưới dạng tọa độ 2D, trong đó:
  - o Trục X (hàng) tăng từ trên xuống dưới.
  - o Trục Y (cột) tăng từ trái sang phải.
- np.ogrid được sử dụng để tạo ra hai ma trận tọa độ:
  - o X: Ma trận tọa độ hàng (dọc).
  - o Y: Ma trận tọa độ cột (ngang).
- Kích thước của X là ( $lx, 1$ ) và Y là ( $1, ly$ ), trong đó:
  - o  $lx$ : Chiều cao (số hàng) của ảnh.
  - o  $ly$ : Chiều rộng (số cột) của ảnh.

### c. Tính mặt nạ (mask)

- Tâm của hình tròn được đặt giữa ảnh tại tọa độ:

$$(cx, cy) = \left(\frac{lx}{2}, \frac{ly}{2}\right)$$

- Công thức tính khoảng cách Euclid bình phương từ mỗi pixel đến tâm

$$d^2 = (X - cx)^2 + (Y - cy)^2$$

- Mặt nạ mask là một mảng boolean có cùng kích thước với ảnh. Các phần tử trong mask có giá trị:
  - o True nếu pixel nằm ngoài hình tròn.
  - o False nếu pixel nằm trong hình tròn.
- Điều kiện xác định pixel nằm ngoài hình tròn:

$$(X - cx)^2 + (Y - cy)^2 > R^2$$

- Trong đó:

- $R^2 = \frac{lx.ly}{4}$  ( bán kính bình phương tỉ lệ với kích thước hình ảnh)
- Áp dụng mặt nạ
  - Dựa trên mask, tất cả các pixel nằm ngoài hình tròn được đặt thành [0,0,0] ( màu đen):
 
$$lena[mark,:] = 0$$
- Hiển thị kết quả

```
# Basic image manipulations

# Đọc ảnh từ đĩa
lena = mpimg.imread("D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\lena.jpg") # đọc ảnh

# Sao chép ảnh để có thể thay đổi
lena_copy = lena.copy()

# Lấy kích thước ảnh
lx, ly, _ = lena_copy.shape

# Tạo lưới tọa độ (X, Y)
X, Y = np.ogrid[0:lx, 0:ly]

# Tạo mặt nạ (mask) cho các pixel ngoài vùng tròn
mask = (X - lx / 2) ** 2 + (Y - ly / 2) ** 2 > lx * ly / 4

# Áp dụng mặt nạ và thay đổi các pixel ngoài vùng tròn thành màu đen
lena_copy[mask, :] = 0

# Hiển thị ảnh đã chỉnh sửa
plt.figure(figsize=(10, 10))
plt.imshow(lena_copy)
plt.axis('off')
plt.show()
```

TestCase 1:



TestCase 2:



TestCase 3:



Giải thích mã code:

```
# Đọc ảnh từ đĩa
lena = mpimg.imread("D:\\Learn_DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\lena.jpg") # đọc ảnh
```

- Đọc ảnh Lena từ đường dẫn chỉ định trên ổ đĩa vào biến lena.
- Thư viện: `mpimg.imread` (Matplotlib Image) đọc ảnh và chuyển đổi thành mảng NumPy (numpy ndarray).

```
# Lấy kích thước ảnh
lx, ly, _ = lena_copy.shape
```

- Xác định kích thước của ảnh (số hàng `lx`, số cột `ly`, và số kênh màu).

```
# Tạo lưới tọa độ (X, Y)
X, Y = np.ogrid[0:lx, 0:ly]
```

- Tạo lưới tọa độ đại diện cho tất cả các pixel trong ảnh.
- Thư viện: `np.ogrid` (Open Grid) tạo hai mảng:
  - o `X`: Mảng có kích thước  $(lx, 1)$  ( $lx, 1$ ), chứa tọa độ hàng (dọc).
  - o `Y`: Mảng có kích thước  $(1, ly)$  ( $1, ly$ ), chứa tọa độ cột (ngang).

```
# Tạo mặt nạ (mask) cho các pixel ngoài vùng tròn
mask = (X - lx / 2) ** 2 + (Y - ly / 2) ** 2 > lx * ly / 4
```

- Xác định pixel nào nằm ngoài vùng tròn.

## 2. Changing pixel values of an image( Thay đổi giá trị pixel của hình ảnh)

- Phân bố ngẫu nhiên của tọa độ:
  - o Sử dụng np.random.randint để chọn  $n$  tọa độ ngẫu nhiên từ phạm vi chiều cao và chiều rộng của ảnh.
  - o Mỗi pixel trong ảnh có xác suất như nhau để được chọn.
- Phân phối giá trị Salt hoặc Pepper:
  - o Mỗi pixel được chọn sẽ có xác suất 50% để trở thành:
    - Salt (trắng) với giá trị (255,255,255)
    - Pepper (đen) với giá trị (0,0,0)
- Điều này dựa trên giá trị ngẫu nhiên sinh ra từ np.random.rand():
  - o  $P(\text{màu đen}) = P(\text{ngẫu nhiên} < 0.5) = 0.5$
  - o  $P(\text{màu trắng}) = P(\text{ngẫu nhiên} \geq 0.5) = 0.5$
- Hiệu ứng trên ảnh:
  - o Pixel ban đầu có thể là bất kỳ giá trị RGB nào. Khi bị thêm nhiễu:
    - Một số pixel bị chuyển sang màu trắng, nổi bật trên nền ảnh.
    - Một số pixel bị chuyển sang màu đen, làm mất thông tin gốc của ảnh tại những vị trí đó.

```
# Adding salt and pepper noise to an image

# choose 5000 random locations inside image
imm = Image.open("D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\parrot.png")
im1 = imm.copy() # keep the original image, create a copy
n = 5000
x, y = np.random.randint(0, im.width, n), np.random.randint(0, im.height,n)
for (x,y) in zip(x,y):
| im1.putpixel((x, y), ((0,0,0) if np.random.rand() < 0.5 else (255,255,255))) # salt-and-pepper noise
im1.show()

✓ 0.1s
```

TestCase 1:



TestCase 2:



TestCase 3:



```
n = 5000
x, y = np.random.randint(0, im.width, n), np.random.randint(0, im.height, n)
```

Chọn số lượng pixel và tạo tọa độ ngẫu nhiên

- $n = 5000$ : Số lượng pixel sẽ bị thay đổi (thêm nhiễu).
- $np.random.randint$ : Tạo  $n$  giá trị ngẫu nhiên trong phạm vi:
  - Tọa độ  $x$  (cột) từ 0 đến  $im.width$  (chiều rộng của ảnh).
  - Tọa độ  $y$  (hàng) từ 0 đến  $im.height$  (chiều cao của ảnh).

```
for (x,y) in zip(x,y):
    im1.putpixel((x, y), ((0,0,0) if np.random.rand() < 0.5 else (255,255,255))) # salt-and-pepper noise
```

Thay đổi giá trị của các pixel tại tọa độ ngẫu nhiên thành màu đen hoặc trắng.

- $zip(x, y)$ : Kết hợp các tọa độ  $x$  và  $y$  thành các cặp  $(x, y)$ .
- Vòng lặp: Lặp qua từng cặp tọa độ.
- $np.random.rand() < 0.5$ : Sinh số ngẫu nhiên trong khoảng [0,1].
  - Nếu giá trị nhỏ hơn 0.5, pixel được gán màu đen (0,0,0).
  - Ngược lại, pixel được gán màu trắng (255,255,255).

- `putpixel((x, y), color)`: Thay đổi giá trị pixel tại tọa độ  $(x, y)$  trong ảnh.

## Chương 2: Sampling, Fourier Transform, and Convolution

1. The Fast Fourier Transform algorithm to compute the DFT
  - a. The FFT with the `scipy.fftpack` module

Discrete Fourier Transform (DFT)

Mục tiêu:

- Chuyển một hình ảnh (ma trận 2D biểu diễn trong miền không gian) sang miền tần số.
- Trong miền tần số, chúng ta có thể phân tích:
  - o Tần số thấp: Thông tin tổng quan, mượt mà, như các vùng màu lớn.
  - o Tần số cao: Thông tin chi tiết, ví dụ như các cạnh, biên (edges) hoặc nhiễu.

Công thức DFT:

$$F[u, v] = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f[x, y] e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})}$$

- $f[x, y]$ : Giá trị điểm ảnh tại vị trí  $(x, y)$  trong miền không gian.
- $F[u, v]$ : Hệ số Fourier tại vị trí  $(u, v)$  trong miền tần số.
- $M, N$ : Kích thước ảnh (ma trận 2D).

Cách DFT hoạt động (từng bước):

1. Ma trận ảnh gốc:
  - o Mỗi giá trị  $f[x, y]$  trong ảnh là một giá trị cường độ xám hoặc giá trị màu tại tọa độ  $(x, y)$ .
2. Chuyển đổi sang miền tần số:
  - o Mỗi vị trí  $(u, v)$  trong miền tần số là một tổng hợp tuyến tính (weighted sum) của toàn bộ giá trị  $f[x, y]$  trong miền không gian.
  - o Tọa độ  $(u, v)$  trong miền tần số đại diện cho một tần số cụ thể:

- $u = 0, v = 0$  : Tần số thấp nhất (thông tin trung bình của ảnh).
- $u, v$  lớn: Tần số cao (chi tiết hoặc nhiễu).

### 3. Hệ số Fourier $F[u, v]$ :

- Hệ số này biểu diễn biên độ (amplitude) và pha (phase) của tín hiệu tần số.

Thuật toán FFT (Fast Fourier Transform)

FFT là gì?

- Là thuật toán tối ưu để tính toán DFT nhanh hơn.
- FFT sử dụng phương pháp chia để trị (divide and conquer) để giảm độ phức tạp tính toán từ  $O(N^2)$  xuống  $O(N \log N)$

Nguyên tắc hoạt động của FFT:

- Phân chia dữ liệu:
  - Tách tín hiệu ban đầu (hoặc ma trận ảnh) thành các phần nhỏ hơn.
  - Chia thành các phần chẵn và lẻ.
- Tính toán đệ quy:
  - Áp dụng FFT cho từng phần nhỏ.
- Kết hợp kết quả:
  - Kết hợp kết quả từ các phần nhỏ để tạo thành DFT của tín hiệu ban đầu.

FFT dựa trên 2 tính chất quan trọng của hàm mũ phức:

- Tính chu kỳ:

$$e^{-i\frac{2\pi(k+N)n}{N}} = e^{-i\frac{2\pi kn}{N}}$$

Điều này cho phép tính toán các hệ số Fourier không cần lặp lại các phép tính đã thực hiện.

- Tính đối xứng:

$$e^{-i\frac{2\pi kn}{N}} = \cos\left(\frac{2\pi kn}{N}\right) - i \sin\left(\frac{2\pi kn}{N}\right)$$

Hàm cos và sin có thể được tái sử dụng, giúp giảm thiểu số phép tính.

```

import numpy as np
from PIL import Image
import matplotlib.pyplot as pylab
# Hàm tính SNR (Signal-to-Noise Ratio)
def signaltonoise(image, axis=None):
    array = np.asarray(image)
    mean = np.mean(array)
    std = np.std(array)
    return mean / std if std != 0 else float('inf')

# Đọc ảnh và chuyển đổi sang ảnh xám (grayscale)
im = np.array(Image.open('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\rhino.jpg').convert('L')) # Đường dẫn tới ảnh

# Tính SNR của ảnh gốc
snr = signaltonoise(im, axis=None)
print(f'SNR for the original image = {snr}')

# Thực hiện FFT và IFFT
freq = np.fft.fft2(im) # Chuyển đổi Fourier nhanh 2D
im1 = np.fft.ifft2(freq).real # Chuyển ngược Fourier (phản thực)

# Tính SNR của ảnh tái tạo
snr_reconstructed = signaltonoise(im1, axis=None)
print(f'SNR for the image obtained after reconstruction = {snr_reconstructed}')

# Kiểm tra xem ảnh ban đầu và ảnh tái tạo có tương tự nhau không
assert np.allclose(im, im1), "The original and reconstructed images are not numerically close."

# Hiển thị ảnh gốc và ảnh tái tạo
pylab.figure(figsize=(20, 10))
# Ảnh gốc
pylab.subplot(121)
pylab.imshow(im, cmap='gray')
pylab.axis('off')
pylab.title('Original Image', size=20)

# Ảnh tái tạo
pylab.subplot(122)
pylab.imshow(im1, cmap='gray')
pylab.axis('off')
pylab.title('Image obtained after reconstruction', size=20)
pylab.show()

```

```

# Hàm tính SNR (Signal-to-Noise Ratio)
def signaltonoise(image, axis=None):
    array = np.asarray(image)
    mean = np.mean(array)
    std = np.std(array)
    return mean / std if std != 0 else float('inf')

```

Hàm tính SNR (Signal-to-Noise Ratio) của ảnh

- SNR đo tỷ lệ tín hiệu trên nhiễu của ảnh. Công thức:

$$SNR = \frac{Mean}{Standard Deviation}$$

- Ảnh có SNR cao hơn nghĩa là ảnh ít nhiễu hơn, chất lượng tốt hơn.

```

# Đọc ảnh và chuyển đổi sang ảnh xám (grayscale)
im = np.array(Image.open('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\rhino.jpg').convert('L'))

```

Chuyển ảnh sang grayscale (ảnh xám). Chuyển đổi ảnh từ định dạng PIL sang mảng NumPy để xử lý.

```

# Tính SNR của ảnh gốc
snr = signaltonoise(im, axis=None)
print(f'SNR for the original image = {snr}')

```

## Tính SNR ảnh gốc

```
# Thực hiện FFT và IFFT
freq = np.fft.fft2(im) # Chuyển đổi Fourier nhanh 2D
im1 = np.fft.ifft2(freq).real # Chuyển ngược Fourier (phần thực)
```

## Thực hiện FFT (Fast Fourier Transform)

- np.fft.fft2(): Thực hiện phép biến đổi Fourier nhanh 2D, chuyển đổi ảnh từ miền không gian sang miền tần số.
- np.fft.ifft2(): Thực hiện phép biến đổi Fourier ngược, tái tạo lại ảnh từ miền tần số về miền không gian.
- .real: Lấy phần thực của kết quả (vì kết quả của FFT có thể là số phức)

```
# Tính SNR của ảnh tái tạo
snr_reconstructed = signaltonoise(im1, axis=None)
print(f'SNR for the image obtained after reconstruction = {snr_reconstructed}')
```

- Tính SNR của ảnh sau khi thực hiện FFT và IDFT.
- So sánh giá trị SNR này với ảnh gốc để xem quá trình tái tạo có làm mất mát thông tin hay không.



Vẽ biểu đồ phô tần số:

```

# Plotting the frequency spectrum

import numpy as np
import matplotlib.pyplot as plt
from scipy import fftpack as fp

# Giả sử 'image' là hình ảnh đầu vào ở dạng mảng numpy 2D
im = np.array(Image.open('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\\\rhino.jpg').convert('L'))

# Thực hiện biến đổi Fourier 2D
freq = fp.fft2(im)

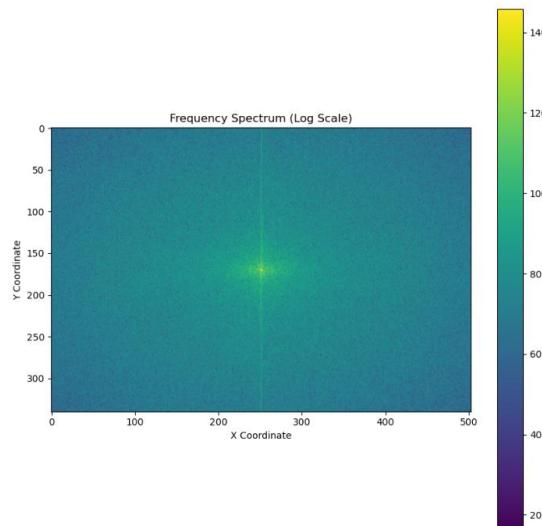
# Dịch chuyển các thành phần tần số để tần số thấp ở trung tâm
freq2 = fp.fftshift(freq)

# Tính phổ tần số và hiển thị logarit của phổ Fourier
plt.figure(figsize=(10, 10))
plt.imshow((20 * np.log10(0.1 + np.abs(freq2))), cmap='viridis') # Sử dụng cmap 'viridis' để giống hình mẫu
plt.colorbar()
plt.xlabel("X Coordinate")
plt.ylabel("Y Coordinate")
plt.title("Frequency Spectrum (Log Scale)")
plt.show()

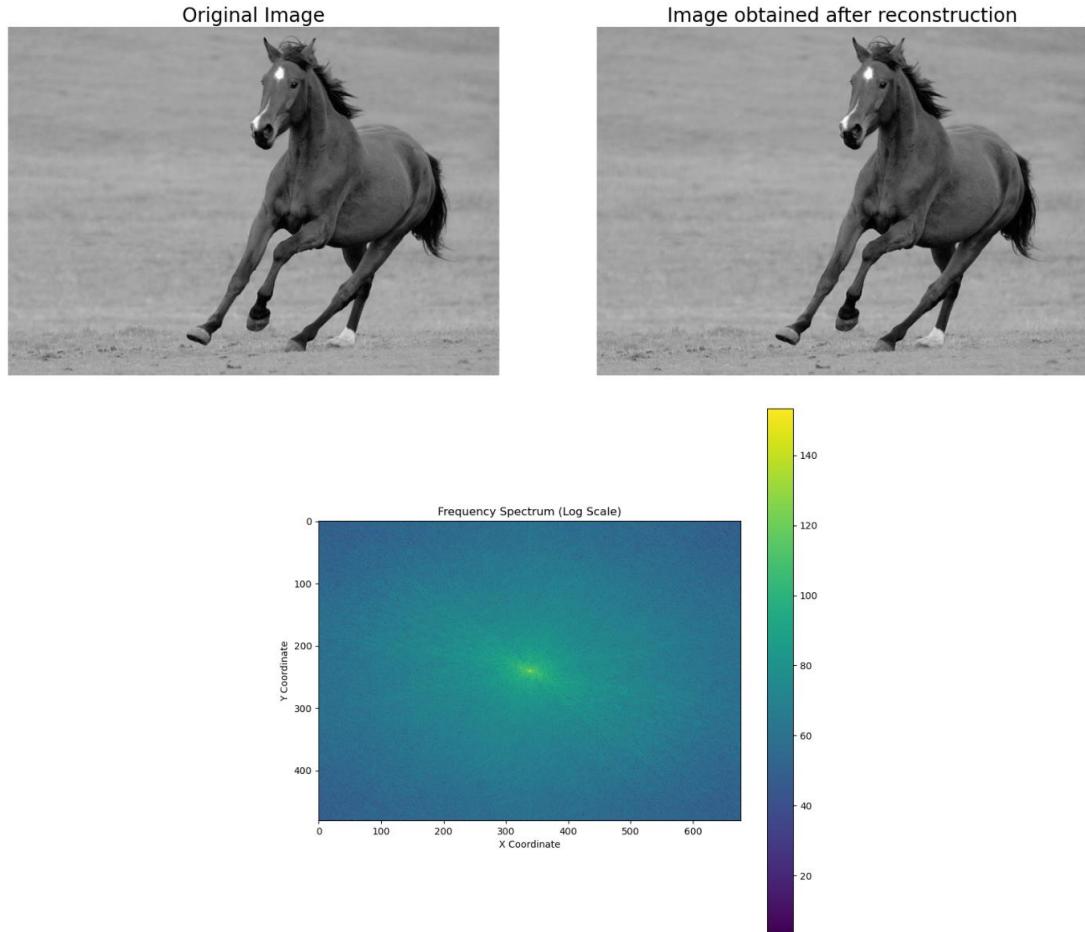
```

Biểu diễn phổ tần số của ảnh, giúp hiểu rõ hơn các thông tin tần số trong ảnh.

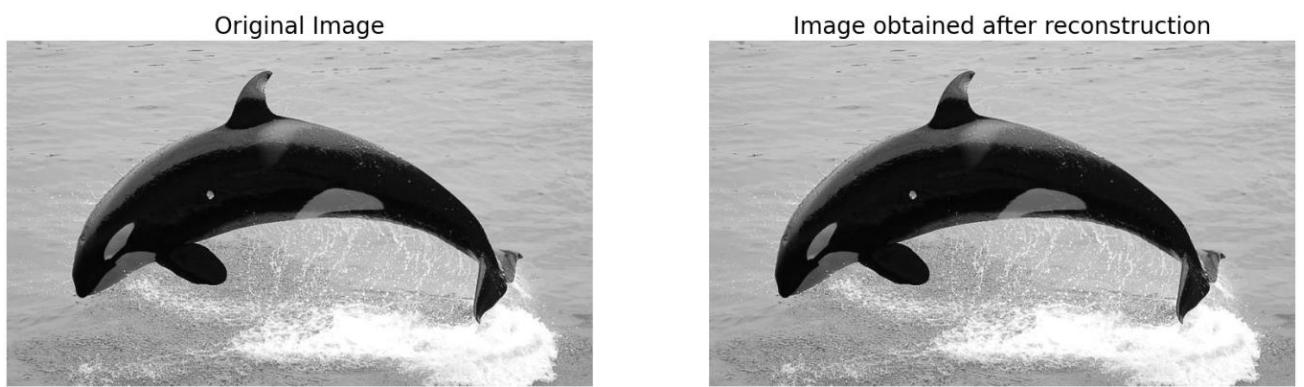
Các tần số thấp (ở trung tâm của phổ) chứa thông tin tổng quát, trong khi các tần số cao (ở ngoài rìa) đại diện cho các chi tiết nhỏ và nhiễu.

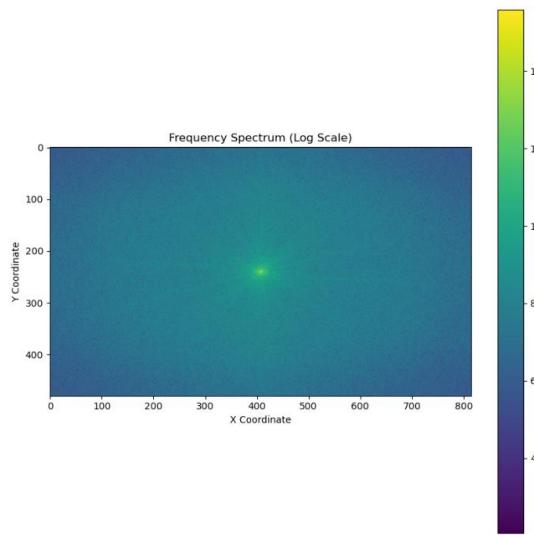


Test case 2: horse



Testcase 3: whale





### b. The FFT with the numpy.fft module

Computing the magnitude and phase of a DFT(Tính toán độ lớn và pha của DFT)

```
# Computing the magnitude and phase of a DFT
# Đọc ảnh và chỉ lấy 3 kênh đầu tiên nếu ảnh có kênh alpha
image = imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\house.png')
if image.shape[-1] == 4: # Kiểm tra xem ảnh có 4 kênh không
|   image = image[:, :, :3] # Giữ lại 3 kênh đầu tiên (RGB)
# Chuyển ảnh sang thang độ xám
im1 = rgb2gray(image)

# Thực hiện biến đổi Fourier 2D
freq1 = fp.fft2(im1)

# Tính độ lớn phổ Fourier
magnitude_spectrum = 20 * np.log10(0.01 + np.abs(fp.fftshift(freq1)))

# Tính pha của phổ Fourier
phase_spectrum = np.angle(fp.fftshift(freq1))

# Tái tạo lại ảnh từ phổ Fourier
im1_reconstructed = fp.ifft2(freq1).real

# Hiển thị kết quả
plt.figure(figsize=(12, 10))

# Ảnh gốc
plt.subplot(2, 2, 1)
plt.imshow(im1, cmap='gray')
plt.title('Original Image', size=20)

# Độ lớn phổ Fourier
plt.subplot(2, 2, 2)
plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('FFT Spectrum Magnitude', size=20)

# Pha phổ Fourier
plt.subplot(2, 2, 3)
plt.imshow(phase_spectrum, cmap='gray')
plt.title('FFT Phase', size=20)

# Ảnh tái tạo từ phổ Fourier
plt.subplot(2, 2, 4)
plt.imshow(np.clip(im1_reconstructed, 0, 255), cmap='gray')
plt.title('Reconstructed Image', size=20)
plt.show()
```

```
# Chuyển ảnh sang thang độ xám  
im1 = rgb2gray(image)
```

```
# Thực hiện biến đổi Fourier 2D  
freq1 = fp.fft2(im1)
```

Chuyển ảnh từ RGB sang grayscale (ảnh mức xám), giúp đơn giản hóa việc tính toán.

Biến đổi ảnh từ miền không gian (spatial domain) sang miền tần số (frequency domain).

