cs512 f24 - assignment 2 (filtering)

Due by: 9/29/2024

General Instructions:

In this assignment, you will implement various image filtering techniques, including noise addition, convolution filters, Gaussian smoothing, and edge detection. You will also explore gradient-based techniques and Gaussian pyramids. You will use Python, NumPy, and OpenCV for image processing tasks.

- For all tasks, use Python libraries such as NumPy and OpenCV.
- Display images at each step where relevant and comment on the observations.
- For convolution tasks, implement the filters manually before relying on library functions to understand the mathematical operations involved.

Submission instructions:

- Follow the submission instructions of assignment 0.
- Submit a fully executed Python notebook with no gaps in execution. To receive full credit, please ensure the following:
  - The notebook includes all cell outputs and contains no error messages.
  - It is easy to match each problem with its corresponding code solution using clear markdown or comments (e.g., “Problem 2”).
  - Re-run your notebook before submitting to ensure that cells are numbered sequentially, starting at [1].

Questions:

1. Add Gaussian Noise and Compute SNR

- **Task**: Load a grayscale image and generate 10 noisy versions by adding Gaussian noise with a fixed standard deviation and zero mean. Compute the noise variance by calculating the pixel-wise standard deviation across the 10 noisy images (without directly accessing the noise itself). Compute the signal power as the variance (square of the standard deviation) of the original grayscale image. Finally, calculate the SNR in dB using the ratio of the signal power to the noise power.
- **Hints:**
  - Use `cv2.imread()` to load the image as grayscale using the `cv2.IMREAD_GRAYSCALE` argument.
  - To generate Gaussian noise, you can use `numpy.random.normal()`.
  - To clip the noisy image within the valid pixel range (0 to 255), use `numpy.clip()`.
  - The variance can be computed with `numpy.var()`.
  - For logarithmic functions, use `numpy.log10()` to calculate the SNR in decibels.
- **Starter code:**

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image and convert to grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
image = image.astype(np.float32)  # Convert to float for precise calculations

# Parameters for Gaussian noise
std_dev = 20  # Known standard deviation for Gaussian noise
mean = 0  # Mean of the Gaussian noise
num_noisy_images = 10  # Number of noisy images to generate

# Generate 10 noisy versions of the grayscale image
noisy_images = []
for i in range(num_noisy_images):
    noise = np.random.normal(mean, std_dev, image.shape).astype(np.float32)
    noisy_image = np.clip(image + noise, 0, 255)  # Clip values to be in valid range
    noisy_images.append(noisy_image)

# Compute the noise power as the mean of the noise variance

# Compute the signal power as the variance of the original image

# Compute the SNR in decibels (dB)
```

2. Implement a Convolution Filter for Smoothing

- **Task**: Load a grayscale image. Implement a basic 3x3 convolution filter. Pad the image with zeros so that the convolution result has the same dimensions as the input. Define a basic 3x3 smoothing filter. Apply the filter to a grayscale image using your function and OpenCV's function, and compare the execution time. Display the original and smoothed images. Explain the performance difference between your implementation and OpenCV's convolution.
- **Hints:**
  - Implement the manual convolution by creating a sliding window over the image. Pad the image with `numpy.pad()` before the convolution.
  - For OpenCV's convolution use `cv2.filter2D` with `borderType=cv2.BORDER_CONSTANT`.
  - To visualize the results, use `matplotlib.pyplot.imshow()`.
- **Starter code:**

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
import time

# Function for manual convolution using zero padding
def manual_convolution(image, kernel):
    return output_image

# Load the image and convert it to grayscale
image = cv2.imread('your_image_path.jpg', cv2.IMREAD_GRAYSCALE)
```

```
# Define a 3x3 averaging filter (kernel)
kernel = np.ones((3, 3)) / 9.0

# Time the convolution implementations
start_manual = time.time()
smoothed_image_manual = manual_convolution(image, kernel)
end_manual = time.time()
start_opencv = time.time()
smoothed_image_opencv = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_CONSTANT)
end_opencv = time.time()
print(f"Manual convolution time: {end_manual - start_manual:.5f} seconds")
print(f"OpenCV convolution time (zero padding): {end_opencv - start_opencv:.5f}
seconds")

# Display the original, manually convolved, and OpenCV-convolved images
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Grayscale Image")
plt.subplot(1, 3, 2)
plt.imshow(smoothed_image_manual, cmap='gray')
plt.title("Manually Convolved Image")
plt.subplot(1, 3, 3)
plt.imshow(smoothed_image_opencv, cmap='gray')
plt.title("OpenCV Convolved Image (Zero Padding)")
plt.show()
```