Kết quả là một mảng phức, mỗi phần tử chứa biên độ và pha của các thành phần tần số trong ảnh

```
# Tính độ lớn phổ Fourier  
magnitude_spectrum = 20 * np.log10(0.01 + np.abs(fp.fftshift(freq1)))
```

Tính độ lớn phổ Fourier (Magnitude Spectrum)

- `np.abs(freq1):`
  - o Lấy giá trị biên độ của các hệ số Fourier từ kết quả freq1.
  - o Hệ số Fourier là số phức (complex number) với phần thực và phần ảo. Biên độ được tính bằng công thức:
$$|freq1| = \sqrt{Re(freq1)^2 + Im(freq1)^2}$$
- `fp.fftshift(freq1):` Dịch chuyển tần số thấp (DC component) từ góc trên-trái (vị trí mặc định trong FFT) về trung tâm để dễ quan sát phổ.
- `np.log10(0.01 + np.abs(...)):`
  - o Chuyển đổi biên độ sang thang logarit, giúp biểu diễn các giá trị tần số lớn và nhỏ trên cùng một đồ thị.
  - o Thêm giá trị 0.01 để tránh lỗi logarit của số 0.
- `20 *:`Hệ số nhân 20 nhằm biểu diễn giá trị theo thang decibel (dB), một tiêu chuẩn trong xử lý tín hiệu:

$$\text{Magnitude (dB)} = 20 \cdot \log_{10}(\text{Amplitude})$$

```
# Tính pha của phổ Fourier  
phase_spectrum = np.angle(fp.fftshift(freq1))
```

Tính pha của phổ Fourier (Phase Spectrum):

`np.angle(freq1):`

- Lấy góc pha của các hệ số Fourier.
- Pha là góc tạo bởi vector số phức với trục thực trong mặt phẳng phức, tính bằng:

$$Phase = \arctan\left(\frac{Im(freq1)}{Re(freq1)}\right)$$

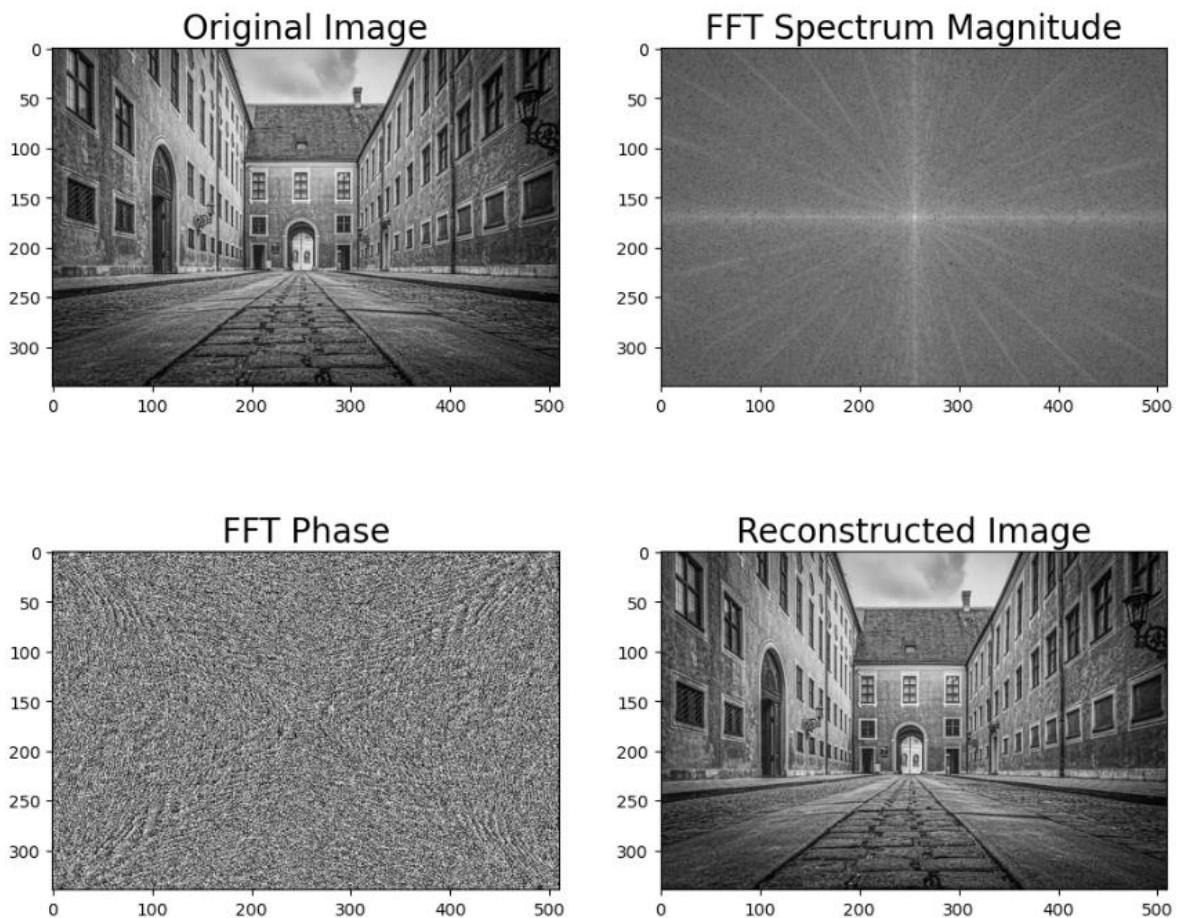
```
# Tái tạo lại ảnh từ phổ Fourier
im1_reconstructed = fp.ifft2(freq1).real
```

Tái tạo lại ảnh từ phổ Fourier

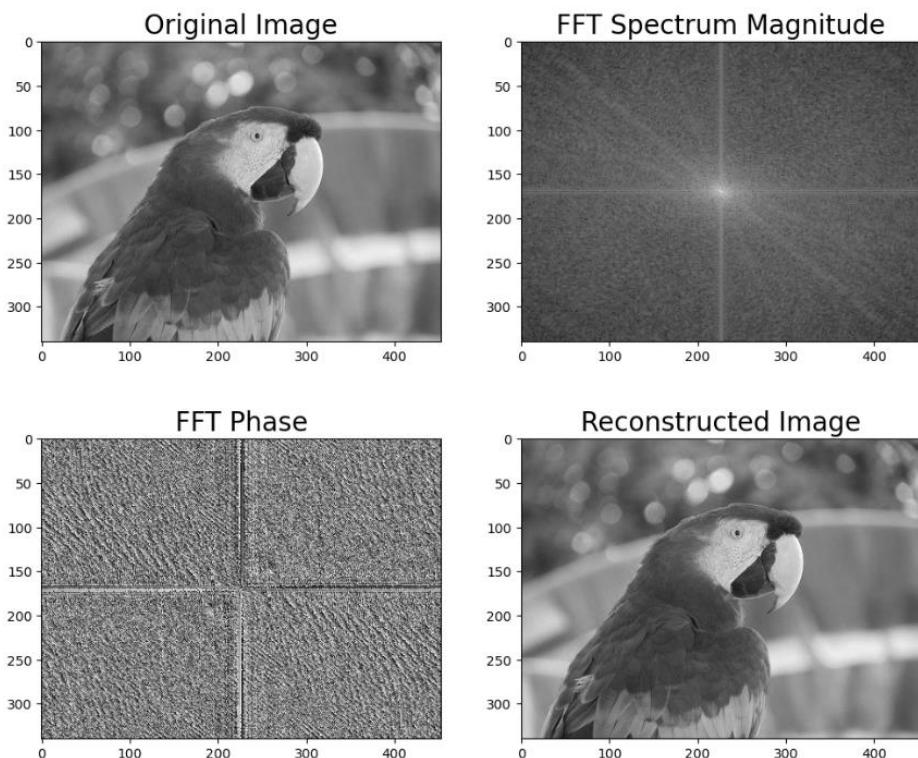
Tính IDFT (Inverse Discrete Fourier Transform) để chuyển từ miền tần số về miền không gian.

Hiển thị kết quả:

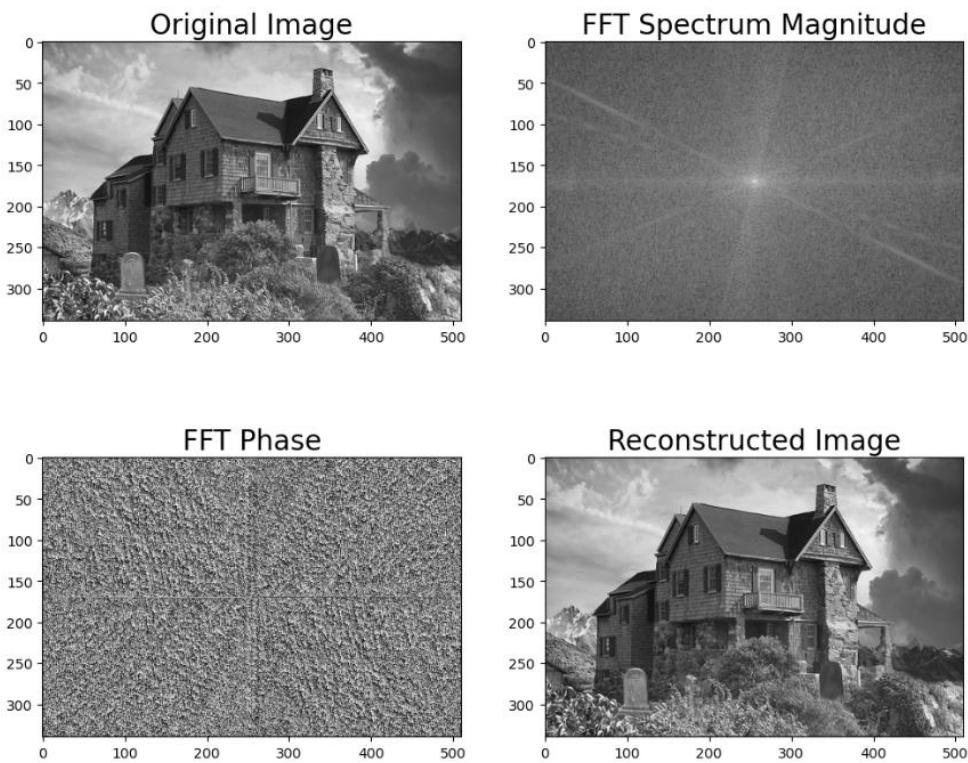
Testcase 1:



TestCase 2:



TestCase 3:



## Chương 3 Convolution and Frequency Domain Filtering

### 1. Application of the convolution theorem( Ứng dụng của định lý tích chập)

Frequency domain Gaussian blur filter with numpy fft

- Định lý tích chập (Convolution Theorem):
  - o Tích chập trong miền không gian (spatial domain) tương đương với phép nhân từng phần tử trong miền tần số (frequency domain).
- Điều này có nghĩa là:
  - o Nếu bạn áp dụng một bộ lọc (filter) lên hình ảnh trong miền không gian, thì phép tính này có thể được chuyển đổi thành phép nhân đơn giản trong miền tần số.
  - o Sử dụng miền tần số giúp tăng tốc xử lý, đặc biệt khi làm việc với các hình ảnh hoặc kernel lớn.
- Các bước thực hiện thuật toán trong miền tần số

Thuật toán để tích chập một hình ảnh với một kernel (bộ lọc) trong miền tần số được thực hiện như sau:

Input:

- $f(x,y)$ : Hình ảnh gốc.
- $h(x,y)$ : Kernel (bộ lọc, ví dụ: Gaussian blur).

Các bước chính:

- Chuyển đổi hình ảnh và kernel sang miền tần số:
  - o Áp dụng DFT (Discrete Fourier Transform) để chuyển  $f(x,y)$  và  $h(x,y)$  sang miền tần số:

$$F(u, v) = DFT(f(x, y)), \quad H(u, v) = DFT(h(x, y))$$

- o Kết quả là  $F(u, v)$  và  $H(u, v)$ , là biểu diễn của hình ảnh và kernel trong miền tần số
- Nhân từng phần tử trong miền tần số:
  - o Áp dụng phép nhân điểm (element-wise multiplication):

$$G(u, v) = F(u, v) \cdot H(u, v)$$

- o  $G(u, v)$  là kết quả tích chập trong miền tần số.
- Chuyển kết quả về miền không gian:
  - o Áp dụng IDFT (Inverse Discrete Fourier Transform) để chuyển  $G(u, v)$  về miền không gian:

$$g(x, y) = IDFT(G(u, v))$$

- o  $g(x, y)$ : Hình ảnh sau khi áp dụng bộ lọc.

```

def main():
    # Đường dẫn ảnh
    image_path = "D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\lena.jpg"
    # Đọc ảnh và chuyển sang grayscale
    im = np.mean(imread(image_path), axis=2) / 255.0 # Chuẩn hóa giá trị pixel về [0, 1]
    # Tạo kernel Gaussian trong miền không gian
    sigma = 10 # Độ rộng của Gaussian
    gauss_kernel = np.outer(
        gaussian_1d(im.shape[0], sigma), # Gaussian theo chiều dọc
        gaussian_1d(im.shape[1], sigma) # Gaussian theo chiều ngang
    )
    gauss_kernel = gauss_kernel / np.sum(gauss_kernel) # Chuẩn hóa kernel
    # Chuyển đổi ảnh và kernel sang miền tần số
    freq = fft2(im)
    freq_kernel = fft2(ifftshift(gauss_kernel)) # Dịch kernel về đúng tâm
    # Áp dụng định lý tích chập (nhấn từng phần tử trong miền tần số)
    convolved = freq * freq_kernel
    # Chuyển kết quả về miền không gian (IDFT)
    im_filtered = np.abs(ifft2(convolved))
    # Hiển thị các kết quả
    display_results(im, gauss_kernel, im_filtered, freq, freq_kernel, convolved)
def gaussian_1d(size, sigma):
    """Hàm tạo Gaussian kernel 1D."""
    x = np.linspace(-size // 2, size // 2, size)
    gauss = np.exp(-x**2 / (2 * sigma**2))
    return gauss
def to_spectrum_display(spectrum):
    """Tăng độ tương phản của phổ bằng cách sử dụng log."""
    spectrum = np.abs(ifftshift(spectrum)) # Lấy giá trị tuyệt đối và dịch tâm
    return np.log(1 + spectrum)
def display_results(im, gauss_kernel, im_filtered, freq, freq_kernel, convolved):
    """Hiển thị kết quả."""
    plt.figure(figsize=(15, 10))
    # Hình ảnh gốc
    plt.subplot(2, 3, 1)
    plt.imshow(im, cmap='gray')
    plt.title("Original Image", fontsize=15)
    plt.axis("off")
    # Kernel Gaussian trong miền không gian
    plt.subplot(2, 3, 2)
    plt.imshow(gauss_kernel, cmap='gray')
    plt.title("Gaussian Kernel", fontsize=15)
    plt.axis("off")
    # Hình ảnh sau khi lọc
    plt.subplot(2, 3, 3)
    plt.imshow(im_filtered, cmap='gray')
    plt.title("Output Image", fontsize=15)
    plt.axis("off")
    # Phổ tần số của hình ảnh gốc
    plt.subplot(2, 3, 4)
    plt.imshow(to_spectrum_display(freq), cmap='gray')
    plt.title("Original Image Spectrum", fontsize=15)
    plt.axis("off")
    # Phổ tần số của kernel Gaussian
    plt.subplot(2, 3, 5)
    plt.imshow(to_spectrum_display(freq_kernel), cmap='gray')
    plt.title("Gaussian Kernel Spectrum", fontsize=15)
    plt.axis("off")
    # Phổ tần số của hình ảnh sau khi tích chập
    plt.subplot(2, 3, 6)
    plt.imshow(to_spectrum_display(convolved), cmap='gray')
    plt.title("Output Image Spectrum", fontsize=15)
    plt.axis("off")
    # Tinh chỉnh khoảng cách giữa các subplot
    plt.subplots_adjust(wspace=0.3, hspace=0.3)
    plt.show()
if __name__ == "__main__":
    main()

```

```

# Tạo kernel Gaussian trong miền không gian
sigma = 10 # Độ rộng của Gaussian
gauss_kernel = np.outer(
    gaussian_1d(im.shape[0], sigma), # Gaussian theo chiều dọc
    gaussian_1d(im.shape[1], sigma) # Gaussian theo chiều ngang
)
gauss_kernel = gauss_kernel / np.sum(gauss_kernel) # Chuẩn hóa kernel

```

- Hàm gaussian\_1d: Tạo kernel Gaussian 1D dựa trên công thức toán học cho Gaussian, dùng  $\sigma$  để điều chỉnh độ mờ.
- Tạo kernel 2D: Nhân ma trận kernel 1D với chính nó để tạo kernel Gaussian 2D.
- Chuẩn hóa kernel: Đảm bảo tổng giá trị các phần tử trong kernel bằng 1 để bảo toàn độ sáng khi làm mịn ảnh.

```

# Chuyển đổi ảnh và kernel sang miền tần số
freq = fft2(im)
freq_kernel = ifftshift(fft2(freq)) # Dịch kernel về đúng tâm

```

- Chuyển kernel từ không gian không tần số sang miền tần số, sử dụng cho phân tích phô Fourier.

```

# Áp dụng định lý tích chập (nhân từng phần tử trong miền tần số)
convolved = freq * freq_kernel

```

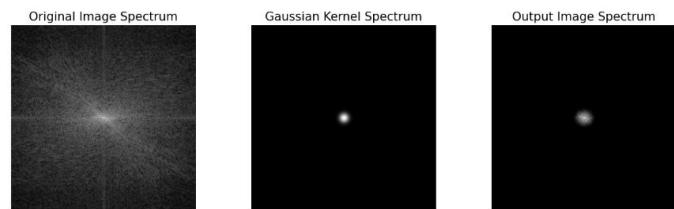
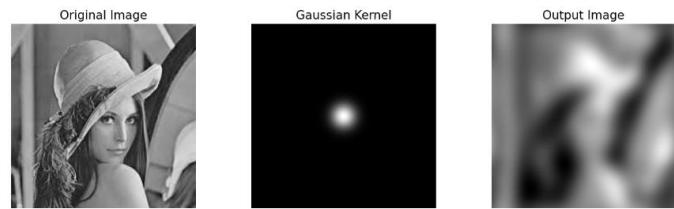
- Nhân tần số: Thực hiện tích chập trong miền tần số bằng cách nhân phô Fourier của ảnh và kernel.

Mục đích chính: Tích hợp Gaussian Blur (làm mờ Gaussian) vào ảnh, phân tích phô tần số, và minh họa quá trình qua đồ thị.

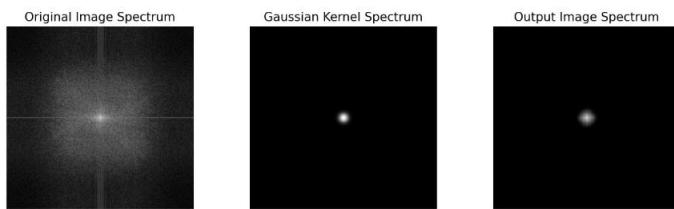
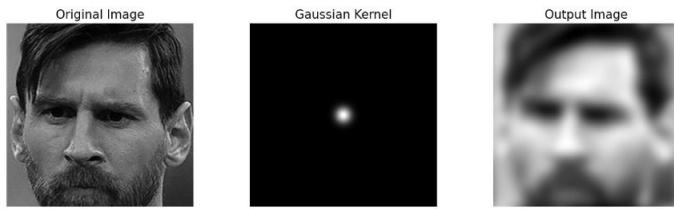
Ứng dụng thực tế: Làm mịn ảnh, giảm nhiễu, và kiểm tra ảnh hưởng của kernel Gaussian trong không gian miền tần số.

In ra kết quả:

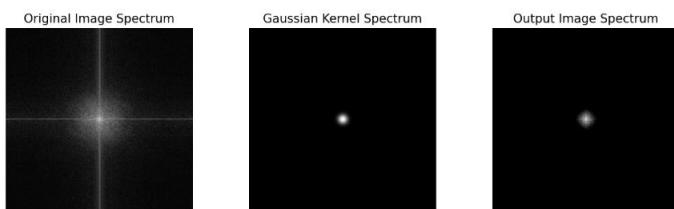
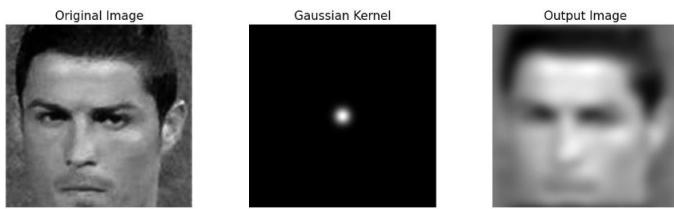
Testcase 1:



TestCase 2:



TestCase 3:



## Chương 4 Image Enhancement

### 1. Contrast stretching and histogram equalization with scikit-image

#### Cân bằng histogram (Histogram Equalization)

- Nguyên tắc hoạt động:

- Phân bố lại các giá trị cường độ pixel của ảnh sao cho histogram của ảnh đều ra gần như phẳng (đồng nhất).
- Điều này làm nổi bật các chi tiết ở vùng sáng và tối, giúp ảnh có độ tương phản cao hơn.

- Công thức chuyển đổi:

- Dựa trên hình ảnh, ánh xạ giá trị pixel rkr\_krk trong ảnh gốc thành sks\_ksk trong ảnh kết quả thông qua phân phối xác suất tích lũy (CDF - Cumulative Distribution Function):

$$s_k = T(r_k) = \sum_{j=0}^k P_r(r_j) = \sum_{j=0}^k \frac{n_j}{N}$$

- $P_r(r_j)$ : Xác suất xuất hiện của mức xám
- $n_j$ : Số lượng pixel có giá trị  $r_j$
- $N$ : Tổng số pixel trong ảnh
- Kết quả là ảnh đầu ra có histogram trải rộng hơn (độ tương phản cao hơn).

#### Cân bằng histogram thích nghi (Adaptive Histogram Equalization - CLAHE)

- Nguyên tắc hoạt động:

- Chia ảnh thành các khối nhỏ (tiles).
- Thực hiện cân bằng histogram cục bộ trên từng khối, sau đó kết hợp các khối lại.
- Tránh tình trạng quá sáng hoặc quá tối ở một số vùng bằng cách đặt giới hạn (clip limit).

- Ưu điểm:

- Làm nổi bật chi tiết tốt hơn so với cân bằng toàn cục (global histogram equalization), đặc biệt ở ảnh có độ tương phản thấp hoặc có các vùng tối và sáng rõ rệt.

```

from skimage.color import rgb2gray
from skimage import exposure
from skimage.io import imread
import pylab

# Đọc ảnh và chuyển đổi sang grayscale
img = rgb2gray(imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\earthfromsky.jpg'))

# Histogram Equalization
img_eq = exposure.equalize_hist(img)

# Adaptive Histogram Equalization
img_adapteq = exposure.equalize_adapthist(img, clip_limit=0.03)

# Hiển thị ảnh bằng thang màu grayscale
pylab.gray()

# Danh sách ảnh và tiêu đề
images = [img, img_eq, img_adapteq]
titles = [
    'Original Input (Earth from Sky)',
    'After Histogram Equalization',
    'After Adaptive Histogram Equalization'
]

# Hiển thị từng ảnh với tiêu đề
for i in range(3):
    pylab.figure(figsize=(20, 10))
    pylab.imshow(images[i], cmap='gray')
    pylab.title(titles[i], size=15)
    pylab.axis('off') # Tắt trục tọa độ

# Hiển thị histogram của từng ảnh
pylab.figure(figsize=(15, 5))
for i in range(3):
    pylab.subplot(1, 3, i + 1)
    pylab.hist(images[i].ravel(), color='g', bins=256)
    pylab.title(titles[i], size=15)

pylab.show()

```

Thực hiện thuật toán

```

# Histogram Equalization
img_eq = exposure.equalize_hist(img)

# Adaptive Histogram Equalization
img_adapteq = exposure.equalize_adapthist(img, clip_limit=0.03)

```

`equalize_hist`: Thực hiện cân bằng histogram toàn cục trên ảnh. Kết quả được lưu trong biến `img_eq`.

`equalize_adapthist`: Thực hiện cân bằng histogram thích nghi.

clip\_limit=0.03: Giới hạn giá trị tối đa của histogram để tránh quá sáng hoặc quá tối trong một số vùng.

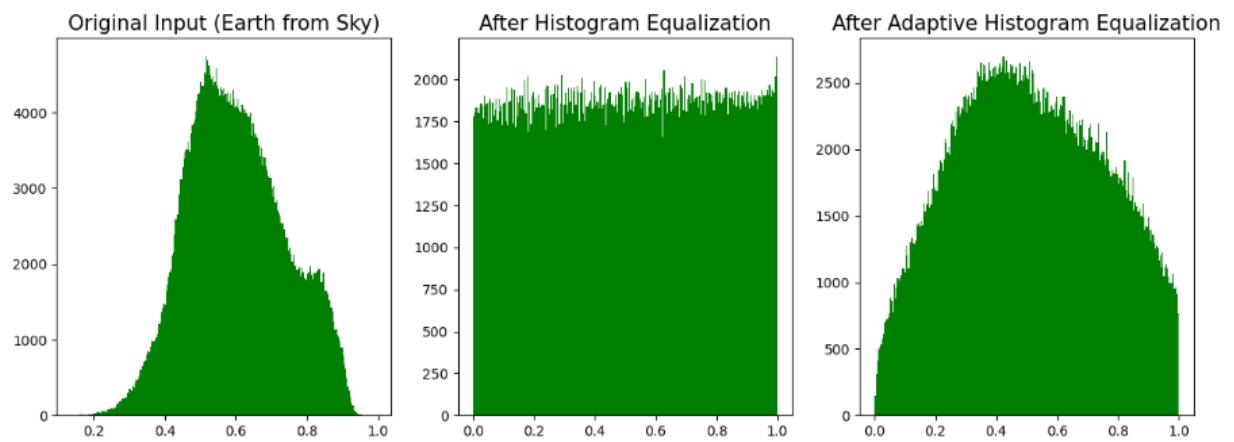
```
# Hiển thị từng ảnh với tiêu đề
for i in range(3):
    pylab.figure(figsize=(20, 10))
    pylab.imshow(images[i], cmap='gray')
    pylab.title(titles[i], size=15)
    pylab.axis('off') # Tắt trục tọa độ

# Hiển thị histogram của từng ảnh
pylab.figure(figsize=(15, 5))
for i in range(3):
    pylab.subplot(1, 3, i + 1)
    pylab.hist(images[i].ravel(), color='g', bins=256)
    pylab.title(titles[i], size=15)
```

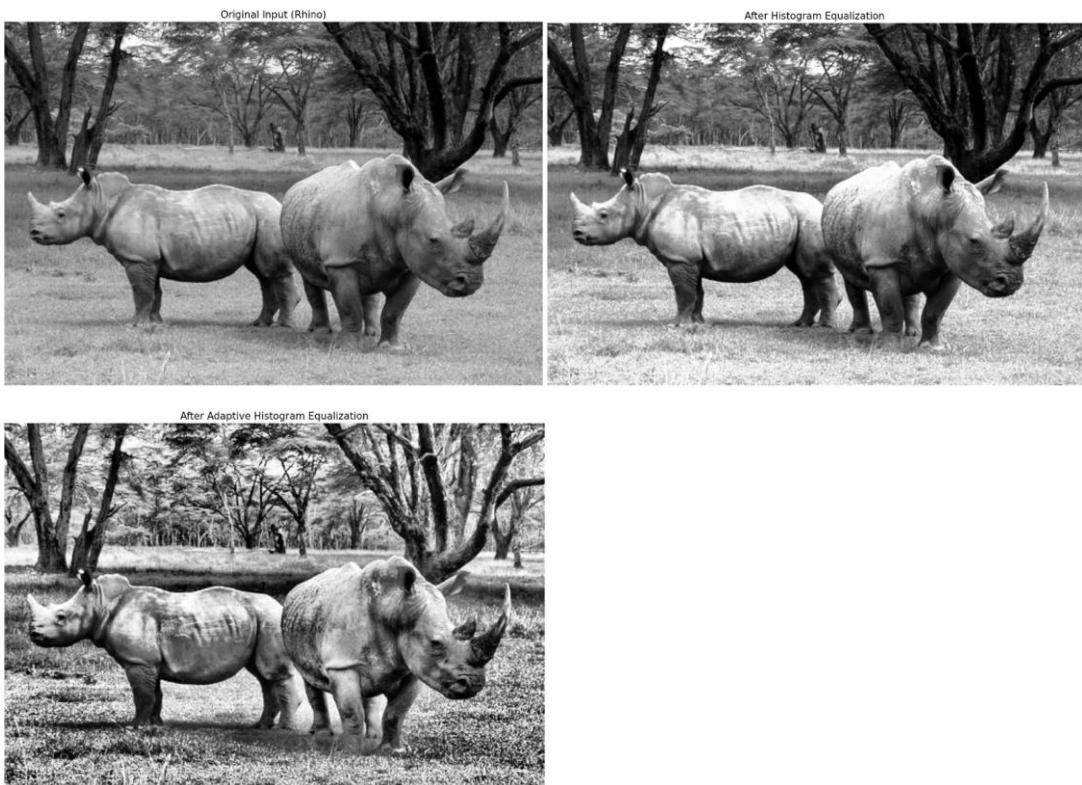
- pylab.hist:
  - images[i].ravel(): Trải phẳng ảnh thành một mảng 1D (các pixel).
  - color='g': Histogram được vẽ với màu xanh lá.
  - bins=256: Chia mức xám thành 256 khoảng.
- pylab.show(): Hiển thị các biểu đồ histogram.

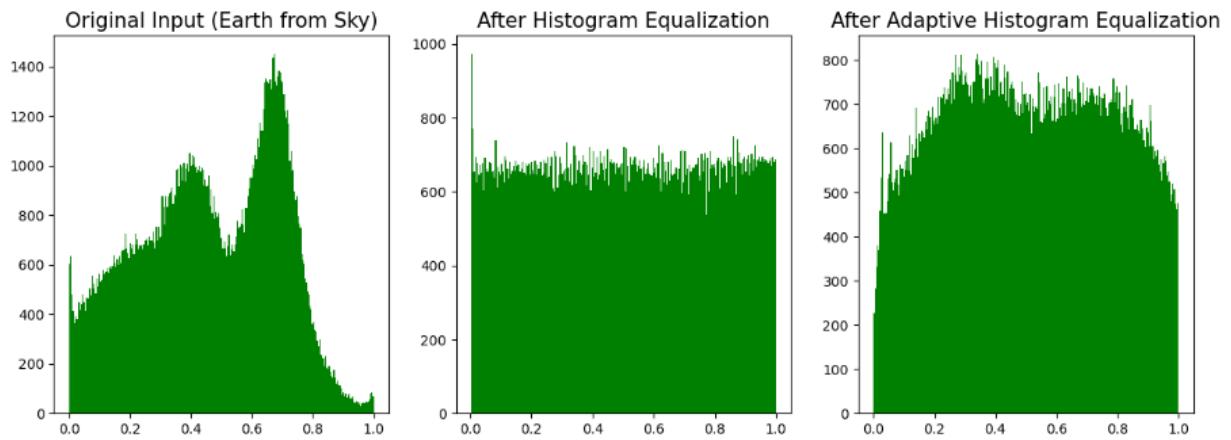
TestCase 1:



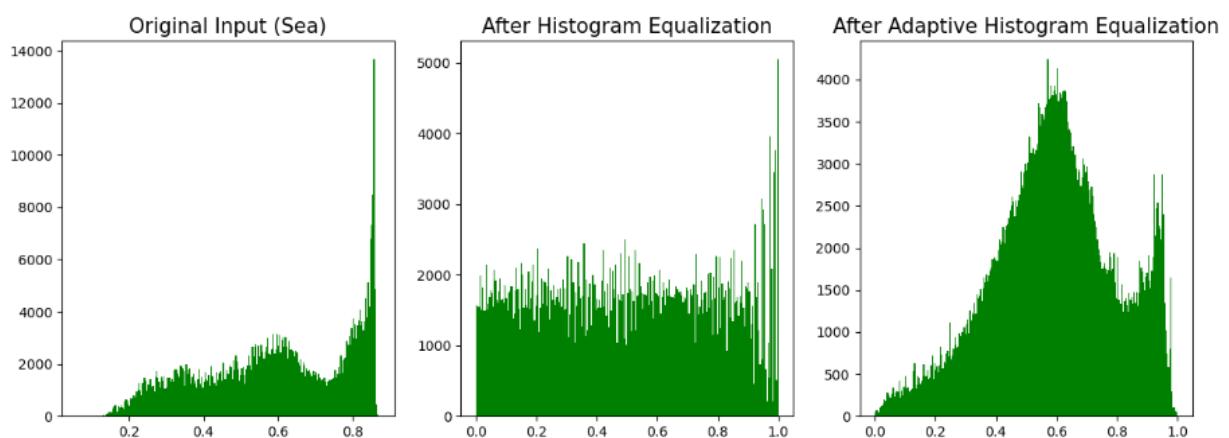


TestCase 2:





TestCase 3:



```

# Hàm để hiển thị ảnh cùng histogram và CDF
def plot_image_and_hist(image, axes, bins=256):
    image = img_as_float(image)
    axes_image, axes_hist = axes
    axes_cdf = axes_hist.twinx()

    # Hiển thị ảnh
    axes_image.imshow(image, cmap='gray', vmin=0, vmax=1)
    axes_image.set_axis_off()

    # Hiển thị histogram
    axes_hist.hist(image.ravel(), bins=bins, histtype='step', color='black')
    axes_hist.set_xlim(0, 1)
    axes_hist.set_xlabel('Pixel Intensity', size=10)
    axes_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
    axes_hist.set_yticks([])

    # Tính và hiển thị CDF
    image_cdf, bins = exposure.cumulative_distribution(image, bins)
    axes_cdf.plot(bins, image_cdf, 'r')
    axes_cdf.set_yticks([])

    return axes_image, axes_hist, axes_cdf

```

Hàm này hiển thị ảnh, biểu đồ histogram và CDF (Cumulative Distribution Function).

- **Hiển thị ảnh:**
  - o axes\_image.imshow: Hiển thị ảnh dưới dạng grayscale.
  - o cmap='gray': Hiển thị ảnh với thang độ xám.
  - o vmin=0, vmax=1: Giới hạn giá trị pixel từ 000 đến 111.
- **Vẽ histogram:**
  - o axes\_hist.hist: Tạo histogram dựa trên giá trị pixel của ảnh.
  - o ravel(): Trải phẳng ảnh thành mảng 1D.
- **Tính và vẽ CDF:**
  - o exposure.cumulative\_distribution: Tính CDF (hàm phân phối tích lũy) của ảnh.
  - o axes\_cdf.plot: Vẽ đường CDF màu đỏ.

## Các phương pháp xử lý ảnh

### 1. Contrast Stretching:

```

# Contrast Stretching
im_rescale = exposure.rescale_intensity(im, in_range=(0, 100), out_range=(0, 1))

```

- Kéo dãn giá trị pixel từ khoảng (0,100) sang (0,1)

- Phương pháp này tăng cường độ tương phản bằng cách dàn trải giá trị pixel.

## 2. Histogram Equalization:

```
# Histogram Equalization
im_eq = exposure.equalize_hist(im)
```

- Cân bằng histogram bằng cách phân phối lại giá trị pixel để histogram trải đều hơn.

- Cải thiện độ tương phản cho ảnh.

## 3. Adaptive Histogram Equalization:

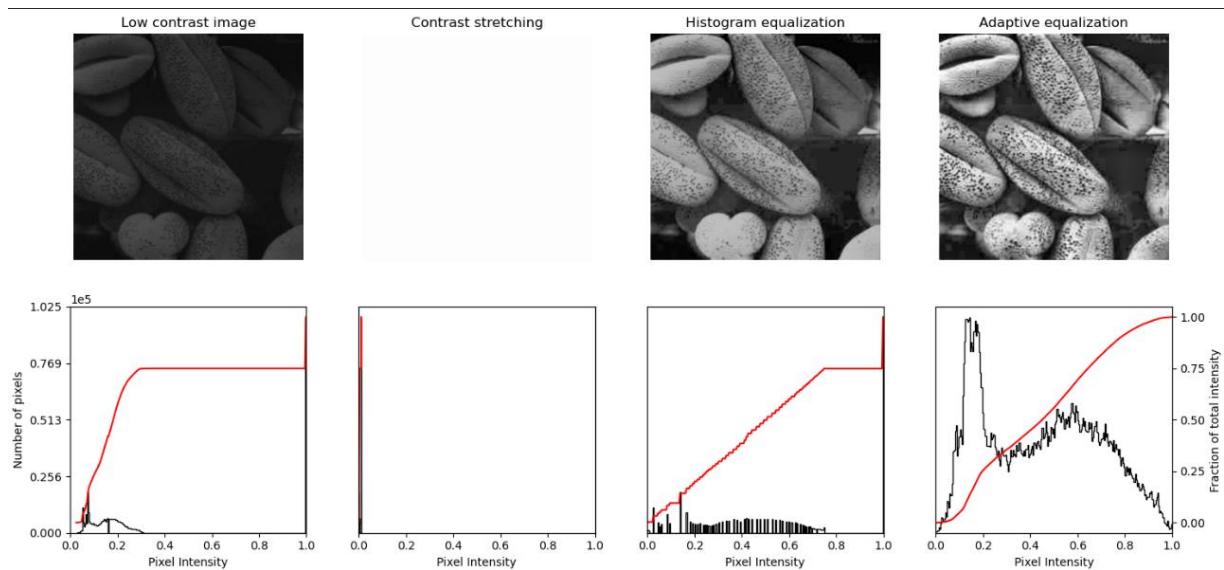
```
# Adaptive Histogram Equalization
im_adapteq = exposure.equalize_adapthist(im, clip_limit=0.03)
```

- Cân bằng histogram cục bộ, tức là áp dụng trên từng vùng nhỏ trong ảnh thay vì toàn bộ ảnh.
- clip\_limit: Giới hạn cường độ để giảm nhiễu.

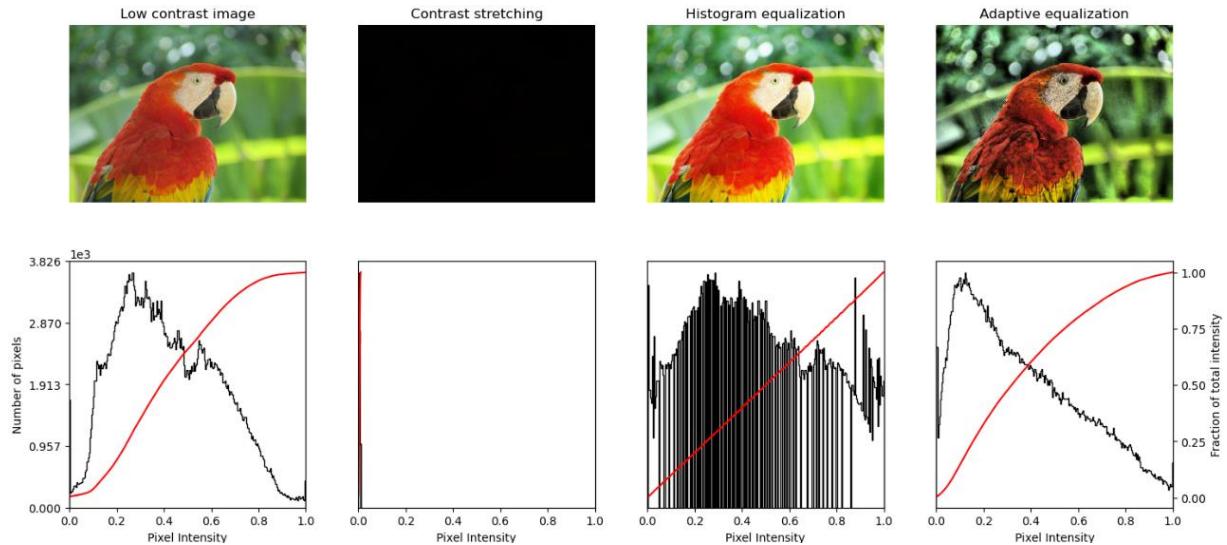
Hiển thị:

- Ảnh gốc và các ảnh sau khi xử lý (Contrast Stretching, Histogram Equalization, Adaptive Histogram Equalization).
- Histogram và CDF tương ứng của từng ảnh, giúp bạn phân tích sự thay đổi phân bố cường độ pixel.

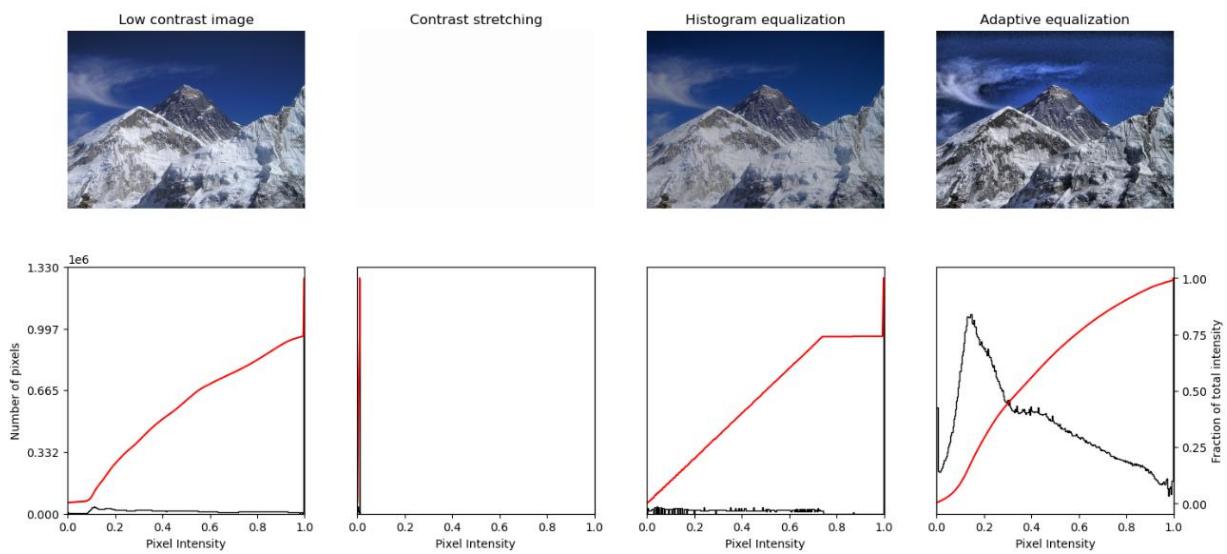
Testcase 1:



### TestCase 2:



### TestCase 3:



## Chương 5 Image Enhancement Using Derivatives

### 1. Unsharp masking with the SciPy ndimage module

Thuật toán Unsharp Masking là một kỹ thuật làm sắc nét ảnh bằng cách sử dụng ảnh gốc và một ảnh làm mờ của nó. Phương pháp này dựa trên ý tưởng rằng sự khác biệt giữa ảnh gốc và ảnh làm mờ sẽ biểu thị các chi tiết của ảnh. Các bước chính bao gồm:

**1. Tạo ảnh làm mờ (Blurred Image):**

- Làm mờ ảnh gốc bằng bộ lọc Gaussian (hoặc phương pháp tương tự). Điều này tạo ra một phiên bản "làm mịn" của ảnh gốc, loại bỏ các chi tiết cường độ cao.

**2. Tạo ảnh chi tiết (Detail Image):**

- Tính toán ảnh chi tiết bằng cách lấy hiệu giữa ảnh gốc và ảnh làm mờ:

$$\text{Detail Image} = \text{Original Image} - \text{Blurred Image}$$

**3. Tạo ảnh sắc nét (Sharpened Image):**

- Kết hợp ảnh gốc và ảnh chi tiết với một trọng số (tham số điều chỉnh, thường gọi là  $\alpha$  hoặc "amount"):

$$\text{Sharpened Image} = \text{Original Image} + \alpha \cdot (\text{Detail Image})$$

- Tham số  $\alpha$  kiểm soát mức độ làm sắc nét. Khi  $\alpha$  lớn, ảnh sẽ trở nên sắc nét hơn, nhưng có thể xuất hiện nhiễu.

```

import numpy as np
from scipy import ndimage
from skimage import img_as_float, io
import matplotlib.pyplot as pylab

def rgb2gray(im):
    return np.clip(0.2989 * im[..., 0] + 0.5870 * im[..., 1] + 0.1140 * im[..., 2], 0, 1)

# Đọc và chuyển ảnh thành dạng grayscale
image_path = 'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\me4.jpg'
im = rgb2gray(img_as_float(io.imread(image_path)))

# Tạo ảnh làm mờ và ảnh chi tiết
sigma = 5
im_blurred = ndimage.gaussian_filter(im, sigma)
im_detail = np.clip(im - im_blurred, 0, 1)

# Cấu hình hiển thị
pylab.gray()
fig, axes = pylab.subplots(nrows=2, ncols=3, sharex=True, sharey=True, figsize=(15, 15))
axes = axes.ravel()

# Hiển thị ảnh gốc, ảnh làm mờ và ảnh chi tiết
axes[0].set_title('Original image', size=15)
axes[0].imshow(im)

axes[1].set_title(f'Blurred image, sigma={sigma}', size=15)
axes[1].imshow(im_blurred)

axes[2].set_title('Detail image', size=15)
axes[2].imshow(im_detail)

# Tạo ảnh sắc nét với các giá trị alpha khác nhau
alpha_values = [1, 5, 10]
for i, alpha in enumerate(alpha_values):
    im_sharp = np.clip(im + alpha * im_detail, 0, 1)
    axes[3 + i].imshow(im_sharp)
    axes[3 + i].set_title(f'Sharpened image, alpha={alpha}', size=15)

#Ẩn các trục và sắp xếp giao diện
for ax in axes:
    ax.axis('off')

fig.tight_layout()
pylab.show()

```

## Chuyển đổi ảnh sang dạng grayscale (RGB → Grayscale)

- Mục tiêu:** Chuyển đổi ảnh màu (RGB) thành ảnh xám để xử lý đơn giản hơn.
- Công thức chuyển đổi:**

$$Grayscale = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

Các trọng số trên được chọn dựa trên độ nhạy của mắt người đối với các màu sắc khác nhau (mắt nhạy nhất với màu xanh lá, sau đó đến đỏ và xanh dương).

```

def rgb2gray(im):
    return np.clip(0.2989 * im[..., 0] + 0.5870 * im[..., 1] + 0.1140 * im[..., 2], 0, 1)

```

## Làm mờ ảnh gốc bằng Gaussian Filter

- **Mục tiêu:** Tạo một phiên bản "làm mờ" của ảnh gốc để loại bỏ các chi tiết cường độ cao, chỉ giữ lại các thành phần lớn hoặc xu hướng chung.
- **Gaussian Filter:** Là bộ lọc làm mờ sử dụng phân phối chuẩn để giảm cường độ của các điểm ảnh. Độ rộng của bộ lọc được xác định bởi tham số  $\sigma$ (sigma), là độ lệch chuẩn:
  - $\sigma$  càng lớn thì ảnh càng bị làm mờ mạnh.

```
# Tạo ảnh làm mờ và ảnh chi tiết
sigma = 5
im_blurred = ndimage.gaussian_filter(im, sigma)
im_detail = np.clip(im - im_blurred, 0, 1)
```

## Tính toán ảnh chi tiết (Detail Image)

- **Mục tiêu:** Xác định các chi tiết bị mất trong quá trình làm mờ bằng cách lấy hiệu giữa ảnh gốc và ảnh làm mờ.
- **Công thức:**  $Detail\ Image = Original\ Image - Blurred\ Image$
- **Ý nghĩa:** Ảnh chi tiết này chứa thông tin tần số cao (ví dụ: cạnh, biên của các đối tượng) mà bộ lọc Gaussian đã loại bỏ.

```
im_detail = np.clip(im - im_blurred, 0, 1)
```

## Làm sắc nét ảnh bằng cách tăng cường chi tiết

- **Mục tiêu:** Thêm các chi tiết từ "Detail Image" trở lại ảnh gốc để tạo hiệu ứng làm sắc nét.
- **Công thức:**

$$Sharpened\ Image = Original\ Image + \alpha.(Detail\ Image)$$

- $\alpha$ : Hệ số điều chỉnh mức độ sắc nét.
- Giá trị  $\alpha$  càng lớn thì chi tiết được tăng cường càng mạnh, nhưng có thể làm xuất hiện nhiễu hoặc hiệu ứng không tự nhiên.

```
# Tạo ảnh sắc nét với các giá trị alpha khác nhau
alpha_values = [1, 5, 10]
for i, alpha in enumerate(alpha_values):
    im_sharp = np.clip(im + alpha * im_detail, 0, 1)
    axes[3 + i].imshow(im_sharp)
    axes[3 + i].set_title(f'Sharpened image, alpha={alpha}', size=15)
```

## Hiệu quả của thuật toán

### 1. Hiệu ứng làm sắc nét:

- Làm nổi bật các cạnh và biên của đối tượng trong ảnh.
- Làm rõ các chi tiết nhỏ, giúp ảnh trở nên sắc nét và dễ quan sát hơn.

### 2. Điều chỉnh mức độ sắc nét bằng $\alpha$ :

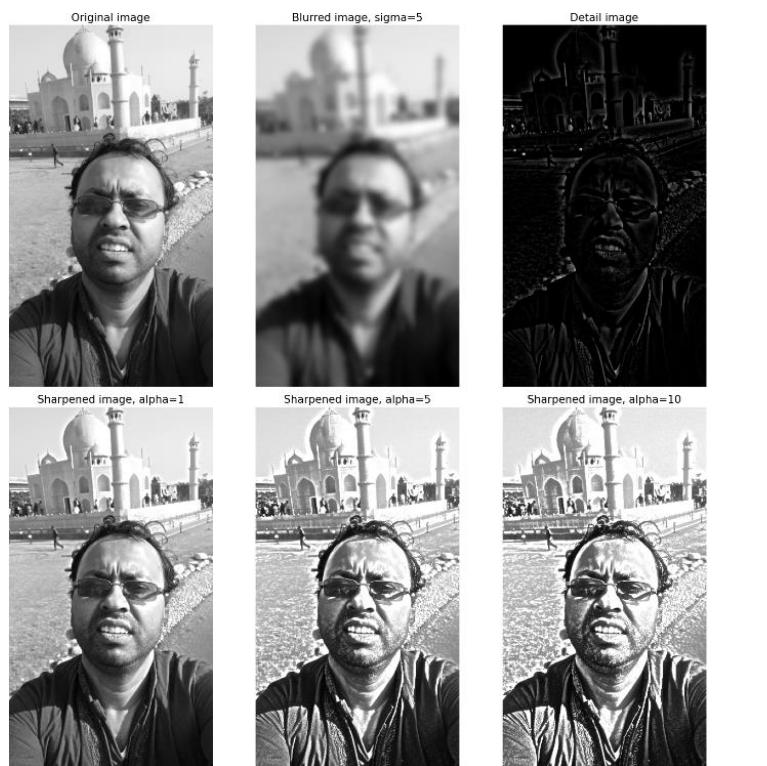
- $\alpha=1$ : Làm sắc nét nhẹ, tăng cường chi tiết một cách tự nhiên.
- $\alpha=5$ : Làm sắc nét mạnh, các cạnh trở nên nổi bật hơn.
- $\alpha=10$ : Quá sắc nét, có thể xuất hiện nhiều hoặc hiệu ứng "halo" không mong muốn.

### 3. Tùy chỉnh $\sigma$ :

- Nếu  $\sigma$  quá nhỏ, ảnh làm mờ sẽ không đủ mịn và các chi tiết nhỏ không được tách biệt.
- Nếu  $\sigma$  quá lớn, ảnh làm mờ sẽ làm mất đi nhiều thông tin của ảnh gốc.

Hiển thị kết quả:

TestCase 1:



## TestCase 2:



### TestCase 3:



## Chương 6 Morphological Image Processing

### 1. The scikit-image morphology module

#### Binary operations

### 2. Erosion (Xói mòn)

- **Erosion là gì?**
  - Xói mòn làm **thu nhỏ** các vùng foreground (các điểm sáng, thường có giá trị 1 trong ảnh nhị phân).
  - Loại bỏ các chi tiết nhỏ hoặc nhiễu (noise), làm **mượt các biên dạng** của đối tượng.
  - Khi áp dụng, các vùng foreground chỉ được giữ lại nếu chúng hoàn toàn nằm trong phần tử cấu trúc (**structuring element**).
- **Ứng dụng:**
  - Loại bỏ nhiễu.
  - Làm mịn đường biên.
  - Phát hiện các thành phần hình dạng nhỏ.
- **Hoạt động:**
  - Một **structuring element** (**phần tử cấu trúc**), ví dụ như hình chữ nhật hoặc hình tròn, được quét qua ảnh.

- Với mỗi vị trí, nếu phần tử cấu trúc không hoàn toàn nằm trong vùng foreground, điểm ảnh tại vị trí đó sẽ bị "xói mòn" thành background (giá trị 0).

```
# Hàm hỗ trợ để hiển thị ảnh với tiêu đề
def plot_image(image, title=''):
    pylab.title(title, size=20)
    pylab.imshow(image)
    pylab.axis('off') # Ẩn các trục của biểu đồ (có thể bỏ dòng này nếu muốn hiển thị trực)

# Đọc ảnh và chuyển sang grayscale
image_path = 'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\clock2.jpg'
im = rgb2gray(imread(image_path))

# Chuyển đổi ảnh grayscale sang ảnh nhị phân dựa trên ngưỡng cố định 0.5
im[im <= 0.5] = 0 # Các giá trị <= 0.5 chuyển thành 0 (background)
im[im > 0.5] = 1 # Các giá trị > 0.5 chuyển thành 1 (foreground)

# Hiển thị ảnh gốc và các kết quả xói mòn
pylab.gray() # Hiển thị ảnh ở chế độ grayscale
pylab.figure(figsize=(20, 10))

# Hiển thị ảnh gốc
pylab.subplot(1, 3, 1)
plot_image(im, 'Original Image')

# Thực hiện phép xói mòn với hình chữ nhật (1, 5)
im_eroded_1 = binary_erosion(im, rectangle(1, 5))
pylab.subplot(1, 3, 2)
plot_image(im_eroded_1, 'Erosion with Rectangle (1, 5)')

# Thực hiện phép xói mòn với hình chữ nhật (1, 15)
im_eroded_2 = binary_erosion(im, rectangle(1, 15))
pylab.subplot(1, 3, 3)
plot_image(im_eroded_2, 'Erosion with Rectangle (1, 15)')

# Hiển thị kết quả
pylab.show()
```

## Đọc và chuyển ảnh sang grayscale

- **Đọc ảnh:** Đoạn mã bắt đầu bằng cách đọc ảnh từ một tệp tin, sau đó chuyển đổi ảnh này sang **grayscale** (ảnh xám) sử dụng hàm `rgb2gray()` từ thư viện **scikit-image**. Điều này giúp giảm bớt phức tạp và chỉ làm việc với một kênh màu thay vì ba kênh màu (RGB).

```
im = rgb2gray(imread(image_path))
```

**Chuyển đổi sang ảnh nhị phân:** Sau khi ảnh được chuyển sang grayscale (với giá trị pixel trong khoảng từ 0 đến 1), ta sử dụng một ngưỡng cố định (0.5) để chuyển ảnh grayscale thành ảnh **nhiệt phân** (binary image).

- Các pixel có giá trị  $\leq 0.5$  sẽ được gán thành **0** (background).
- Các pixel có giá trị  $> 0.5$  sẽ được gán thành **1** (foreground).

```
# Chuyển đổi ảnh grayscale sang ảnh nhị phân dựa trên ngưỡng cố định 0.5
im[im <= 0.5] = 0 # Các giá trị <= 0.5 chuyển thành 0 (background)
im[im > 0.5] = 1 # Các giá trị > 0.5 chuyển thành 1 (foreground)
```

## Áp dụng phép xói mòn (Erosion)

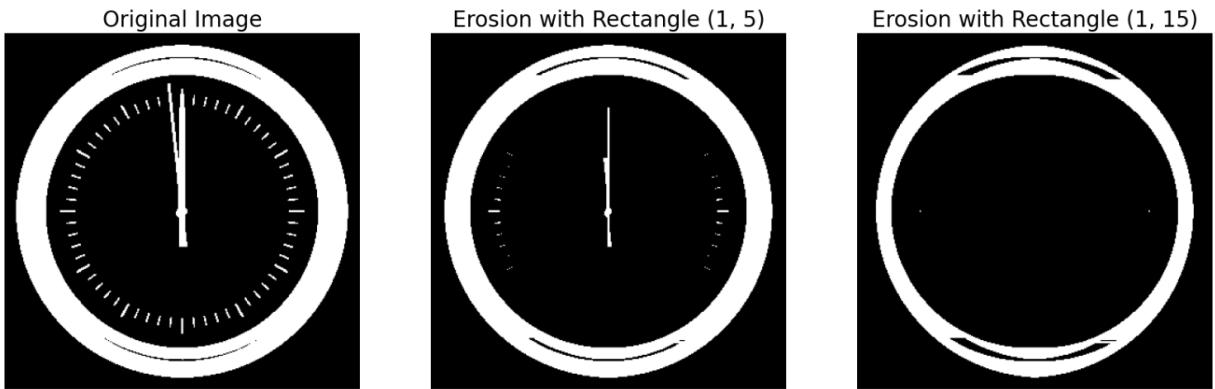
- **Phép xói mòn:** Phép xói mòn được thực hiện bằng cách sử dụng hàm `binary_erosion()` từ **scikit-image**, với một **structuring element** là hình chữ nhật. Đây là bước quan trọng trong xử lý hình thái học, giúp loại bỏ các chi tiết nhỏ hoặc làm mịn các biên dạng của các đối tượng trong ảnh.
- **Structuring element:** Trong ví dụ này, hình chữ nhật có kích thước **(1, 5)** và **(1, 15)** được sử dụng làm phần tử cấu trúc.
  - `rectangle(1, 5)` tạo ra một phần tử cấu trúc có chiều cao là 1 và chiều rộng là 5, giúp loại bỏ các chi tiết nhỏ, ví dụ như các dấu vết nhỏ trên ảnh đồng hồ.
  - `rectangle(1, 15)` tạo ra một phần tử cấu trúc có chiều rộng lớn hơn (15), giúp làm xói mòn mạnh mẽ hơn và loại bỏ các đối tượng lớn hơn, chẳng hạn như kim đồng hồ.

```
# Thực hiện phép xói mòn với hình chữ nhật (1, 5)
im_eroded_1 = binary_erosion(im, rectangle(1, 5))
pylab.subplot(1, 3, 2)
plot_image(im_eroded_1, 'Erosion with Rectangle (1, 5)')

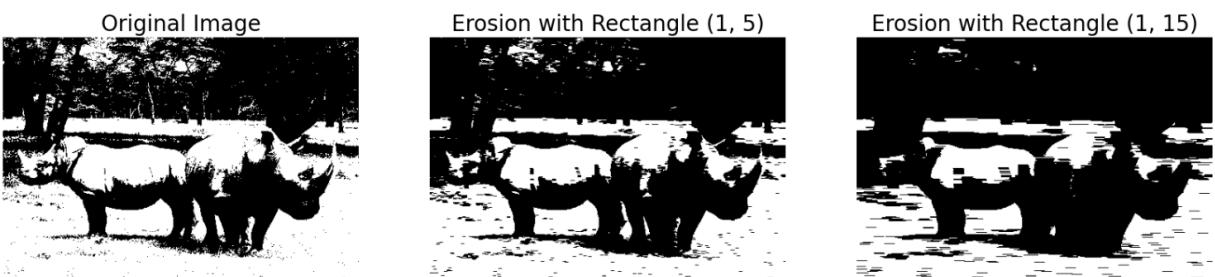
# Thực hiện phép xói mòn với hình chữ nhật (1, 15)
im_eroded_2 = binary_erosion(im, rectangle(1, 15))
pylab.subplot(1, 3, 3)
plot_image(im_eroded_2, 'Erosion with Rectangle (1, 15)')
```

Hiển thị kết quả

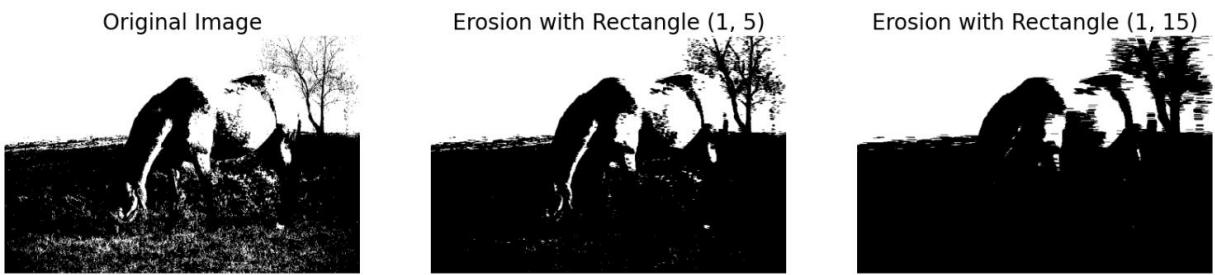
TestCase 1:



TestCase 2:



TestCase 3:



## 2. Skeletonizing

**Skeletonization** là một phép toán trong xử lý ảnh nhằm giảm một đối tượng kết nối trong ảnh nhị phân xuống thành một "xương sống" (skeleton), thường là một đường mảnh, có độ rộng chỉ bằng một pixel. Phép toán này được thực hiện bằng cách sử dụng phương pháp mỏng đi (thinning) trong hình thái học. Mục tiêu là giữ lại các cấu trúc hình học quan trọng trong ảnh trong khi loại bỏ các chi tiết không cần thiết.

**Đọc ảnh:** Đầu tiên, ảnh được đọc vào từ tệp (ví dụ: `dinosaur.png` trong đoạn mã). Ảnh có thể có nhiều kênh màu (RGB), nhưng ta chỉ sử dụng kênh alpha (mã số 3, trong trường hợp này là `[..., 3]`), vì đây thường là kênh trong suốt hoặc một lớp đặc biệt cho ảnh.

**Chuyển ảnh sang nhị phân:** Sau khi đọc ảnh, các giá trị pixel trong ảnh được chuyển sang nhị phân bằng cách sử dụng ngưỡng xác định (0.5 trong ví dụ này):

- Các pixel có giá trị lớn hơn ngưỡng sẽ được gán là 1 (foreground).

- Các pixel có giá trị nhỏ hơn hoặc bằng ngưỡng sẽ được gán là **0** (background).

**Thực hiện Skeletonization:** Sau khi ảnh đã được chuyển sang dạng nhị phân, ta áp dụng phép toán **skeletonize** từ thư viện **scikit-image**. Phép toán này sẽ biến đổi hình dạng của các đối tượng trong ảnh thành một đường mảnh, giữ lại các đặc điểm chính của các đối tượng ban đầu nhưng giảm chúng xuống còn một pixel rộng.

**Hiển thị ảnh:** Cuối cùng, ảnh gốc và ảnh đã skeletonize được hiển thị để so sánh. Hàm **plot\_images\_horizontally** sẽ hiển thị cả hai ảnh cạnh nhau để dễ dàng so sánh.

```
# Hàm để hiển thị ảnh gốc và ảnh skeleton cạnh nhau
def plot_images_horizontally(original, filtered, filter_name, sz=(18,7)):
    pylab.gray() # Thiết lập chế độ màu xám cho ảnh
    pylab.figure(figsize = sz) # Thiết lập kích thước ảnh
    pylab.subplot(1, 2, 1) # Tạo subplot đầu tiên để hiển thị ảnh gốc
    plot_image(original, 'Original') # Hiển thị ảnh gốc
    pylab.subplot(1, 2, 2) # Tạo subplot thứ hai để hiển thị ảnh skeleton
    plot_image(filtered, filter_name) # Hiển thị ảnh đã skeletonize
    pylab.show() # Hiển thị ảnh

# Đọc ảnh và chuyển đổi ảnh thành dạng nhị phân
im = img_as_float(imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\dynasaur.png')[..., 3]) # Đọc ảnh và lấy kênh alpha (4 kênh)
threshold = 0.5 # Đặt ngưỡng để chuyển ảnh sang nhị phân
im[im < threshold] = 0 # Các pixel nhỏ hơn hoặc bằng ngưỡng được gán thành 0
im[im > threshold] = 1 # Các pixel lớn hơn ngưỡng được gán thành 1

# Áp dụng thuật toán skeletonize
skeleton = skeletonize(im)

# Hiển thị ảnh gốc và ảnh skeleton cạnh nhau
plot_images_horizontally(im, skeleton, 'Skeleton', sz=(18, 9))
```

### Hàm **plot\_images\_horizontally**:

Hàm này hiển thị hai ảnh: ảnh gốc và ảnh sau khi áp dụng thuật toán skeletonize cạnh nhau để người dùng có thể so sánh kết quả.

#### Chi tiết:

- **pylab.gray()**: Thiết lập chế độ màu xám cho ảnh.
- **pylab.figure(figsize=sz)**: Thiết lập kích thước của cửa sổ hiển thị ảnh.
- **pylab.subplot(1, 2, 1)** và **pylab.subplot(1, 2, 2)**: Tạo hai ô (subplot) trong một cửa sổ để hiển thị ảnh gốc và ảnh skeletonized.
- **plot\_image(original, 'Original')** và **plot\_image(filtered, filter\_name)**: Gọi hàm **plot\_image** để vẽ ảnh gốc và ảnh đã qua skeletonization.
- **pylab.show()**: Hiển thị cửa sổ ảnh.

```

def plot_images_horizontally(original, filtered, filter_name, sz=(18,7)):
    pylab.gray() # Thiết lập chế độ màu xám cho ảnh
    pylab.figure(figsize = sz) # Thiết lập kích thước ảnh
    pylab.subplot(1, 2, 1) # Tạo subplot đầu tiên để hiển thị ảnh gốc
    plot_image(original, 'Original') # Hiển thị ảnh gốc
    pylab.subplot(1, 2, 2) # Tạo subplot thứ hai để hiển thị ảnh skeleton
    plot_image(filtered, filter_name) # Hiển thị ảnh đã skeletonize
    pylab.show() # Hiển thị ảnh

```

### Đọc và chuyển ảnh thành nhị phân:

Đoạn mã này xử lý ảnh đầu vào và chuyển nó thành ảnh nhị phân (binary image) bằng cách áp dụng một ngưỡng (threshold). Quá trình này bao gồm:

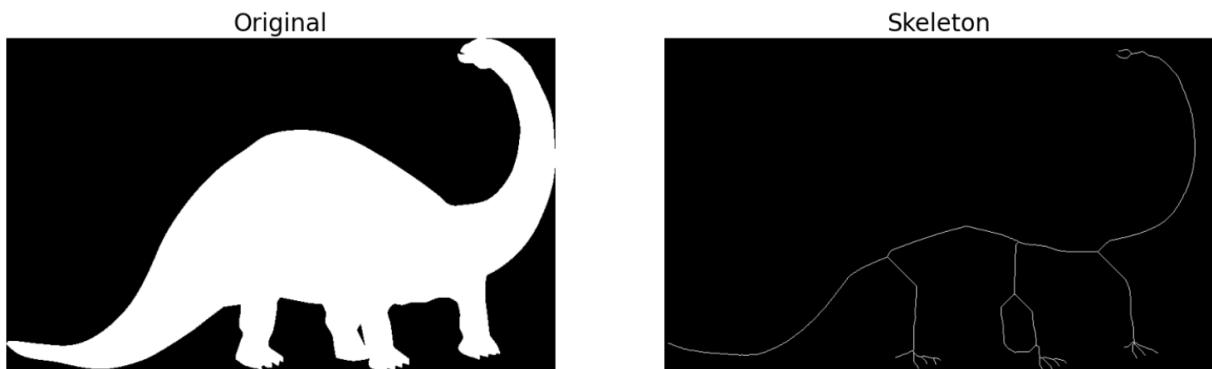
- **im = img\_as\_float(imread('D:\\...\\dynasaur.png')[..., 3]):** Đọc ảnh từ đường dẫn cụ thể và lấy kênh alpha (kênh trong suốt). Mỗi pixel trong ảnh này được chuyển đổi thành giá trị thực.
- **threshold = 0.5:** Đặt ngưỡng giá trị để phân loại các pixel.
- **im[im <= threshold] = 0:** Các pixel có giá trị nhỏ hơn hoặc bằng ngưỡng sẽ được gán thành 0 (background).
- **im[im > threshold] = 1:** Các pixel có giá trị lớn hơn ngưỡng sẽ được gán thành 1 (foreground).

### 3. Áp dụng thuật toán Skeletonization:

- **skeleton = skeletonize(im):** Dùng hàm skeletonize từ thư viện skimage.morphology để áp dụng thuật toán skeletonization lên ảnh nhị phân. Kết quả là một ảnh mà các đối tượng trong ảnh được rút gọn xuống còn các đường mảnh, giúp giữ lại các đặc điểm chính nhưng loại bỏ các chi tiết nhỏ không cần thiết.

Hiển thị kết quả:

TestCase 1:



TestCase 2:



## Chương 7 Extracting Image Features and Descriptors

### 1. Determinant of Hessian (DoH)

#### Laplacian of Gaussian (LoG)

Laplacian of Gaussian (LoG) là một phương pháp phát hiện blob dựa trên việc kết hợp giữa toán học của phép toán Gaussian (một bộ lọc làm mờ) và Laplacian (một phép toán vi phân bậc hai).

Quy trình:

- Gaussian được sử dụng để làm mờ ảnh, làm mượt các chi tiết nhỏ, giúp tăng cường các cấu trúc lớn trong ảnh.
- Laplacian là phép toán vi phân bậc hai, nó giúp phát hiện các cực trị trong ảnh (vùng sáng hay tối), đặc biệt hữu ích để phát hiện các điểm đặc biệt, ví dụ như các vùng blob.

Quy trình thực hiện:

1. Làm mờ ảnh bằng bộ lọc Gaussian: Bộ lọc Gaussian giúp giảm bớt chi tiết nhỏ trong ảnh.
2. Áp dụng Laplacian: Laplacian sẽ tìm các cực trị trong ảnh đã qua làm mờ Gaussian. Những cực trị này thường xuất hiện ở các khu vực có sự thay đổi lớn về độ sáng (blobs).
3. Phát hiện cực trị: Các điểm mà Laplacian cho ra giá trị cực đại hoặc cực tiểu sẽ được xem là blob.

## Difference of Gaussian (DoG)

Difference of Gaussian (DoG) là một phương pháp thay thế gần đúng cho LoG và sử dụng sự khác biệt giữa hai ảnh Gaussian với độ mờ khác nhau để phát hiện blob.

Quy trình:

1. Làm mờ ảnh bằng bộ lọc Gaussian với hai độ rộng khác nhau: Tạo ra hai ảnh đã được làm mờ với các giá trị sigma (độ rộng của Gaussian) khác nhau.
2. Tính hiệu giữa hai ảnh Gaussian: Phương pháp này lấy hiệu của hai ảnh đã được làm mờ khác nhau (sự khác biệt giữa các giá trị Gaussian với độ mờ khác nhau). Sự khác biệt này giúp làm nổi bật các vùng có sự thay đổi mạnh mẽ về độ sáng, tức là các blob.
3. Phát hiện các điểm cực trị: Sau khi tính hiệu, các điểm cực trị (nơi có sự thay đổi lớn nhất) sẽ được xem là blob.

## Determinant of Hessian (DoH)

Determinant of Hessian (DoH) là một phương pháp phát hiện blob dựa trên ma trận Hessian (ma trận đạo hàm bậc hai). Nó xác định các vùng trong ảnh có sự thay đổi đáng kể về độ sáng, đặc biệt là các vùng có độ cong mạnh, nơi xuất hiện blob.

Quy trình:

1. Hessian matrix: Ma trận Hessian của ảnh được tính bằng các đạo hàm bậc hai theo cả hai chiều không gian (theo x và y). Ma trận Hessian biểu thị sự cong của ảnh tại mỗi điểm.
2. Determinant (định thức) của Hessian: Định thức của ma trận Hessian giúp xác định mức độ thay đổi độ sáng tại mỗi điểm. Nếu định thức lớn, điều đó có nghĩa là có sự thay đổi lớn về độ sáng tại điểm đó, tạo ra các blob.
3. Phát hiện blob: Các điểm có giá trị định thức cao sẽ được xem là blob, đặc biệt là các vùng có đặc trưng về sự thay đổi mạnh mẽ về độ sáng.

```

# Đọc ảnh và loại bỏ kênh alpha (nếu có)
im = imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\butterfly.png')

# Chuyển đổi ảnh RGBA thành RGB bằng cách chỉ lấy 3 kênh đầu tiên
im_rgb = im[:, :, :3]

# Chuyển ảnh RGB thành ảnh grayscale
im_gray = rgb2gray(im_rgb)

# Phát hiện blob bằng phương pháp Laplacian of Gaussian (LoG)
log_blobs = blob_log(im_gray, max_sigma=30, num_sigma=10, threshold=0.1)
log_blobs[:, 2] = sqrt(2) * log_blobs[:, 2] # Tính bán kính ở cột thứ 3

# Phát hiện blob bằng phương pháp Difference of Gaussian (DoG)
dog_blobs = blob_dog(im_gray, max_sigma=30, threshold=0.1)
dog_blobs[:, 2] = sqrt(2) * dog_blobs[:, 2] # Tính bán kính ở cột thứ 3

# Phát hiện blob bằng phương pháp Determinant of Hessian (DoH)
doh_blobs = blob_doh(im_gray, max_sigma=30, threshold=0.005)

# Danh sách các blob và tiêu đề tương ứng
list_blobs = [log_blobs, dog_blobs, doh_blobs]
colors = ['yellow', 'lime', 'red']
titles = ['Laplacian of Gaussian', 'Difference of Gaussian', 'Determinant of Hessian']

# Vẽ các kết quả
fig, axes = plt.subplots(2, 2, figsize=(20, 20), sharex=True, sharey=True)
axes = axes.ravel()
axes[0].imshow(im, interpolation='nearest')
axes[0].set_title('Original Image', size=30)
axes[0].set_axis_off()

# Duyệt qua các phương pháp và vẽ kết quả
for idx, (blobs, color, title) in enumerate(zip(list_blobs, colors, titles)):
    axes[idx + 1].imshow(im, interpolation='nearest')
    axes[idx + 1].set_title(f'Blobs with {title}', size=30)
    for blob in blobs:
        y, x, radius = blob
        col = plt.Circle((x, y), radius, color=color, linewidth=2, fill=False)
        axes[idx + 1].add_patch(col)
    axes[idx + 1].set_axis_off()

# Hiển thị các kết quả
plt.tight_layout()
plt.show()

```

Phát hiện blob sử dụng Laplacian of Gaussian (LoG)

```

# Phát hiện blob bằng phương pháp Laplacian of Gaussian (LoG)
log_blobs = blob_log(im_gray, max_sigma=30, num_sigma=10, threshold=0.1)
log_blobs[:, 2] = sqrt(2) * log_blobs[:, 2] # Tính bán kính ở cột thứ 3

```

**blob\_log:** Phát hiện blob bằng phương pháp **Laplacian of Gaussian (LoG)**. Các tham số:

- **max\_sigma=30:** Giá trị tối đa của sigma, xác định kích thước của các blob được phát hiện.
- **num\_sigma=10:** Số lượng giá trị sigma khác nhau để áp dụng bộ lọc Gaussian.

- **threshold=0.1**: Nguồn độ sáng để phân biệt blob. Các điểm có giá trị dưới nguồn này sẽ không được xem là blob.

Sau khi phát hiện blob, chúng ta cần tính lại bán kính của blob trong cột thứ ba (*log\_blobs[:, 2]*). Công thức  $\sqrt{2} * \text{log\_blobs}[:, 2]$  được sử dụng để điều chỉnh lại bán kính theo công thức chuẩn của LoG.

Phát hiện blob sử dụng Difference of Gaussian (DoG)

```
# Phát hiện blob bằng phương pháp Difference of Gaussian (DoG)
dog_blobs = blob_dog(im_gray, max_sigma=30, threshold=0.1)
dog_blobs[:, 2] = sqrt(2) * dog_blobs[:, 2] # Tính bán kính ở cột thứ 3
```

**blob\_dog**: Phát hiện blob bằng phương pháp **Difference of Gaussian** (DoG). Các tham số:

- **max\_sigma=30**: Giá trị tối đa của sigma cho phép.
- **threshold=0.1**: Nguồn phát hiện blob.

Sau khi phát hiện blob, bán kính blob trong cột thứ ba của *dog\_blobs* cũng được điều chỉnh bằng công thức  $\sqrt{2} * \text{dog\_blobs}[:, 2]$ .

Phát hiện blob sử dụng Determinant of Hessian (DoH)

```
# Phát hiện blob bằng phương pháp Determinant of Hessian (DoH)
doh_blobs = blob_doh(im_gray, max_sigma=30, threshold=0.005)
```

**blob\_doh**: Phát hiện blob bằng phương pháp **Determinant of Hessian** (DoH). Các tham số:

- **max\_sigma=30**: Giá trị tối đa của sigma.
- **threshold=0.005**: Nguồn phát hiện blob, chỉ nhận các blob có giá trị định thức Hessian lớn hơn nguồn này.

Hiển thị các blob phát hiện từ ba phương pháp

```
# Danh sách các blob và tiêu đề tương ứng
list_blobs = [log_blobs, dog_blobs, doh_blobs]
colors = ['yellow', 'lime', 'red']
titles = ['Laplacian of Gaussian', 'Difference of Gaussian', 'Determinant of Hessian']
```

**list\_blobs**: Lưu trữ các blob đã phát hiện từ ba phương pháp: LoG, DoG, và DoH.

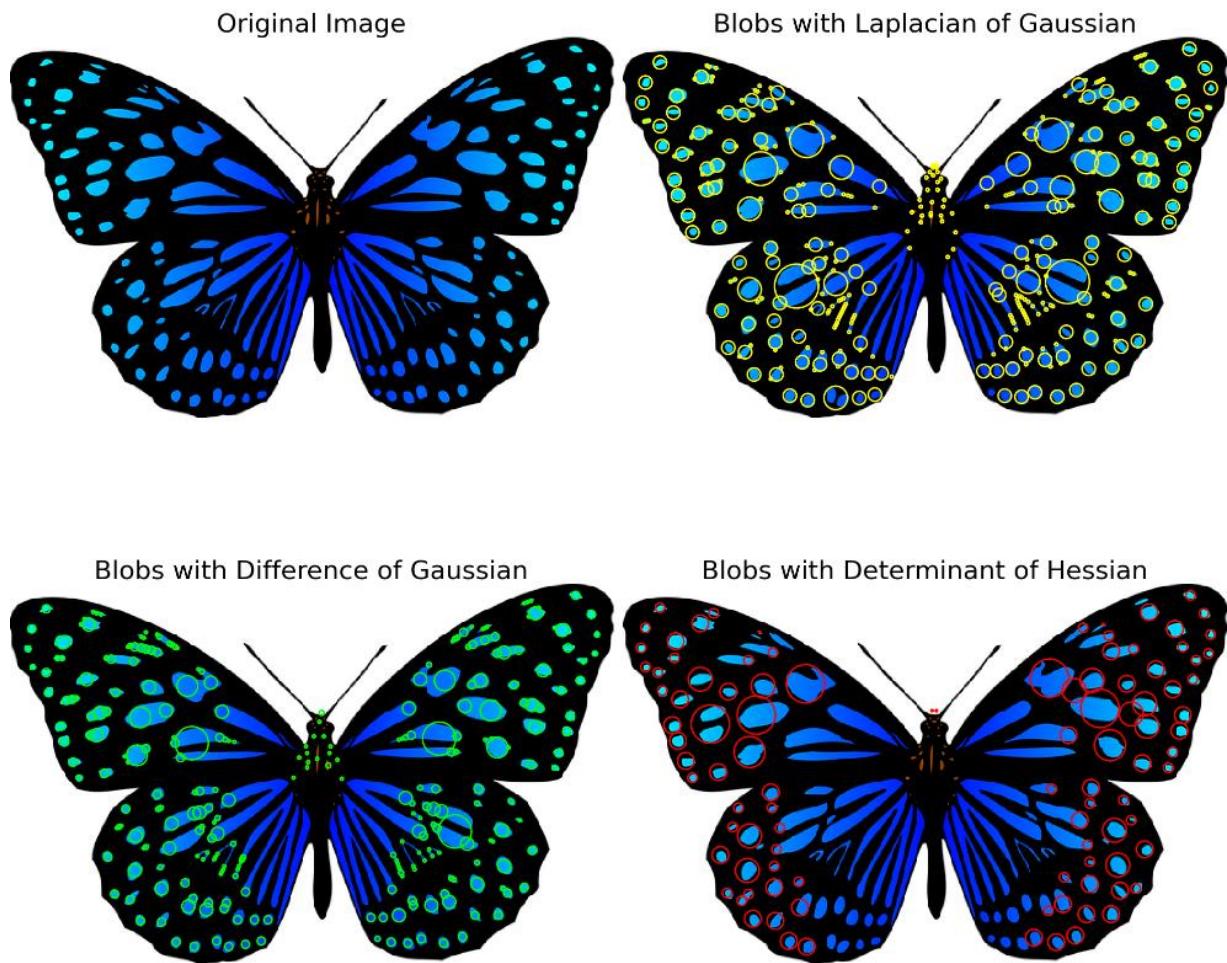
**color**: Mảng màu dùng để vẽ các blob trên ảnh (mỗi phương pháp sẽ có một màu riêng biệt).

**titles**: Các tiêu đề tương ứng cho mỗi phương pháp để hiển thị trên đồ thị.

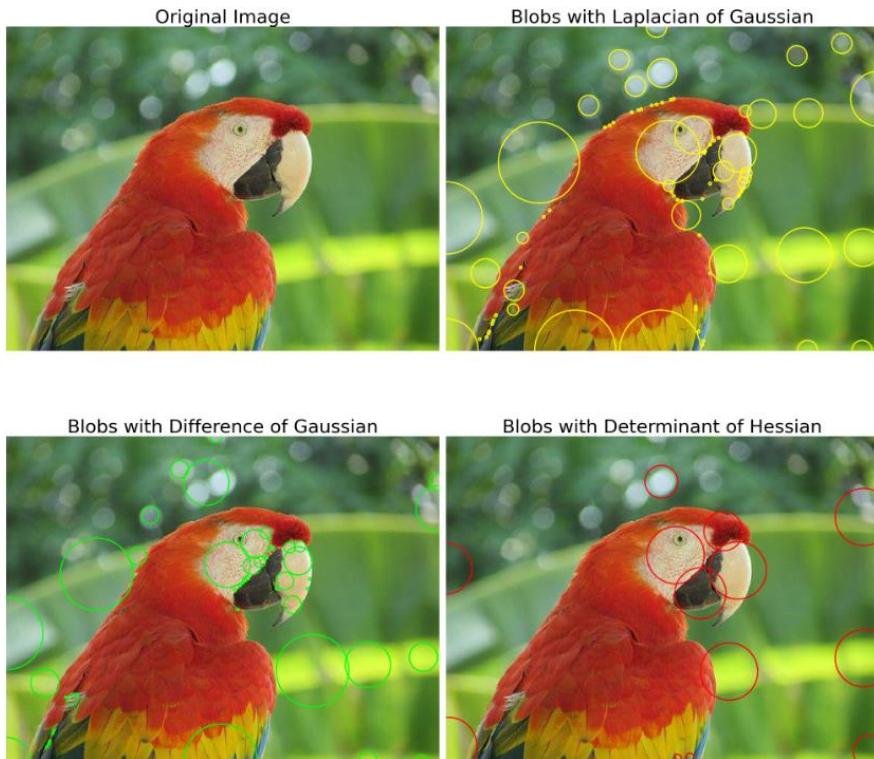
**zip**: Kết hợp ba mảng (list\_blobs, color, titles) thành một đối tượng tuần tự để dễ dàng lặp qua khi vẽ.

Hiển thị kết quả:

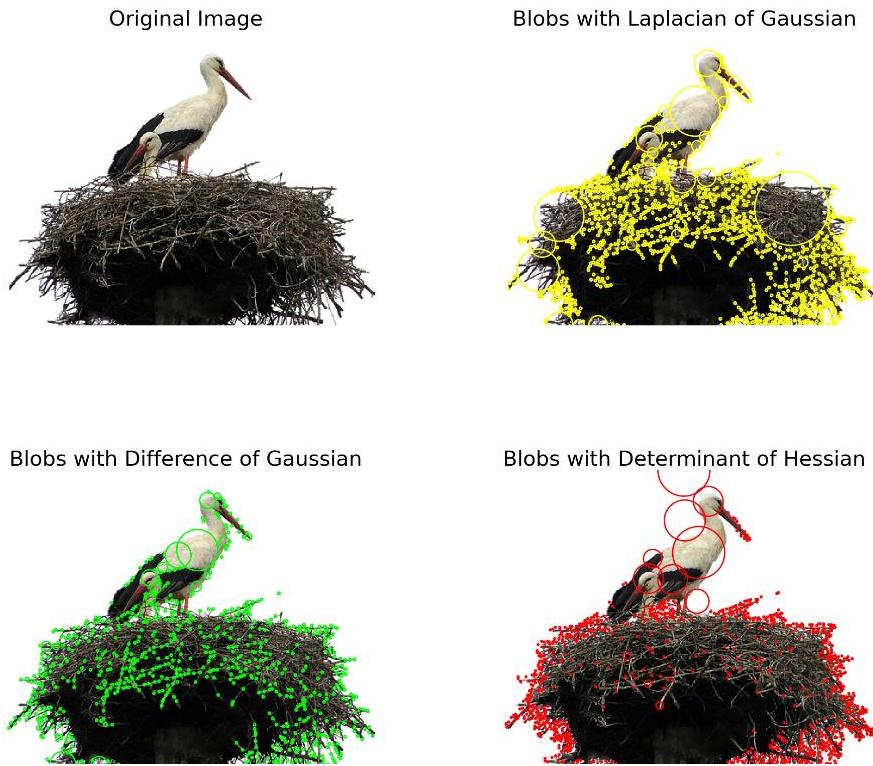
TestCase 1:



### TestCase 2:



### TestCase 3:



## 2. Compute HOG descriptors with scikit-image

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import hog
from skimage import exposure
from skimage.io import imread
from skimage.color import rgb2gray

# Đọc ảnh và chuyển thành ảnh grayscale (loại bỏ kênh alpha nếu có)
image = imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\butterfly.png') # Đảm bảo đường dẫn ảnh đúng
if image.shape[2] == 4: # Nếu ảnh có 4 kênh (RGBA)
    image = image[:, :, :3] # Chỉ lấy 3 kênh đầu tiên (RGB)

image_gray = rgb2gray(image)

# Tính toán HOG descriptors
fd, hog_image = hog(image_gray, orientations=8, pixels_per_cell=(16, 16),
                     cells_per_block=(1, 1), visualize=True)

# In kích thước của ảnh và độ dài của vector đặc trưng HOG
print("Image shape:", image_gray.shape)
print("Length of HOG descriptor:", len(fd))

# Vẽ ảnh gốc và ảnh HOG
fig, (axes1, axes2) = plt.subplots(1, 2, figsize=(15, 10), sharex=True, sharey=True)

# Vẽ ảnh gốc
axes1.axis('off')
axes1.imshow(image_gray, cmap=plt.cm.gray)
axes1.set_title('Input image')

# Rescale ảnh HOG để hiển thị dễ dàng hơn
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# Vẽ ảnh HOG
axes2.axis('off')
axes2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
axes2.set_title('Histogram of Oriented Gradients')

# Hiển thị
plt.show()
```

Tính toán HOG descriptors

```
# Tính toán HOG descriptors
fd, hog_image = hog(image_gray, orientations=8, pixels_per_cell=(16, 16),
                     cells_per_block=(1, 1), visualize=True)
```

- **hog(image):** Đây là hàm tính toán **HOG descriptors**. Hàm này trả về:
  - **fd** (feature descriptor): Đây là vector đặc trưng HOG, chứa thông tin về gradient hướng của ảnh. Đặc trưng này được sử dụng trong các tác vụ như nhận dạng đối tượng và phân loại.
  - **hog\_image:** Đây là ảnh **HOG visualization**, thể hiện hình ảnh gradient của các hướng trong ảnh.

Các tham số của hàm hog():

- **orientations=8:** Số lượng hướng của gradient mà thuật toán sẽ chia. Mỗi pixel sẽ được gán vào một trong 8 hướng, thường là chia đều từ 0 đến 180 độ.

- **pixels\_per\_cell=(16, 16)**: Kích thước của mỗi cell (ô vuông nhỏ), mỗi cell chứa 16x16 pixel. HOG sẽ tính toán hướng gradient trong mỗi cell này.
- **cells\_per\_block=(1, 1)**: Số lượng cell trong một block. Block là tập hợp các cell mà thuật toán sẽ tính toán đặc trưng gradient tổng hợp. Mỗi block ở đây chứa 1x1 cell.
- **visualize=True**: Trả về ảnh hiển thị HOG giúp trực quan hóa gradient hướng trên ảnh.

Vẽ ảnh gốc và HOG descriptor:

```
# Vẽ ảnh gốc và ảnh HOG
fig, (axes1, axes2) = plt.subplots(1, 2, figsize=(15, 10), sharex=True, sharey=True)

# Vẽ ảnh gốc
axes1.axis('off')
axes1.imshow(image_gray, cmap=plt.cm.gray)
axes1.set_title('Input image')
```

Chuyển đổi và vẽ ảnh HOG:

```
# Rescale ảnh HOG để hiển thị dễ dàng hơn
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# Vẽ ảnh HOG
axes2.axis('off')
axes2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
axes2.set_title('Histogram of Oriented Gradients')
```

**exposure.rescale\_intensity(hog\_image, in\_range=(0, 10))**: Hàm này điều chỉnh cường độ của ảnh HOG để dễ dàng hiển thị hơn. Các giá trị trong ảnh HOG có thể rất lớn hoặc rất nhỏ, do đó việc rescale lại giúp tăng độ tương phản và dễ quan sát các đặc trưng hơn.

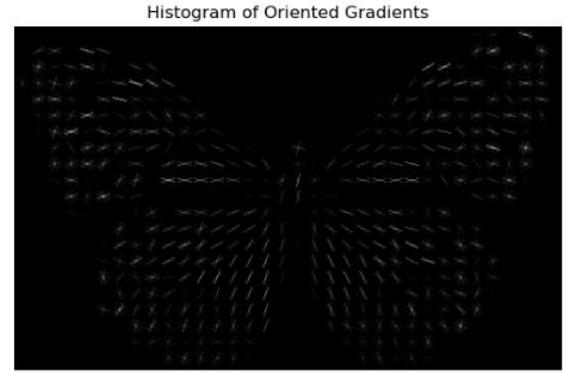
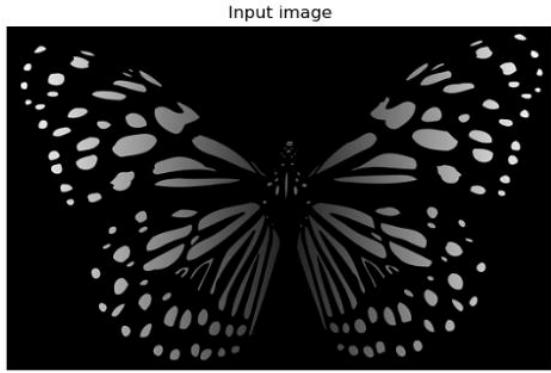
- **in\_range=(0, 10)**: Thiết lập giá trị cường độ ảnh HOG từ 0 đến 10.

**axes2.imshow(hog\_image\_rescaled, cmap=pylab.cm.gray)**: Vẽ ảnh HOG đã được điều chỉnh cường độ lên subplot thứ hai (axes2).

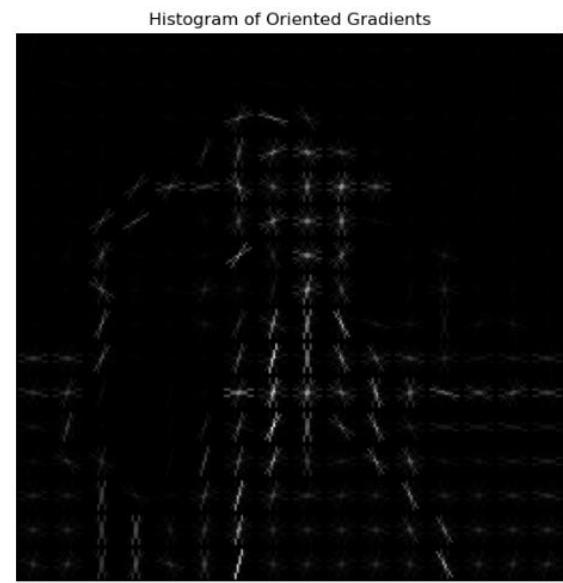
**axes2.set\_title('Histogram of Oriented Gradients')**: Đặt tiêu đề cho ảnh HOG.

Hiển thị kết quả:

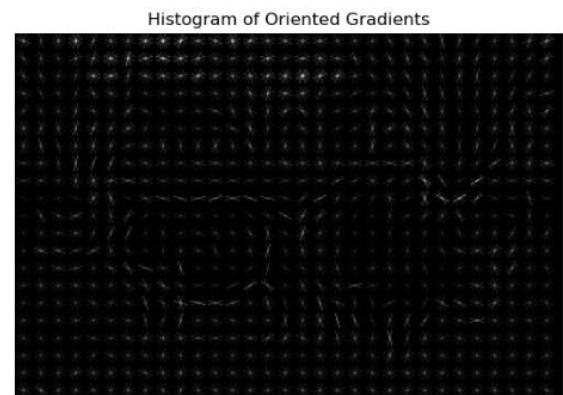
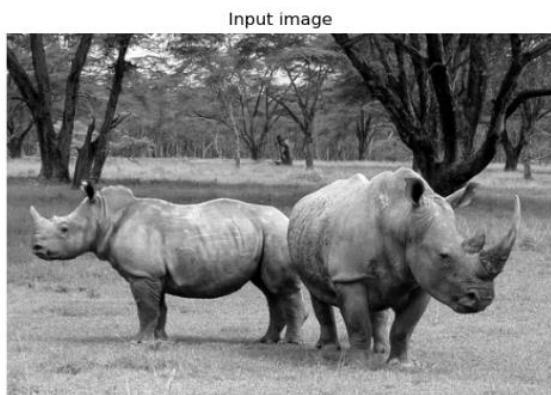
TestCase 1:



TestCase 2:



TestCase 3:



# Chương 8 Image Segmentation

## Felzenszwalb, SLIC, QuickShift, and Compact Watershed algorithms

### 1. Felzenszwalb's efficient graph-based image segmentation

Thuật toán **Felzenszwalb's Graph-Based Image Segmentation** dựa trên việc chia hình ảnh thành các thành phần liên thông (connected components) trên một đồ thị dựa trên trọng số của các cạnh (edges). Đây là một thuật toán mạnh mẽ để thực hiện phân đoạn hình ảnh dựa trên cấu trúc đồ thị và được sử dụng phổ biến để tạo ra **superpixels**.

#### Cách hoạt động của thuật toán

##### 1. Biểu diễn hình ảnh dưới dạng đồ thị:

- **Đỉnh (vertices):** Mỗi pixel trong ảnh được biểu diễn như một đỉnh của đồ thị.
- **Cạnh (edges):** Mỗi cặp pixel lân cận được nối với nhau bằng một cạnh.
- **Trọng số của cạnh (edge weights):** Trọng số biểu diễn độ khác biệt giữa hai pixel, ví dụ:

$$w(p_i, p_j) = |I(p_i) - I(p_j)|$$

- Trong đó,  $I(p_i)$  là giá trị cường độ (intensity) của pixel  $p_i$

##### 2. Tính toán mức độ tương đồng (similarity):

- Trọng số cạnh thấp cho biết hai pixel tương tự nhau (ít khác biệt).
- Trọng số cạnh cao cho biết hai pixel khác biệt.

##### 3. Chia đồ thị thành các cụm liên thông (connected components):

- Sử dụng thuật toán dựa trên đồ thị để tìm các cụm (clusters) sao cho các cạnh trong cùng một cụm có trọng số thấp.
- Các cụm này tương ứng với các vùng (regions) trong ảnh.

##### 4. Tham số hóa: Thuật toán sử dụng các tham số quan trọng để kiểm soát cách phân đoạn:

- **Scale (scale):** Điều chỉnh kích thước của các vùng. Giá trị lớn tạo các vùng lớn hơn.

- **Sigma** (sigma): Làm mịn ảnh trước khi phân đoạn để giảm nhiễu.
- **Minimum size** (min\_size): Đặt kích thước tối thiểu của vùng được phân đoạn.

```

import os
import numpy as np
import matplotlib.pyplot as plt
from skimage.segmentation import felzenszwalb, find_boundaries
from skimage.io import imread
from skimage.util import img_as_float
from matplotlib.colors import LinearSegmentedColormap

# Danh sách file ảnh
image_files = [
    'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\parrot.png',
    'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\birds.png',
    'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\flowers.png',
    'D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\butterfly.png'
]

for imfile in image_files:
    if not os.path.exists(imfile):
        print(f"File not found: {imfile}")
        continue

    # Đọc và xử lý ảnh
    img = img_as_float(imread(imfile)[::2, ::2, :3])

    # Áp dụng thuật toán Felzenszwalb
    segments_fz = felzenszwalb(img, scale=100, sigma=0.5, min_size=400)

    # Xác định ranh giới
    borders = find_boundaries(segments_fz)
    unique_colors = np.unique(segments_fz.ravel())
    segments_fz[borders] = -1 # Đánh dấu ranh giới

    # Tính màu trung bình cho từng phân đoạn
    colors = [np.zeros(3)] # Màu nền
    for color in unique_colors:
        colors.append(np.mean(img[segments_fz == color], axis=0))

    # Tạo colormap
    cm = LinearSegmentedColormap.from_list('palette', colors, N=len(colors))

    # Hiển thị kết quả
    plt.figure(figsize=(20, 10))
    plt.subplot(121)
    plt.imshow(img)
    plt.title('Original Image', size=20)
    plt.axis('off')

    plt.subplot(122)
    plt.imshow(segments_fz, cmap=cm)
    plt.title('Segmented with Felzenszwalb's Method', size=20)
    plt.axis('off')

    plt.show()

```

Thực hiện thuật toán Felzenszwalb:

```
# Áp dụng thuật toán Felzenszwalb
segments_fz = felzenszwalb(img, scale=100, sigma=0.5, min_size=400)
```

Tham số của hàm felzenszwalb:

- img: Ảnh đầu vào.
- scale=100: Tham số điều chỉnh độ lớn của các vùng (vùng lớn hơn nếu giá trị lớn).
- sigma=0.5: Làm mịn ảnh bằng bộ lọc Gaussian để giảm nhiễu.
- min\_size=400: Kích thước tối thiểu của một vùng sau khi phân đoạn.

Kết quả:

- segments\_fz: Một ma trận, mỗi pixel được gán một nhãn số nguyên đại diện cho vùng mà nó thuộc về.

Tìm đường biên phân đoạn

```
# Xác định ranh giới
borders = find_boundaries(segments_fz)
unique_colors = np.unique(segments_fz.ravel())
segments_fz[borders] = -1 # Đánh dấu ranh giới
```

**find\_boundaries:**

- Xác định đường biên giữa các vùng phân đoạn trong ma trận segments\_fz.
- Trả về một ma trận nhị phân (True tại đường biên, False ở phần còn lại).

Tạo bảng màu cho vùng phân đoạn:

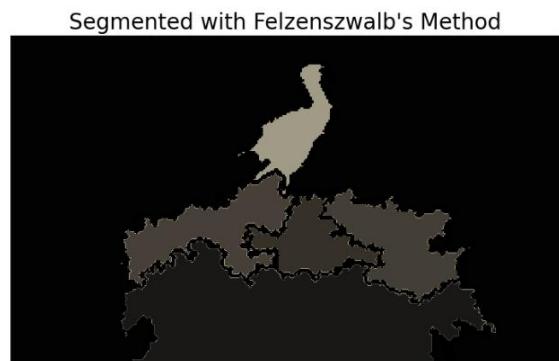
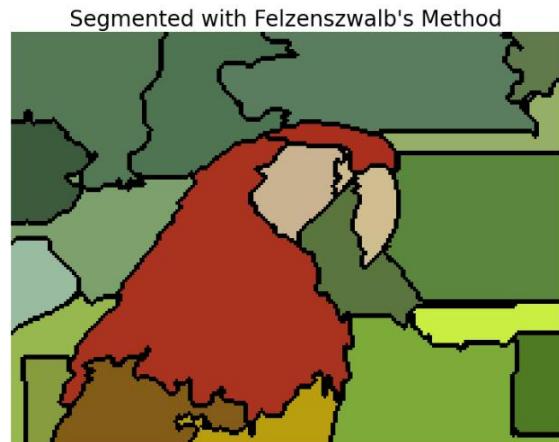
```
# Tính màu trung bình cho từng phân đoạn
colors = [np.zeros(3)] # Màu nền
for color in unique_colors:
    colors.append(np.mean(img[segments_fz == color], axis=0))
```

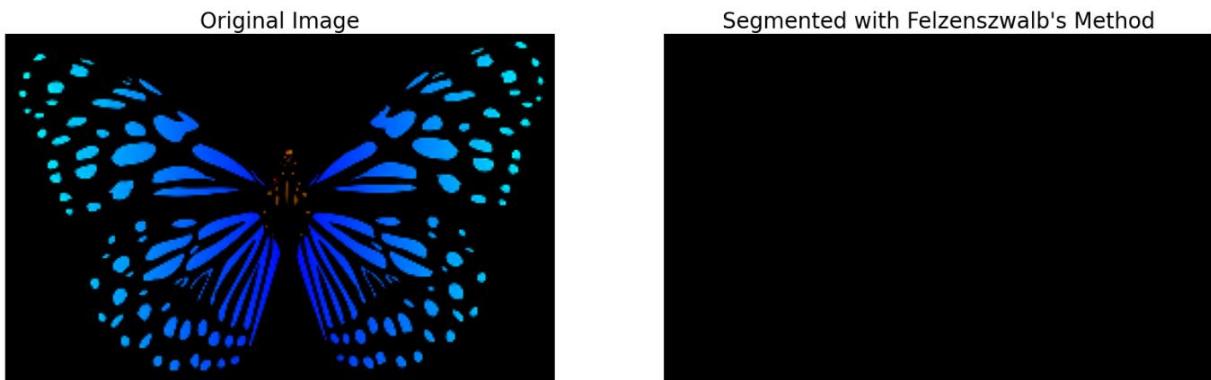
- **unique\_colors:** Lấy danh sách các nhãn vùng trong segments\_fz.
- **Màu sắc trung bình:**
  - Duyệt qua từng vùng (segments\_fz == color).
  - Tính trung bình giá trị RGB của các pixel trong vùng và thêm vào bảng màu colors.

- **Tạo bảng màu:**

- `LinearSegmentedColormap.from_list`: Tạo bảng màu tùy chỉnh từ danh sách colors.

Hiển thị kết quả:





## 2. RAG merging

**RAG (Region Adjacency Graph)** là một đồ thị không có hướng, trong đó:

- **Nút (Node):** Đại diện cho các vùng trong ảnh (ví dụ: vùng từ phân đoạn SLIC).
- **Cạnh (Edge):** Kết nối giữa hai vùng liền kề.
- **Trọng số (Weight):** Thể hiện mức độ khác biệt hoặc tương đồng giữa hai vùng.

RAG được xây dựng để biểu diễn mối quan hệ giữa các vùng liền kề. Dựa trên thông tin này, các vùng tương đồng có thể được hợp nhất lại để tạo ra một phân đoạn hợp lý hơn.

### Quy trình thực hiện RAG Merging

#### Bước 1: Phân đoạn ảnh thành các vùng nhỏ (Superpixels)

- Ban đầu, ảnh được phân đoạn thành các vùng nhỏ hơn (superpixels) bằng một thuật toán như **SLIC** (Simple Linear Iterative Clustering).
- Mỗi vùng nhỏ này được xem là một "nút" trong đồ thị RAG.

#### Bước 2: Xây dựng RAG

- Sau khi phân đoạn, đồ thị RAG được xây dựng. Các cạnh giữa hai nút được gán trọng số dựa trên sự khác biệt giữa các vùng:
  - Ví dụ: Trọng số có thể là **khoảng cách màu sắc trung bình** giữa hai vùng.

#### Bước 3: Hợp nhất các vùng

- **Tiêu chí hợp nhất:** Các vùng liền kề có **trọng số nhỏ hơn một ngưỡng (threshold)** sẽ được hợp nhất.
- Khi hợp nhất:

- Hai vùng được nối thành một vùng mới.
- Các thuộc tính (như màu trung bình) của vùng mới được cập nhật.

#### **Bước 4: Lặp lại cho đến khi không còn vùng nào đáp ứng tiêu chí hợp nhất**

- Quá trình hợp nhất diễn ra liên tục cho đến khi không còn cặp vùng nào có trọng số thấp hơn ngưỡng đã đặt.

```

# Đọc ảnh và chuyển đổi sang kiểu float
image = img_as_float(imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\me12.jpg'))

# Phân đoạn ảnh bằng thuật toán SLIC
segments_slic = segmentation.slic(image, n_segments=300, compactness=10, start_label=1)

# Hàm xây dựng đồ thị dựa trên màu trung bình của các vùng
def build_rag(image, segments):
    rag = Graph() # sử dụng đồ thị từ networkx
    for region in np.unique(segments):
        mask = segments == region
        mean_color = np.mean(image[mask], axis=0)
        rag.add_node(region, mean_color=mean_color, pixel_count=np.sum(mask), total_color=np.sum(image[mask], axis=0))

    # Liên kết các vùng lân cận
    for y in range(image.shape[0] - 1):
        for x in range(image.shape[1] - 1):
            src = segments[y, x]
            neighbors = {segments[y + 1, x], segments[y, x + 1]} # Lấy các vùng lân cận
            for dst in neighbors:
                if src != dst:
                    rag.add_edge(src, dst)

    return rag

# Hàm tính trọng số (dựa trên độ khác biệt màu trung bình giữa các vùng)
def weight_mean_color(graph, src, dst):
    diff = graph.nodes[src]['mean_color'] - graph.nodes[dst]['mean_color']
    return np.linalg.norm(diff)

# Hợp nhất các vùng
def merge_regions(graph, src, dst):
    graph.nodes[dst]['total_color'] += graph.nodes[src]['total_color']
    graph.nodes[dst]['pixel_count'] += graph.nodes[src]['pixel_count']
    graph.nodes[dst]['mean_color'] = graph.nodes[dst]['total_color'] / graph.nodes[dst]['pixel_count']
    graph.remove_node(src)

# Xây dựng đồ thị RAG
rag = build_rag(image, segments_slic)

# Lặp qua các cạnh trong đồ thị, hợp nhất các vùng có trọng số nhỏ hơn ngưỡng
threshold = 35
edges_to_merge = sorted(rag.edges(data=True), key=lambda x: weight_mean_color(rag, x[0], x[1]))
while edges_to_merge:
    src, dst, _ = edges_to_merge.pop(0)
    if src in rag.nodes and dst in rag.nodes: # Kiểm tra xem các node có tồn tại
        if weight_mean_color(rag, src, dst) < threshold:
            merge_regions(rag, src, dst)
            # Cập nhật danh sách cạnh sau mỗi lần hợp nhất
            edges_to_merge = sorted(rag.edges(data=True), key=lambda x: weight_mean_color(rag, x[0], x[1]))

# Chuyển đổi nhãn vùng thành ảnh màu
out_image = color.label2rgb(segments_slic, image, kind='avg', bg_label=0)

# Hiển thị ảnh gốc và ảnh sau khi phân đoạn
fig, ax = plt.subplots(1, 2, figsize=(15, 8))
ax[0].imshow(image)
ax[0].set_title('Original Image')
ax[0].axis('off')

ax[1].imshow(out_image)
ax[1].set_title('Segmented Image (RAG Merging)')
ax[1].axis('off')

```

### 3. Phân đoạn ảnh bằng thuật toán SLIC

```

# Phân đoạn ảnh bằng thuật toán SLIC
segments_slic = segmentation.slic(image, n_segments=300, compactness=10, start_label=1)

```

- Mục đích: Chia ảnh thành các vùng nhỏ bằng thuật toán SLIC (Simple Linear Iterative Clustering).

- n\_segments=300: Số vùng muốn tạo ra (300 vùng nhỏ).
- compactness=10: Tham số kiểm soát sự đồng đều trong phân đoạn.
- start\_label=1: Đặt nhãn vùng bắt đầu từ 1.

Kết quả: Ảnh được chia thành các "superpixel" (vùng nhỏ) để xử lý dễ dàng hơn.

Xây dựng đồ thị RAG (Region Adjacency Graph)

```
# Hàm xây dựng đồ thị dựa trên màu trung bình của các vùng
def build_rag(image, segments):
    rag = Graph() # Sử dụng đồ thị từ networkx
    for region in np.unique(segments):
        mask = segments == region
        mean_color = np.mean(image[mask], axis=0)
        rag.add_node(region, mean_color=mean_color, pixel_count=np.sum(mask), total_color=np.sum(image[mask], axis=0))

    # Liên kết các vùng lân cận
    for y in range(image.shape[0] - 1):
        for x in range(image.shape[1] - 1):
            src = segments[y, x]
            neighbors = {segments[y + 1, x], segments[y, x + 1]} # Lấy các vùng lân cận
            for dst in neighbors:
                if src != dst:
                    rag.add_edge(src, dst)

    return rag
```

**Mục đích:** Tạo đồ thị trong đó mỗi vùng là một nút và mỗi cạnh đại diện cho mối quan hệ giữa các vùng lân cận.

Các bước chính:

### 1. Thêm các nút vào đồ thị:

- Duyệt qua từng vùng trong ảnh (np.unique(segments)).
- Tính **màu trung bình** (mean\_color), **tổng màu** (total\_color), và **số pixel** (pixel\_count) của từng vùng.

### 2. Thêm các cạnh vào đồ thị:

- Xác định các vùng lân cận theo vị trí pixel.
- Thêm cạnh giữa các vùng nếu chúng tiếp xúc.

Hàm tính trọng số dựa trên màu trung bình

```
# Hàm tính trọng số (dựa trên độ khác biệt màu trung bình giữa các vùng)
def weight_mean_color(graph, src, dst):
    diff = graph.nodes[src]['mean_color'] - graph.nodes[dst]['mean_color']
    return np.linalg.norm(diff)
```

Tính độ khác biệt màu trung bình giữa hai vùng (sử dụng chuẩn Euclidean).

## Hàm hợp nhất các vùng

```
# Hợp nhât các vùng
def merge_regions(graph, src, dst):
    graph.nodes[dst]['total_color'] += graph.nodes[src]['total_color']
    graph.nodes[dst]['pixel_count'] += graph.nodes[src]['pixel_count']
    graph.nodes[dst]['mean_color'] = graph.nodes[dst]['total_color'] / graph.nodes[dst]['pixel_count']
    graph.remove_node(src)
```

Hợp nhât hai vùng lân cận:

- Cộng tổng màu (total\_color) và số pixel (pixel\_count) từ vùng src sang vùng dst.
- Cập nhật màu trung bình (mean\_color) của vùng dst.
- Xóa vùng src khỏi đồ thị.

Hợp nhât các vùng

```
# Lặp qua các cạnh trong đồ thị, hợp nhât các vùng có trọng số nhỏ hơn ngưỡng
threshold = 35
edges_to_merge = sorted(rag.edges(data=True), key=lambda x: weight_mean_color(rag, x[0], x[1]))
while edges_to_merge:
    src, dst, _ = edges_to_merge.pop(0)
    if src in rag.nodes and dst in rag.nodes: # Kiểm tra xem các node có tồn tại
        if weight_mean_color(rag, src, dst) < threshold:
            merge_regions(rag, src, dst)
            # Cập nhật danh sách cạnh sau mỗi lần hợp nhât
            edges_to_merge = sorted(rag.edges(data=True), key=lambda x: weight_mean_color(rag, x[0], x[1]))
```

Hợp nhât các vùng có độ khác biệt màu trung bình nhỏ hơn ngưỡng (threshold).

Các bước chính:

1. Duyệt qua các cạnh trong đồ thị (rag.edges), sắp xếp theo trọng số (weight\_mean\_color).
2. Hợp nhât các vùng có trọng số nhỏ hơn ngưỡng.
3. Cập nhật danh sách cạnh sau mỗi lần hợp nhât để đảm bảo tính chính xác.

Chuyển đổi nhãn vùng thành ảnh màu:

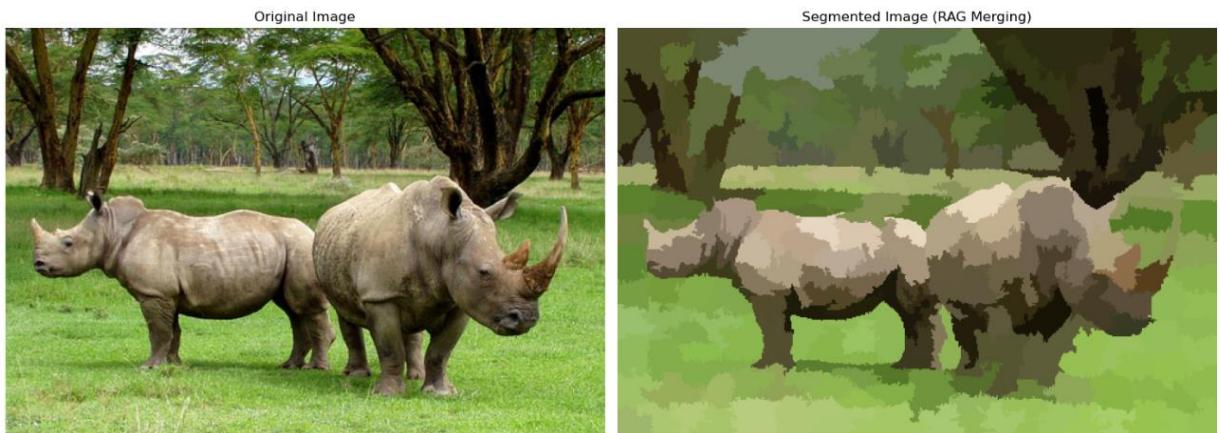
```
# Chuyển đổi nhãn vùng thành ảnh màu
out_image = color.label2rgb(segments_slic, image, kind='avg', bg_label=0)
```

Tạo ảnh màu hiển thị các vùng đã hợp nhât

Hiển thị kết quả:



TestCase 2:



TestCase 3:



## Chương 9 Classical Machine Learning Methods in Image Processing

### 1. K-means clustering for image segmentation with color quantization

## Lượng tử hóa màu sắc bằng K-Means

- Mục tiêu là giảm số lượng màu sắc trong hình ảnh từ số lượng ban đầu (250 màu) xuống một số lượng cụ thể, ví dụ: 64, 32, 16, hoặc 4 màu, trong khi vẫn giữ được chất lượng hình ảnh.
- **Cách thực hiện:**
  - Biểu diễn mỗi pixel trong hình ảnh bằng một điểm dữ liệu trong không gian 3 chiều (tương ứng với các giá trị màu RGB).
  - Áp dụng thuật toán K-Means để phân cụm các điểm dữ liệu (màu sắc) thành kkk cụm. Mỗi cụm sẽ có một **trung tâm cụm (centroid)** đại diện cho các màu sắc trong cụm đó.
  - Gán mỗi pixel trong hình ảnh ban đầu vào màu sắc gần nhất với trung tâm cụm tương ứng, từ đó giảm số lượng màu sắc trong hình ảnh.
- **Chuẩn bị dữ liệu hình ảnh:**
  - Hình ảnh được chuyển đổi sang dạng mảng 3 chiều (w, h, d), trong đó www là chiều rộng, hhh là chiều cao, và ddd là số kênh màu (RGB, tức là 3 kênh).
  - Mảng này được chuyển đổi thành dạng 2D với mỗi hàng là một pixel và mỗi cột là một giá trị kênh màu.
- **Áp dụng K-Means:**
  - Chọn số lượng cụm (kkk) tương ứng với số lượng màu muốn giữ lại.
  - Sử dụng tập mẫu gồm 1000 pixel được chọn ngẫu nhiên từ hình ảnh để huấn luyện mô hình K-Means.
  - Sau khi huấn luyện, thuật toán dự đoán cụm tương ứng cho toàn bộ pixel trong hình ảnh.
- **Tái tạo hình ảnh:**
  - Dựa trên nhãn cụm và các trung tâm cụm, thuật toán thay thế mỗi pixel trong hình ảnh bằng trung tâm cụm tương ứng, tạo ra hình ảnh đã được lượng tử hóa.

## Lượng tử hóa ngẫu nhiên (Random Quantization)

- Để so sánh, hình ảnh cũng được lượng tử hóa bằng cách chọn ngẫu nhiên các màu từ hình ảnh làm trung tâm cụm.
- Điều này không sử dụng thuật toán K-Means, mà chỉ đơn giản chọn kkk màu ngẫu nhiên và gán mỗi pixel vào màu gần nhất trong số các màu được chọn.

Chuẩn bị dữ liệu:

```
# Đọc và chuẩn hóa hình ảnh
pepper = imread("D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\pepper.jpg") # Đổi đường dẫn hình ảnh của bạn
pepper = np.array(pepper, dtype=np.float64) / 255 # Chuẩn hóa giá trị từ 0-255 về 0-1
w, h, d = pepper.shape # Lấy kích thước gốc của hình ảnh
assert d == 3 # Đảm bảo ảnh có 3 kênh (RGB)
image_array = np.reshape(pepper, (w * h, d)) # Chuyển ảnh thành mảng 2D
```

Hàm tái tạo hình ảnh:

```
# Hàm để tái tạo hình ảnh từ các trung tâm cụm và nhãn
def recreate_image(codebook, labels, width, height):
    """
    Tái tạo hình ảnh từ bảng mã màu (codebook) và nhãn cụm.
    """

    depth = codebook.shape[1] # Số kênh màu (RGB)
    image = np.zeros((width, height, depth)) # Tạo mảng 0 cho hình ảnh
    label_idx = 0
    for i in range(width):
        for j in range(height):
            image[i][j] = codebook[labels[label_idx]] # Gán màu từ bảng mã
            label_idx += 1
    return image
```

Duyệt qua từng pixel theo thứ tự, thay thế pixel bằng màu từ trung tâm cụm tương ứng trong bảng màu.

Lượng tử hóa màu với K-Means:

```
# Lượng tử hóa màu bằng K-Means
plt.figure(2, figsize=(10, 10))
for i, k in enumerate([64, 32, 16, 4], 1):
    # Mẫu hóa ảnh cho quá trình K-Means
    image_array_sample = shuffle(image_array, random_state=0)[:1000]

    # Áp dụng thuật toán K-Means
    print(f"Running K-Means with k={k}...")
    t0 = time()
    kmeans = KMeans(n_clusters=k, random_state=0).fit(image_array_sample)
    print(f"K-Means done in {time() - t0:.3f}s.")

    # Dự đoán nhãn cụm cho tất cả pixel
    print("Predicting labels for the full image...")
    t0 = time()
    labels = kmeans.predict(image_array)
    print(f"Prediction done in {time() - t0:.3f}s.")

    # Tái tạo hình ảnh đã lượng tử hóa
    plt.subplot(2, 2, i)
    plt.axis('off')
    plt.title(f'Quantized Image ({k} Colors, K-Means)')
    quantized_image = recreate_image(kmeans.cluster_centers_, labels, w, h)
    plt.imshow(quantized_image)
```

Lặp qua các giá trị kkk: Với  $k=64, 32, 16, 4$ ,  $k=64, 32, 16, 4$ ,  $k=64, 32, 16, 4$ , thực hiện lượng tử hóa màu sử dụng K-Means.

**Lấy mẫu ngẫu nhiên:** Lấy 1000 pixel ngẫu nhiên để huấn luyện mô hình K-Means (giảm khối lượng tính toán).

**Huấn luyện K-Means:** Chia dữ liệu thành kkk cụm và tính toán trung tâm cụm (bảng màu).

**Dự đoán nhãn cụm:** Dựa trên trung tâm cụm, gán mỗi pixel vào cụm gần nhất.

**Hiển thị hình ảnh đã lượng tử hóa:** Tái tạo hình ảnh sử dụng bảng màu mới với kkk màu.

Lượng tử hóa màu ngẫu nhiên:

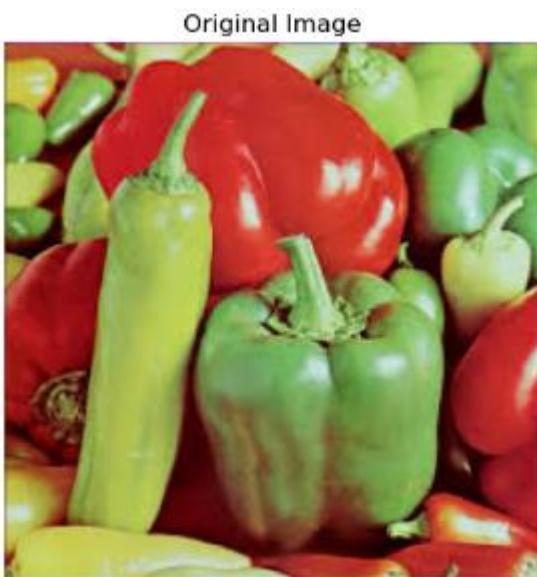
```
# Lượng tử hóa màu bằng cách chọn ngẫu nhiên bảng mã
plt.figure(3, figsize=(10, 10))
for i, k in enumerate([64, 32, 16, 4], 1):
    # Tạo bảng mã màu ngẫu nhiên
    codebook_random = shuffle(image_array, random_state=0)[:k]

    # Dự đoán nhãn cụm dựa trên bảng mã ngẫu nhiên
    print(f"Running Random Quantization with k={k}...")
    t0 = time()
    labels_random = pairwise_distances_argmin(codebook_random, image_array, axis=0)
    print(f"Random Quantization done in {time() - t0:.3f}s.")

    # Tái tạo hình ảnh đã lượng tử hóa
    plt.subplot(2, 2, i)
    plt.axis('off')
    plt.title(f'Quantized Image ({k} Colors, Random)')
    random_image = recreate_image(codebook_random, labels_random, w, h)
    plt.imshow(random_image)
```

Thực hiện lượng tử hóa màu bằng cách chọn kkk màu ngẫu nhiên thay vì sử dụng thuật toán K-Means.

Hiển thị kết quả:



Quantized Image (64 Colors, K-Means)



Quantized Image (32 Colors, K-Means)



Quantized Image (16 Colors, K-Means)



Quantized Image (4 Colors, K-Means)



Quantized Image (64 Colors, Random)



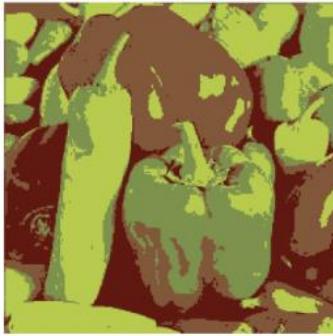
Quantized Image (32 Colors, Random)



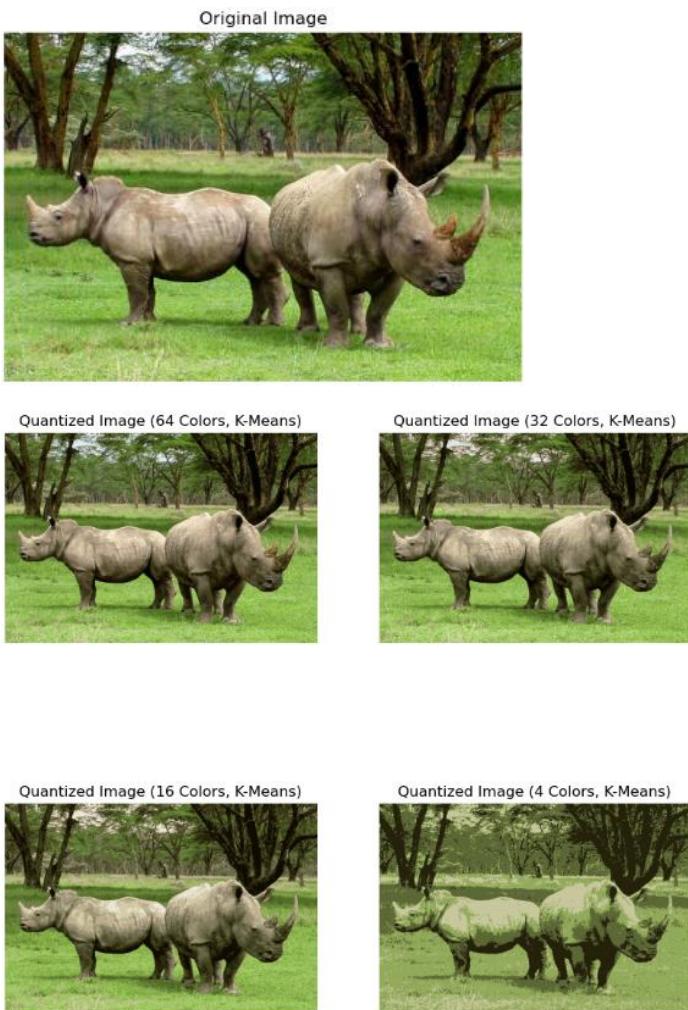
Quantized Image (16 Colors, Random)



Quantized Image (4 Colors, Random)



## TestCase 2:



Quantized Image (64 Colors, Random)



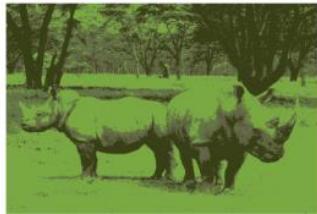
Quantized Image (32 Colors, Random)



Quantized Image (16 Colors, Random)



Quantized Image (4 Colors, Random)



### TestCase 3:

Original Image



Quantized Image (64 Colors, K-Means)



Quantized Image (32 Colors, K-Means)



Quantized Image (16 Colors, K-Means)



Quantized Image (4 Colors, K-Means)



Quantized Image (64 Colors, Random)



Quantized Image (32 Colors, Random)



Quantized Image (16 Colors, Random)



Quantized Image (4 Colors, Random)



## 2. Supervised machine learning – image classification

Chuẩn bị data: Downloading the MNIST (handwritten digits) dataset

```

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Tải dữ liệu MNIST từ TensorFlow
def load_mnist_data():
    # Tải tập dữ liệu (tự động tải về nếu chưa có)
    (train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()

    # Chuẩn hóa dữ liệu: đưa các giá trị pixel về khoảng [0, 1]
    train_images = train_images.reshape(-1, 784).astype('float32') / 255.0
    test_images = test_images.reshape(-1, 784).astype('float32') / 255.0

    return train_images, train_labels, test_images, test_labels

# Hàm hiển thị một số hình ảnh từ tập dữ liệu
def display_samples(images, labels, num_samples=10):
    plt.figure(figsize=(10, 2))
    for i in range(num_samples):
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        plt.title(f"Label: {labels[i]}")
        plt.axis('off')
    plt.show()

# Tải và xử lý dữ liệu MNIST
train_data, train_labels, test_data, test_labels = load_mnist_data()

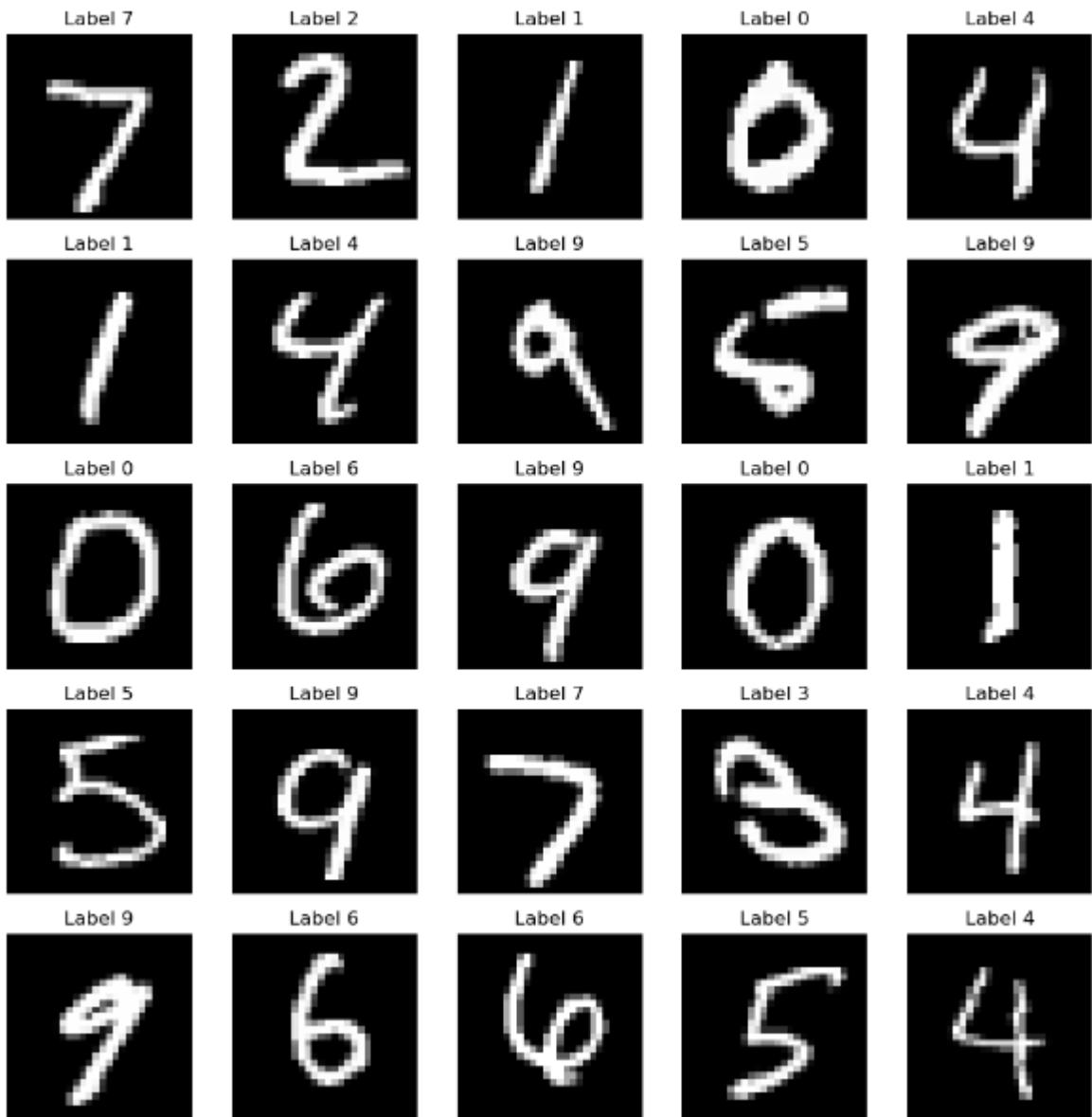
# Hiển thị thông tin dataset
print(f"Training data shape: {train_data.shape}")
print(f"Training labels shape: {train_labels.shape}")
print(f"Testing data shape: {test_data.shape}")
print(f"Testing labels shape: {test_labels.shape}")

# Hiển thị một số mẫu từ tập dữ liệu huấn luyện
display_samples(train_data, train_labels, num_samples=10)

```

Hiển thị kết quả data:

Label: 5 Label: 0 Label: 4 Label: 1 Label: 9 Label: 2 Label: 1 Label: 3 Label: 1 Label: 4



k-nearest neighbors (KNN) classifier:

```

# Computing the nearest neighbors
import time
from sklearn.neighbors import BallTree
import numpy as np

# Build nearest neighbor structure on training data
t_before = time.time()
ball_tree = BallTree(train_data)
t_after = time.time()

# Compute training time
t_training = t_after - t_before
print("Time to build data structure (seconds):", t_training)

# Get nearest neighbor predictions on testing data
t_before = time.time()
test_neighbors = np.squeeze(
    ball_tree.query(test_data, k=1, return_distance=False)
)
test_predictions = train_labels[test_neighbors]
t_after = time.time()

# Compute testing time
t_testing = t_after - t_before
print("Time to classify test set (seconds):", t_testing)

```

Sử dụng cấu trúc BallTree từ thư viện scikit-learn để thực hiện tìm kiếm láng giềng gần nhất (Nearest Neighbors Search)

Evaluating the performance of the classifier( Đánh giá hiệu suất của bộ phân loại):

```

# Evaluate the classifier
t_accuracy = sum(test_predictions == test_labels) / float(len(test_labels))
print(f"Accuracy: {t_accuracy:.4f}") # In ra độ chính xác (với 4 chữ số thập phân)

# Example accuracy output: 0.9691

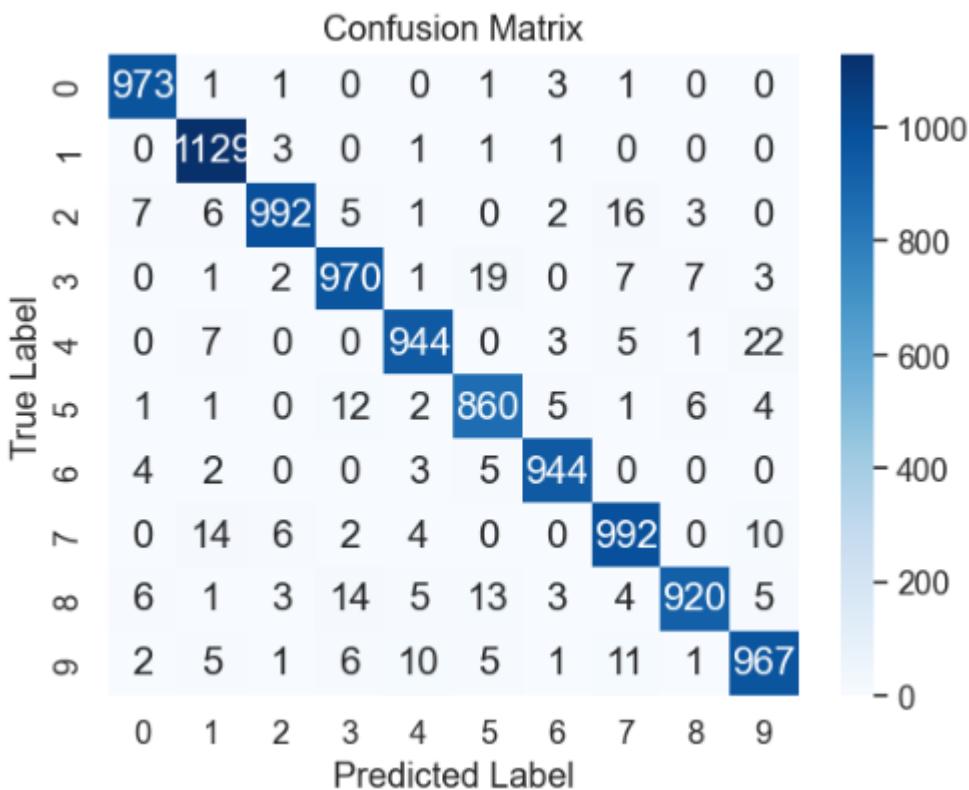
# Import libraries for confusion matrix visualization
import pandas as pd
import seaborn as sn
from sklearn import metrics

# Compute the confusion matrix
cm = metrics.confusion_matrix(test_labels, test_predictions)

# Create a DataFrame for better visualization
df_cm = pd.DataFrame(cm, range(10), range(10))

# Plot the confusion matrix
sn.set(font_scale=1.2) # Set font scale for labels
sn.heatmap(df_cm, annot=True, annot_kws={"size": 16}, fmt="g", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

```



Bayes classifier (Gaussian generative model):

```

import numpy as np
import matplotlib.pyplot as plt

# Function to display a single character image
def display_char(image):
    plt.imshow(np.reshape(image, (28, 28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()

# Function to fit a generative model
def fit_generative_model(x, y):
    k = 10 # Number of labels: 0, 1, ..., k-1
    d = x.shape[1] # Number of features (dimensionality of each data point)

    # Initialize mean (mu), covariance (sigma), and prior probabilities (pi)
    mu = np.zeros((k, d))
    sigma = np.zeros((k, d, d))
    pi = np.zeros(k)

    c = 3500 # Regularization constant to avoid singularity in covariance matrix

    # Compute the parameters for each label
    for label in range(k):
        indices = (y == label) # Get indices of samples belonging to the current label
        pi[label] = sum(indices) / float(len(y)) # Prior probability for the label
        mu[label] = np.mean(x[indices, :], axis=0) # Mean vector for the label
        sigma[label] = np.cov(x[indices, :], rowvar=False, bias=True) + c * np.eye(d) # Regularized covariance matrix

    return mu, sigma, pi

# Fit the generative model to training data
mu, sigma, pi = fit_generative_model(train_data, train_labels)

# Display the mean image for each label
display_char(mu[0]) # Display the mean image for label 0
display_char(mu[1]) # Display the mean image for label 1
display_char(mu[2]) # Display the mean image for label 2

```

Hàm fit\_generative\_model:

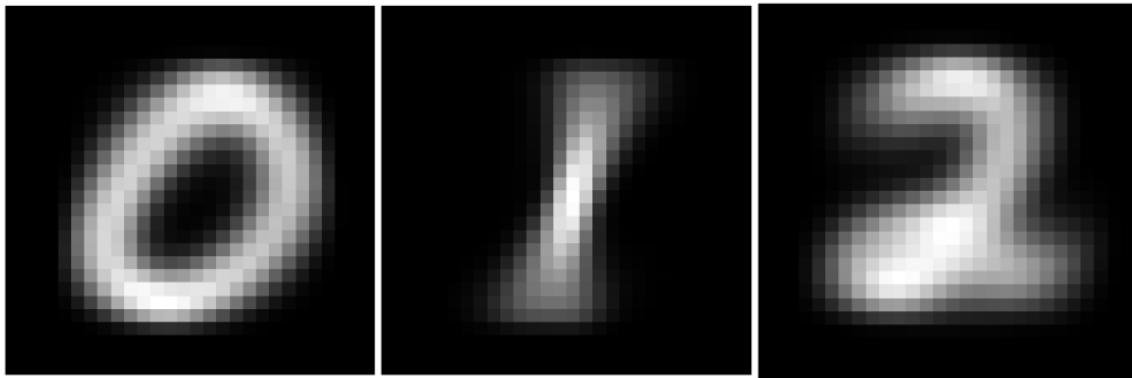
học mô hình sinh (generative model) cho bài toán phân loại. Nó tính toán các tham số sau cho mỗi nhãn (0-9):

- **mu[label]**: Trung bình của các điểm dữ liệu trong lớp label.
- **sigma[label]**: Ma trận hiệp phương sai của các điểm dữ liệu trong lớp label.
- **pi[label]**: Xác suất tiên nghiệm của lớp label (tỷ lệ phần trăm mẫu thuộc lớp đó trong dữ liệu).

**Mô hình sinh (Generative Model)** là mô hình học xác suất, trong đó chúng ta học được các phân phối xác suất của dữ liệu cho từng lớp (label). Đối với mỗi lớp học các tham số như:

- **Trung bình (mu)**: Trung bình của các đặc trưng trong lớp đó.
- **Ma trận hiệp phương sai (sigma)**: Đo lường sự phân tán của các đặc trưng trong lớp.
- **Xác suất tiên nghiệm (pi)**: Xác suất của mỗi lớp trong dữ liệu.

**Hiển thị ảnh trung bình:** Sau khi tính toán trung bình cho mỗi lớp, hiển thị ảnh trung bình đó, giúp hình dung được "khuôn mặt" đại diện của mỗi chữ số từ 0 đến 9.



Tính toán xác suất sau để đưa ra dự đoán về dữ liệu thử nghiệm và đánh giá mô hình:

$$\max_j \mathbf{P}(y=j|x) \propto \max_j \mathbf{P}(x|y=j) \mathbf{P}(y=j) = \max_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \pi_j$$

↓

$$\max_j \log \mathbf{P}(y=j|x) \propto \max_j (\log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) + \log \pi_j)$$

```
# Compute log Pr(label|image) for each [test image, label] pair.
k = 10 # Number of classes (0, 1, ..., 9)
score = np.zeros((len(test_labels), k)) # Array to store log probabilities for each class

# Loop over each label (class)
for label in range(k):
    # Create a multivariate normal distribution for each label using its mean and covariance
    rv = multivariate_normal(mean=mu[label], cov=sigma[label])

    # Loop over each test image
    for i in range(len(test_labels)):
        # Compute the log of the posterior probability for the current test image and label
        score[i, label] = np.log(pi[label]) + rv.logpdf(test_data[i, :])

# Get the class predictions by selecting the label with the highest log probability
test_predictions = np.argmax(score, axis=1)

# Finally, tally up the errors
errors = np.sum(test_predictions != test_labels)
print(f"The generative model makes {errors} errors out of 10000")

# Calculate accuracy
t_accuracy = sum(test_predictions == test_labels) / float(len(test_labels))
print(f"Accuracy: {t_accuracy}")
```

SVM classifier:

```

from sklearn.svm import SVC
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import metrics

# Khởi tạo mô hình SVM với kernel polynomial và degree = 2
clf = SVC(C=1, kernel='poly', degree=2)

# Huấn luyện mô hình với dữ liệu huấn luyện
clf.fit(train_data, train_labels)

# In ra độ chính xác của mô hình trên tập kiểm tra
print(clf.score(test_data, test_labels)) # Output ví dụ: 0.9806

# Dự đoán nhãn của tập kiểm tra
test_predictions = clf.predict(test_data)

# Tính ma trận nhầm lẫn (confusion matrix)
cm = metrics.confusion_matrix(test_labels, test_predictions)

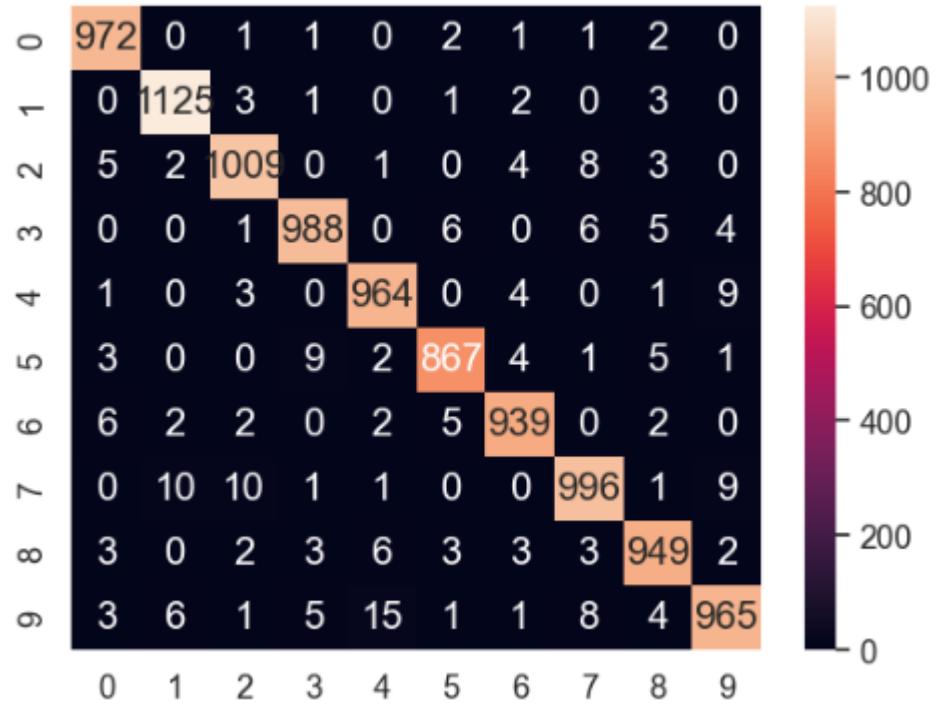
# Chuyển ma trận nhầm lẫn thành DataFrame để dễ dàng thao tác và hiển thị
df_cm = pd.DataFrame(cm, range(10), range(10))

# Cài đặt phông chữ cho kích thước nhãn
sn.set(font_scale=1.2)

# Hiển thị ma trận nhầm lẫn dưới dạng biểu đồ heatmap
sn.heatmap(df_cm, annot=True, annot_kws={"size": 16}, fmt="g")

# Hiển thị biểu đồ
plt.show()

```



Ma trận cho thấy độ chính xác cao trong việc phân loại các chữ số

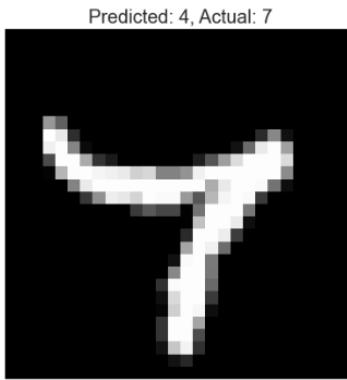
Thực hiện dự đoán:

```
wrong_indices = test_predictions != test_labels
wrong_digits, wrong_preds, correct_labs = test_data[wrong_indices], test_predictions[wrong_indices], test_labels[wrong_indices]

# Print the number of wrong predictions
print(len(wrong_preds)) # Output: 194

# Display the first incorrect prediction
pylab.title('Predicted: ' + str(wrong_preds[1]) + ', Actual: ' + str(correct_labs[1]))
display_char(wrong_digits[1])
```

Kết quả dự đoán:



## Chương 10 Deep Learning in Image Processing - Image Classification

### 1. Classification with TF

**TensorFlow** là thư viện phổ biến nhất trong sản xuất mô hình deep learning. Nó có cộng đồng rộng lớn và mạnh mẽ, nhưng hơi khó sử dụng đối với người mới bắt đầu.

**Keras** là một API cấp cao xây dựng trên TensorFlow, giúp người dùng dễ dàng xây dựng các mô hình deep learning mà không cần quan tâm quá nhiều đến các chi tiết kỹ thuật thấp (low-level). Tuy nhiên, điều này cũng có nghĩa là nó không linh hoạt bằng TensorFlow khi người dùng cần can thiệp sâu vào mô hình.

#### Mạng nơ-ron sâu đơn giản (DNN)

- **Mô hình mạng:** Sử dụng một **mạng nơ-ron sâu** với chỉ một lớp ẩn (Fully Connected - FC) sử dụng hàm kích hoạt **ReLU**, và một lớp **softmax FC** cho lớp đầu ra. Không sử dụng lớp tích chập (Convolutional Layers) trong mô hình này.
- **Cấu trúc mạng:**
  - **Input Layer:** Mỗi hình ảnh có kích thước 28x28 pixels, tức là **784** giá trị đầu vào ( $28 * 28$ ).
  - **Hidden Layer:** Lớp ẩn với **1,024** nơ-ron.

- **Output Layer:** Lớp đầu ra có **10 nơ-ron**, mỗi nơ-ron tương ứng với một chữ số từ 0 đến 9 (10 lớp cho các chữ số cần phân loại).

## Tiền xử lý dữ liệu

- **Tải dữ liệu MNIST:** Tập dữ liệu MNIST bao gồm các hình ảnh chữ số viết tay và nhãn tương ứng. Mỗi hình ảnh có kích thước 28x28 pixels, với nhãn là các chữ số từ 0 đến 9.
- **Chia tập dữ liệu:** Chia dữ liệu huấn luyện thành hai phần:
  - **Tập huấn luyện:** Sử dụng 50.000 hình ảnh.
  - **Tập kiểm tra (Validation):** Sử dụng 10.000 hình ảnh để kiểm tra mô hình trong quá trình huấn luyện.

## Chuẩn hóa và One-hot Encoding

- **Chuyển đổi nhãn thành One-hot Encoding:** Để mô hình có thể hiểu được nhãn, các nhãn (số từ 0 đến 9) được chuyển thành các vector nhị phân one-hot. Ví dụ, nhãn 3 sẽ được mã hóa thành [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].
- **Chuẩn hóa hình ảnh:** Đưa các giá trị pixel vào khoảng [0, 1] bằng cách chia cho 255.

## Xây dựng mô hình trong TensorFlow

- **Khởi tạo TensorFlow Graph:** Khởi tạo đồ thị tính toán của TensorFlow, nơi các phép toán (như hàm kích hoạt, tính toán loss, cập nhật trọng số) sẽ được thực hiện.
- **Các Tensor:**
  - **Placeholder Tensors:** Dùng để chứa dữ liệu đầu vào (hình ảnh và nhãn) trong quá trình huấn luyện.
  - **Variable Tensors:** Dùng để lưu trữ các trọng số (weights) của mô hình.
  - **Constant Tensors:** Dùng cho các hằng số không thay đổi trong quá trình huấn luyện.

## Sử dụng Stochastic Gradient Descent (SGD)

- **Optimizer:** Sử dụng thuật toán **Mini-batch Stochastic Gradient Descent (SGD)** để tối ưu hóa mô hình. SGD sử dụng các mini-batch để cập nhật trọng số sau mỗi bước huấn luyện thay vì tính toán trên toàn bộ dữ liệu.
- **Batch Size:** Sử dụng kích thước mini-batch là **256**.

- **Loss Function:** Sử dụng hàm mất mát **Softmax Cross-Entropy** để tính toán lỗi giữa dự đoán của mô hình và nhãn thực tế.
- **Regularization:** Áp dụng **L2 regularization** cho các trọng số của các lớp để tránh quá khớp (overfitting). Các hyperparameter  $\lambda_1=\lambda_2=1$  được dùng để điều chỉnh mức độ regularization.

### **Huấn luyện và Đánh giá mô hình**

- **Số bước huấn luyện:** Mô hình sẽ được huấn luyện qua **6.000 bước** (mini-batches).
- **Forward/backpropagation:** Trong mỗi bước, sẽ thực hiện quá trình **forward propagation** (đi qua các lớp của mô hình) để tính toán đầu ra, và **backpropagation** để tính toán gradient và cập nhật trọng số.
- **Đánh giá mô hình:** Sau khi huấn luyện, mô hình được kiểm tra trên tập kiểm tra để đánh giá độ chính xác của nó.

```

# 1. Tải dữ liệu MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# 2. Chuẩn hóa dữ liệu và chuyển đổi nhãn thành dạng one-hot
X_train = X_train / 255.0
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 3. Chia tập validation từ tập training
np.random.seed(0)
train_indices = np.random.choice(X_train.shape[0], 50000, replace=False)
valid_indices = [i for i in range(X_train.shape[0]) if i not in train_indices]
X_valid, y_valid = X_train[valid_indices], y_train[valid_indices]
X_train, y_train = X_train[train_indices], y_train[train_indices]

# 4. Xây dựng mô hình Sequential
model = Sequential([
    Flatten(input_shape=(28, 28)), # Chuyển ảnh 28x28 thành vector 784 phần tử
    Dense(1024, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.1)), # Lớp ẩn
    Dense(10, activation='softmax') # Lớp đầu ra với 10 lớp
])

# 5. Compile mô hình
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.008),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 6. Huấn luyện mô hình
history = model.fit(X_train, y_train,
                      epochs=10,
                      batch_size=256,
                      validation_data=(X_valid, y_valid))

# 7. Đánh giá mô hình trên tập test
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc * 100:.2f}%")

# 8. Trực quan hóa trọng số của lớp ẩn đầu tiên
weights_layer_1, _ = model.layers[1].get_weights()

plt.figure(figsize=(18, 18))
indices = np.random.choice(weights_layer_1.shape[1], 225) # Chọn 225 trọng số ngẫu nhiên
for i, idx in enumerate(indices):
    plt.subplot(15, 15, i + 1)
    plt.imshow(weights_layer_1[:, idx].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.suptitle("Visualization of Weights in the Hidden Layer")
plt.show()

# 9. Vẽ biểu đồ Accuracy và Loss trong quá trình huấn luyện
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

```

Chia tập validation:

```
# 3. Chia tập validation từ tập training
np.random.seed(0)
train_indices = np.random.choice(X_train.shape[0], 50000, replace=False)
valid_indices = [i for i in range(X_train.shape[0]) if i not in train_indices]
X_valid, y_valid = X_train[valid_indices], y_train[valid_indices]
X_train, y_train = X_train[train_indices], y_train[train_indices]
```

**Chọn ngẫu nhiên 50,000 mẫu để huấn luyện:**

- np.random.choice: Lấy 50,000 mẫu ngẫu nhiên từ tập huấn luyện gốc.

**Tách tập validation:**

- Tập còn lại (10,000 mẫu) được sử dụng làm tập **validation** để đánh giá mô hình trong quá trình huấn luyện.

**Vì sao cần tập validation?:**

- Tập validation giúp kiểm tra khả năng tổng quát của mô hình trong lúc huấn luyện, tránh việc **overfitting** (học quá kỹ trên tập huấn luyện).

```
# 4. Xây dựng mô hình Sequential
model = Sequential([
    Flatten(input_shape=(28, 28)), # Chuyển ảnh 28x28 thành vector 784 phần tử
    Dense(1024, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.1)), # Lớp ẩn
    Dense(10, activation='softmax') # Lớp đầu ra với 10 lớp
])
```

**Mô hình Sequential:**

- Dùng để xếp chồng các lớp của mạng nơ-ron (theo thứ tự).

**Lớp Flatten:**

- Chuyển hình ảnh  $28 \times 28$  thành một vector 784 phần tử (1 chiều) để đưa vào lớp Dense.
- Ví dụ: Một hình  $28 \times 28 \rightarrow [x_1, x_2, \dots, x_{784}]$ .

**Lớp Dense(1024):**

- **1024**: Số nơ-ron trong lớp ẩn.
- **activation='relu'**: Dùng hàm ReLU (Rectified Linear Unit) để kích hoạt. Công thức:
  - $ReLU(x) = \max(0, x)$  (lọc ra giá trị âm).
- **kernel\_regularizer=l2(0.1)**: Regularization L2 giúp giảm overfitting bằng cách thêm ràng buộc vào trọng số.

## Lớp Dense(10):

- **10:** Số nơ-ron tương ứng với 10 chữ số (0-9).
- **activation='softmax':** Softmax chuyển đổi ra thành xác suất, với tổng tất cả giá trị = 1.

Huấn luyện mô hình:

```
# 6. Huấn luyện mô hình
history = model.fit(X_train, y_train,
                     epochs=10,
                     batch_size=256,
                     validation_data=(X_valid, y_valid))
```

Trực quan hóa trọng số:

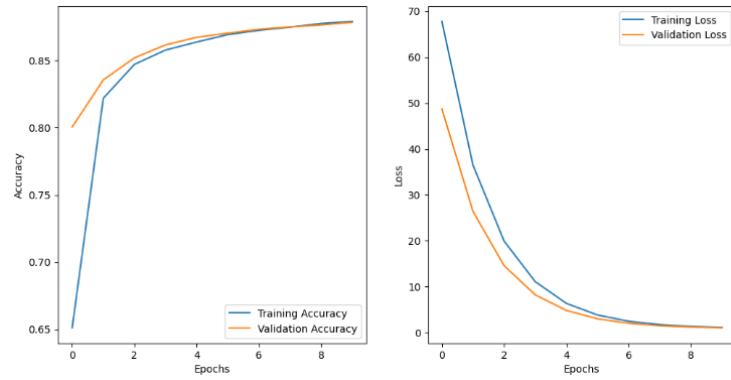
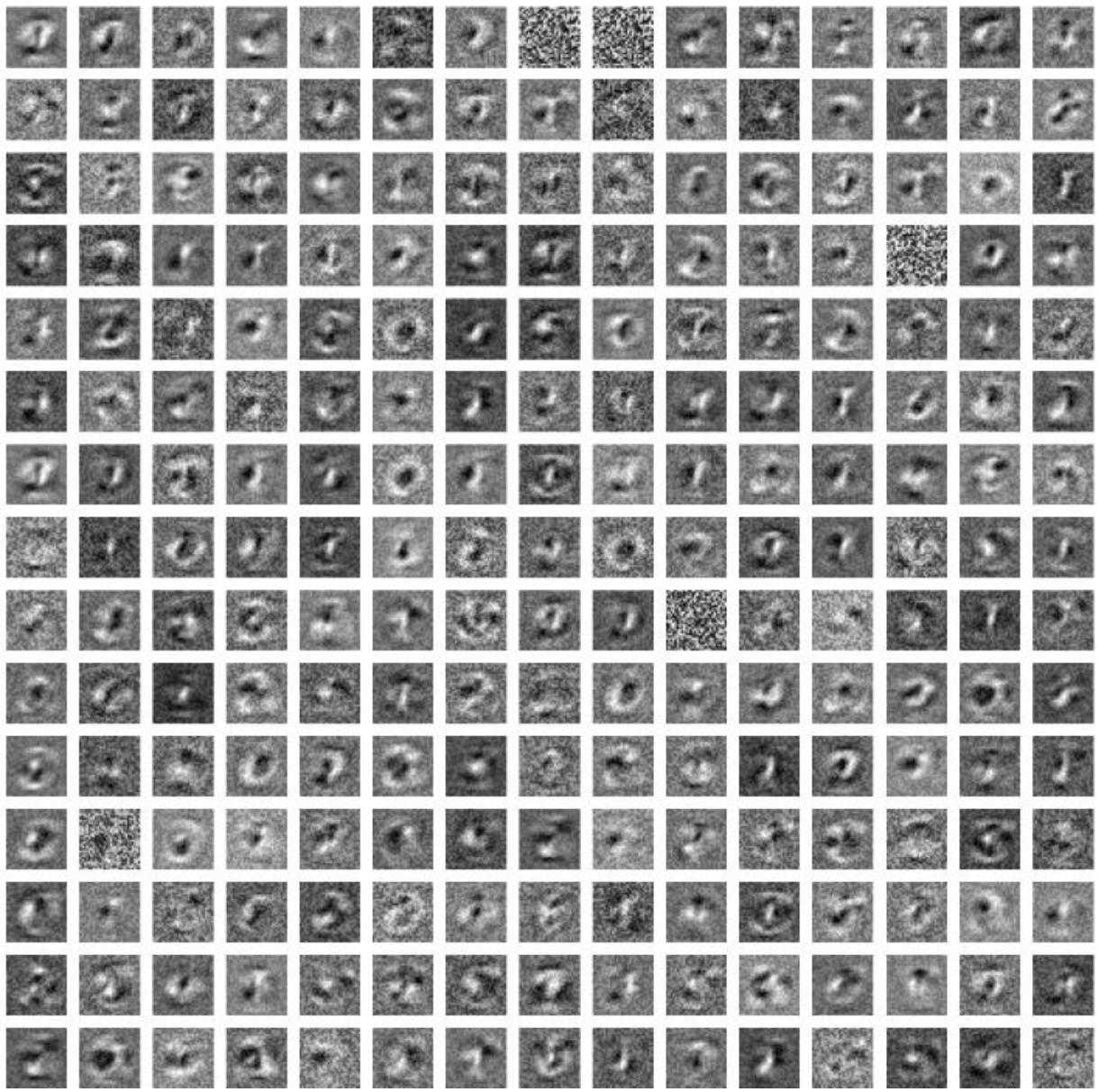
```
# 8. Trực quan hóa trọng số của lớp ẩn đầu tiên
weights_layer_1, _ = model.layers[1].get_weights()

plt.figure(figsize=(18, 18))
indices = np.random.choice(weights_layer_1.shape[1], 225) # Chọn 225 trọng số ngẫu nhiên
for i, idx in enumerate(indices):
    plt.subplot(15, 15, i + 1)
    plt.imshow(weights_layer_1[:, idx].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.suptitle("Visualization of Weights in the Hidden Layer")
plt.show()
```

Trực quan hóa **trọng số** (weights) từ lớp ẩn đầu tiên trong mạng nơ-ron sâu (neural network), giúp chúng ta hiểu cách mà mô hình học và trích xuất các đặc trưng từ dữ liệu đầu vào

Hiển thị kết quả:

```
Epoch 1/10
196/196 4s 15ms/step - accuracy: 0.4801 - loss: 78.4869 - val_accuracy: 0.8005 - val_loss: 48.7064
Epoch 2/10
196/196 3s 13ms/step - accuracy: 0.8111 - loss: 42.1397 - val_accuracy: 0.8356 - val_loss: 26.4272
Epoch 3/10
196/196 3s 13ms/step - accuracy: 0.8420 - loss: 22.9427 - val_accuracy: 0.8519 - val_loss: 14.5709
Epoch 4/10
196/196 3s 13ms/step - accuracy: 0.8550 - loss: 12.7096 - val_accuracy: 0.8616 - val_loss: 8.2383
Epoch 5/10
196/196 2s 12ms/step - accuracy: 0.8634 - loss: 7.2433 - val_accuracy: 0.8672 - val_loss: 4.8474
Epoch 6/10
196/196 2s 12ms/step - accuracy: 0.8686 - loss: 4.3136 - val_accuracy: 0.8703 - val_loss: 3.0278
Epoch 7/10
196/196 2s 12ms/step - accuracy: 0.8723 - loss: 2.7425 - val_accuracy: 0.8732 - val_loss: 2.0469
Epoch 8/10
196/196 3s 13ms/step - accuracy: 0.8725 - loss: 1.8926 - val_accuracy: 0.8750 - val_loss: 1.5149
Epoch 9/10
196/196 3s 14ms/step - accuracy: 0.8773 - loss: 1.4281 - val_accuracy: 0.8764 - val_loss: 1.2233
Epoch 10/10
196/196 3s 16ms/step - accuracy: 0.8778 - loss: 1.1743 - val_accuracy: 0.8784 - val_loss: 1.0604
313/313 1s 3ms/step - accuracy: 0.8690 - loss: 1.1003
Test accuracy: 88.74%
```



**Biểu đồ Accuracy - Độ chính xác:**

- **Trục hoành (Epochs):** Số lượng epoch (vòng lặp huấn luyện toàn bộ dữ liệu).

- **Trục tung (Accuracy):** Độ chính xác của mô hình (giá trị từ 0 đến 1).
- **Đường màu xanh:** Độ chính xác trên tập huấn luyện (Training Accuracy).
- **Đường màu cam:** Độ chính xác trên tập kiểm tra/đánh giá (Validation Accuracy).
- Nhận xét:
  - Độ chính xác tăng lên qua các epoch, cho thấy mô hình đang học tốt hơn.
  - Đường "Training Accuracy" và "Validation Accuracy" khá sát nhau, điều này cho thấy mô hình không bị overfitting (quá khớp).

### **Biểu đồ Loss - Hàm mất mát:**

- **Trục hoành (Epochs):** Số lượng epoch.
- **Trục tung (Loss):** Giá trị hàm mất mát (Loss).
- **Đường màu xanh:** Hàm mất mát trên tập huấn luyện (Training Loss).
- **Đường màu cam:** Hàm mất mát trên tập kiểm tra/đánh giá (Validation Loss).
- Nhận xét:
  - Hàm mất mát giảm dần qua từng epoch, cho thấy mô hình đang hội tụ tốt.
  - Đường "Training Loss" và "Validation Loss" cũng khá gần nhau, biểu hiện tốt của mô hình, không xảy ra hiện tượng underfitting (không đủ khớp).

## **Chương 11 Deep Learning in Image Processing - Object Detection, and more**

### **1. Phân đoạn sứ ảnh sử dụng DeepLab-V3**

Thuật toán trong đoạn mã trên sử dụng DeeplabV3, một mô hình phân đoạn ảnh tiên tiến, thường được sử dụng cho các tác vụ phân đoạn semantic ảnh. DeeplabV3 sử dụng một cấu trúc mạng học sâu mạnh mẽ với các lớp Convolutional, Batch Normalization và Atrous Convolutions (dilated convolutions) để phân đoạn các đối tượng trong ảnh.

DeeplabV3 có thể phân loại mỗi pixel trong ảnh thành các lớp khác nhau (ví dụ: người, xe, cây, v.v.) dựa trên đặc điểm của từng vùng trong ảnh. Việc phân đoạn ảnh giúp tách biệt các đối tượng riêng biệt trong bức ảnh và rất hữu ích trong các ứng dụng như xe tự lái, phân tích hình ảnh y tế, và nhiều ứng dụng khác.

### **Các bước thực hiện thuật toán:**

#### **1. Tiền xử lý ảnh:**

- Đọc ảnh từ đường dẫn đã chỉ định.
- Resize ảnh sao cho chiều dài hoặc chiều rộng lớn nhất bằng 512 pixels mà vẫn giữ tỷ lệ khung hình ban đầu.
- Chuẩn hóa ảnh để đầu vào có giá trị từ -1 đến 1, giúp cải thiện hiệu suất của mô hình.

## 2. Thêm padding (nếu cần):

- Đảm bảo ảnh có kích thước cố định 512x512 để DeeplabV3 có thể xử lý một cách hiệu quả. Nếu chiều cao hoặc chiều rộng của ảnh không đủ 512, ta sẽ thêm padding vào.

## 3. Dự đoán phân đoạn:

- Ảnh đã tiền xử lý sẽ được đưa vào mô hình DeeplabV3 để phân đoạn.
- Mô hình trả về các xác suất của các lớp phân đoạn cho từng pixel. Chúng ta sử dụng np.argmax để chọn lớp phân đoạn có xác suất cao nhất cho mỗi pixel.

## 4. Hiển thị và lưu ảnh phân đoạn:

Dùng matplotlib để hiển thị ảnh phân đoạn và lưu kết quả phân đoạn vào thư mục đầu ra.

```
from matplotlib import pyplot as pylab
import cv2 # used for resize
import numpy as np
from model import Deeplabv3

# Load DeepLab model
deeplab_model = Deeplabv3()

# Input and output paths
pathIn = 'input' # Path for the input image
pathOut = 'output' # Output path for the segmented image

# Read the input image
img = pylab.imread(pathIn + r"D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\cycle.jpg")
w, h, _ = img.shape
ratio = 512. / np.max([w, h])

# Resize and preprocess the image
resized = cv2.resize(img, (int(ratio * h), int(ratio * w)))
resized = resized / 127.5 - 1.
pad_x = int(512 - resized.shape[0])
resized2 = np.pad(resized, ((0, pad_x), (0, 0), (0, 0)), mode='constant')

# Perform segmentation
res = deeplab_model.predict(np.expand_dims(resized2, 0))
labels = np.argmax(res.squeeze(), -1)

# Visualize and save the result
pylab.figure(figsize=(20, 20))
pylab.imshow(labels[:-pad_x], cmap='inferno')
pylab.axis('off')
pylab.colorbar()
pylab.show()
pylab.savefig(pathOut + "\\segmented.jpg", bbox_inches='tight', pad_inches=0)
pylab.close()
```

## Chương 12 Additional Problems in Image Processing

### Seam carving

#### 1. Content-aware image resizing with seam carving

Seam Carving là một thuật toán resize ảnh thông minh (content-aware resizing) được phát minh bởi Shai Avidan và Ariel Shamir vào năm 2007. Nó cho phép thay đổi kích thước hình ảnh theo chiều ngang hoặc chiều dọc mà không làm biến dạng hoặc mất đi các chi tiết quan trọng.

Seam carving hoạt động bằng cách thêm hoặc xóa các "đường seam" (seam) khỏi ảnh:

- Seam là một đường nối các pixel từ cạnh trên đến cạnh dưới (đối với seam dọc) hoặc từ cạnh trái đến cạnh phải (đối với seam ngang).
- Mỗi seam chỉ chứa một pixel trên mỗi hàng (hoặc mỗi cột), và các pixel trong seam phải liền kề nhau (theo đường chéo hoặc thẳng đứng).

Các seam có năng lượng thấp (ít thông tin quan trọng) sẽ bị xóa, giúp giảm kích thước ảnh mà không làm ảnh hưởng nhiều đến các chi tiết quan trọng.

#### Các bước thực hiện

##### Bước 1: Tính toán năng lượng của từng pixel

- **Năng lượng của pixel** đại diện cho mức độ quan trọng của nó.
- Thường sử dụng phương pháp **dual-gradient energy function**:
  - Dùng gradient của ảnh để tính mức độ thay đổi cường độ giữa các pixel lân cận.
  - Ví dụ: Sử dụng bộ lọc Sobel để tính gradient theo trục x và y, sau đó tính tổng như sau:

$$E(x, y) = \sqrt{(I_x)^2 + (I_y)^2}$$

- Ảnh năng lượng (energy map) cho thấy các vùng có năng lượng cao, thường là cạnh hoặc chi tiết nổi bật.

##### Bước 2: Tìm đường seam có năng lượng thấp nhất

- Xác định một **seam tối ưu** (đường đi từ cạnh trên đến cạnh dưới, hoặc từ trái sang phải) sao cho **tổng năng lượng trên seam là nhỏ nhất**.
- Quá trình này sử dụng **lập trình động (dynamic programming)**:

- Xây dựng bảng năng lượng tích lũy  $M(i,j)$  cho pixel tại hàng  $i$ , cột  $j$ , đại diện cho tổng năng lượng tối thiểu từ hàng đầu tiên đến pixel đó.

- Công thức:

$$M(i,j) = E(i,j) + \min(M(i-1, j-1), M(i-1, j), M(i-1, j+1))$$

- Sau khi tính toán, seam tối ưu được xác định bằng cách **truy ngược** từ hàng cuối lên hàng đầu.

### Bước 3: Xóa hoặc thêm seam

- Khi đã xác định được seam tối ưu:
  - **Xóa seam:** Loại bỏ các pixel thuộc seam đó khỏi ảnh, giảm kích thước ảnh.
  - **Thêm seam:** Nếu muốn mở rộng ảnh, seam tối ưu sẽ được sao chép và thêm vào ảnh.

```
from skimage import data, draw, transform, util, filters, color, io
import numpy as np
from matplotlib import pyplot as plt

# Đọc ảnh từ file
image = io.imread('D:\\Learn DL\\Xử lý ảnh\\BTL\\Sandipan_Dey_2018_Sample_Images\\images\\aero.jpg')
print(f"Image shape: {image.shape}") # (821, 616, 3)

# Chuyển ảnh sang định dạng float
image = util.img_as_float(image)

# Tính năng lượng ảnh sử dụng bộ lọc Sobel
energy_image = filters.sobel(color.rgb2gray(image))

# Hiển thị ảnh gốc
plt.figure(figsize=(20, 16))
plt.title('Original Image')
plt.imshow(image)
plt.axis('off')
plt.show()
```

Chuẩn bị ảnh và tính năng lượng là bước đầu tiên trong thuật toán seam carving.

Bản đồ năng lượng sẽ được sử dụng để xác định các **seam** ít quan trọng (có năng lượng thấp) nhằm giảm kích thước ảnh mà vẫn giữ được các chi tiết quan trọng.

```

from skimage import transform
from matplotlib import pyplot as plt

# Thay đổi kích thước ảnh
resized = transform.resize(image, (image.shape[0], image.shape[1] - 200), mode='reflect')
print(f"Resized image shape: {resized.shape}") # (821, 416, 3)

# Hiển thị ảnh đã thay đổi kích thước
plt.figure(figsize=(20, 11))
plt.title('Resized Image')
plt.imshow(resized)
plt.axis('off')
plt.show()

```

Thay đổi kích thước ảnh:

**transform.resize()**:

- Hàm thay đổi kích thước ảnh về một kích thước cụ thể.
- Tham số:
  - **image**: Ảnh đầu vào.
  - **(image.shape[0], image.shape[1] - 200)**:
    - Chiều cao **giữ nguyên**: image.shape[0].
    - Chiều rộng **giảm 200 pixel**: image.shape[1] - 200.
  - **mode='reflect'**: Khi cần nội suy để thu nhỏ hoặc phóng to, giá trị pixel ngoài vùng biên được tính bằng cách phản chiếu pixel lân cận.
- **Kết quả**:
  - Ảnh có kích thước mới là (821, 416, 3), tức là chiều rộng đã giảm từ 616 xuống còn 416, trong khi chiều cao (821) và số kênh màu (3) được giữ nguyên.

```

# Tính năng lượng ảnh sử dụng Sobel
def compute_energy(image):
    gray = color.rgb2gray(image) # Chuyển ảnh sang xám
    dx = sobel(gray, axis=1) # Gradient theo trục x
    dy = sobel(gray, axis=0) # Gradient theo trục y
    return np.hypot(dx, dy) # Tổng hợp năng lượng

# Tìm seam có năng lượng nhỏ nhất theo chiều dọc
def find_seam(energy):
    h, w = energy.shape
    cost = energy.copy()
    backtrack = np.zeros_like(cost, dtype=int) # Sửa lỗi tại đây

    for row in range(1, h):
        for col in range(w):
            min_cost = cost[row - 1, col]
            offset = 0

            if col > 0 and cost[row - 1, col - 1] < min_cost:
                min_cost = cost[row - 1, col - 1]
                offset = -1
            if col < w - 1 and cost[row - 1, col + 1] < min_cost:
                min_cost = cost[row - 1, col + 1]
                offset = 1

            cost[row, col] += min_cost
            backtrack[row, col] = offset

    seam = np.zeros(h, dtype=int)
    seam[-1] = np.argmax(cost[-1])
    for row in range(h - 2, -1, -1):
        seam[row] = seam[row + 1] + backtrack[row + 1, seam[row + 1]]

    return seam

# Xóa seam từ ảnh
def remove_seam(image, seam):
    h, w, c = image.shape
    output = np.zeros((h, w - 1, c))
    for row in range(h):
        col = seam[row]
        output[row, :, :] = np.delete(image[row, :, :], col, axis=0)
    return output

# Seam Carving
def seam_carving(image, num_seams):
    for _ in range(num_seams):
        energy = compute_energy(image)
        seam = find_seam(energy)
        image = remove_seam(image, seam)
    return image

```

Hàm tính năng lượng ảnh:

```

# Tính năng lượng ảnh sử dụng Sobel
def compute_energy(image):
    gray = color.rgb2gray(image) # Chuyển ảnh sang xám
    dx = sobel(gray, axis=1) # Gradient theo trục x
    dy = sobel(gray, axis=0) # Gradient theo trục y
    return np.hypot(dx, dy) # Tổng hợp năng lượng

```

Xác định năng lượng (tâm quan trọng) của từng pixel trong ảnh.

Các bước:

1. **Chuyển sang ảnh xám:** Dùng hàm color.rgb2gray để giảm từ 3 kênh màu (RGB) xuống 1 kênh grayscale.
2. **Tính gradient:**
  - o Sử dụng bộ lọc Sobel để tính gradient theo trục x (độ thay đổi theo chiều ngang) và trục y (độ thay đổi theo chiều dọc).
3. **Kết hợp năng lượng:** Tính tổng hợp gradient bằng **hypotenuse**:  $\sqrt{dx^2 + dy^2}$   
Pixel có gradient cao (biên, cạnh) sẽ có năng lượng cao.

Tìm đường seam năng lượng thấp nhất:

```
# Tìm seam có năng lượng nhỏ nhất theo chiều dọc
def find_seam(energy):
    h, w = energy.shape
    cost = energy.copy()
    backtrack = np.zeros_like(cost, dtype=int) # Sửa lỗi tại đây

    for row in range(1, h):
        for col in range(w):
            min_cost = cost[row - 1, col]
            offset = 0

            if col > 0 and cost[row - 1, col - 1] < min_cost:
                min_cost = cost[row - 1, col - 1]
                offset = -1
            if col < w - 1 and cost[row - 1, col + 1] < min_cost:
                min_cost = cost[row - 1, col + 1]
                offset = 1

            cost[row, col] += min_cost
            backtrack[row, col] = offset

    seam = np.zeros(h, dtype=int)
    seam[-1] = np.argmin(cost[-1])
    for row in range(h - 2, -1, -1):
        seam[row] = seam[row + 1] + backtrack[row + 1, seam[row + 1]]

    return seam
```

Tìm đường seam có tổng năng lượng nhỏ nhất từ trên xuống dưới.

Các bước:

1. **Khởi tạo:**
  - o **cost:** Ma trận lưu tổng năng lượng của các seam kết thúc tại mỗi pixel.

- o **backtrack**: Ma trận lưu hướng đi ngược (về hàng trên) để tái tạo seam.

## 2. Tính toán chi phí (cost):

- o Tại mỗi pixel, cộng năng lượng của nó với chi phí thấp nhất từ hàng trước đó (bao gồm pixel trái, trên, và phải).

## 3. Truy vết seam:

- o Bắt đầu từ pixel có năng lượng nhỏ nhất ở hàng cuối.
- o Dùng ma trận backtrack để truy vết seam lên hàng đầu.

Xóa seam:

```
# Xóa seam từ ảnh
def remove_seam(image, seam):
    h, w, c = image.shape
    output = np.zeros((h, w - 1, c))
    for row in range(h):
        col = seam[row]
        output[row, :, :] = np.delete(image[row, :, :], col, axis=0)
    return output
```

Xóa một đường seam ra khỏi ảnh.

**Cách hoạt động:**

1. Tạo mảng kết quả với chiều rộng nhỏ hơn 1 pixel so với ảnh gốc.
2. Tại mỗi hàng, xóa pixel thuộc seam (chỉ số cột được lưu trong seam).

Hàm Seam Carving:

```
# Seam Carving
def seam_carving(image, num_seams):
    for _ in range(num_seams):
        energy = compute_energy(image)
        seam = find_seam(energy)
        image = remove_seam(image, seam)
    return image
```

Lặp lại quá trình xóa seam để giảm chiều rộng ảnh.

Hiển thị kết quả:

Original Image



Resized Image



Resized using Seam Carving