3. Convolution with Stride

- **Task**: Implement a 3x3 convolution with a stride on a grayscale image. Experiment with different stride values and observe the differences in output.
- **Hints:**
  - For convolution with a stride, implement a loop with a step size that corresponds to the stride value.
  - Pad the image with `numpy.pad()` before the convolution.
- **Starter code:**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import time

# Function for convolution with stride and zero padding, accepting kernel as
argument
def convolution_with_stride(image, kernel, stride=2):
    return output_image

# Load the image and convert it to grayscale
image = cv2.imread('your_image_path.jpg', cv2.IMREAD_GRAYSCALE)

# Define a 3x3 averaging filter (kernel) as an argument
kernel = np.ones((3, 3)) / 9.0

# Apply the convolution
smoothed_image_stride = convolution_with_stride(image, kernel, stride=2)
```

```
# Display the original and convolved images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Grayscale Image")
plt.subplot(1, 2, 2)
plt.imshow(smoothed_image_stride, cmap='gray')
plt.title("Convolved Image with Stride and Zero Padding")
plt.show()
```

4. Compute and apply a Gaussian Smoothing Filter

- **Task**: Compute an arbitrary size 2D Gaussian filter using the 2D Gaussian function. Print the Gaussian kernel. Apply the kernel to a grayscale image and display the result. Determine what the maximum allowed standard deviation (stdv) should be for a 5x5 filter, and explain your reasoning.
- **Hints:**
  - Write a `gaussian_kernel(size, sigma)` function to generate the Gaussian kernel.
  - Apply the kernel using `cv2.filter2D`.
- **Starter code:**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Function to generate a 2D Gaussian kernel
def gaussian_kernel(size, sigma):
    return kernel

# Load the image and convert to grayscale
image = cv2.imread('your_image_path.jpg', cv2.IMREAD_GRAYSCALE)

# Generate and display the 2D Gaussian kernel
kernel_size = 5  # 5x5 kernel
sigma = 1.0  # Standard deviation for the Gaussian
gaussian_kernel_2d = gaussian_kernel(kernel_size, sigma)
print("Gaussian Kernel (5x5):\n", gaussian_kernel_2d)

# Apply the Gaussian filter to the image using OpenCV's filter2D
smoothed_image = cv2.filter2D(image, -1, gaussian_kernel_2d)

# Display the original and smoothed images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Grayscale Image")
plt.subplot(1, 2, 2)
plt.imshow(smoothed_image, cmap='gray')
plt.title("Smoothed Image with 2D Gaussian Filter")
plt.show()
```

5. Gaussian Pyramid Construction

- **Task**: Compute a Gaussian pyramid for an input image and display the pyramid layers. Scale the pyramid layers to the size of the original image for comparison. Report what you observe in the successive pyramid layers..
- **Hints:**
  - Use `cv2.pyrDown()` to create each successive level of the Gaussian pyramid.
  - Use `cv2.resize()` to scale the pyramid layers back to the original image size for visualization.
  - To display multiple images side by side, use `matplotlib.pyplot`.
- **Starter code:**

```
# Compute Gaussian pyramid
def gaussian_pyramid(image, levels=3):
    return pyramid

# Display pyramid with resized layers
pyramid_layers = gaussian_pyramid(image)
resized_pyramid_layers = [cv2.resize(layer, (image.shape[1], image.shape[0])) for
layer in pyramid_layers]

# Display results
plt.figure(figsize=(15, 5))
for i, layer in enumerate(resized_pyramid_layers):
    plt.subplot(1, len(resized_pyramid_layers), i + 1)
    plt.imshow(layer)
    plt.title(f"Pyramid Level {i}")
plt.show()
```

6. Image Gradients and Histogram of Gradient Directions

- **Task**: Compute and display the image gradients for a given grayscale test image. Display gradients only above a certain magnitude. Then, compute and plot a histogram of the gradient directions. Use simple test images (e.g., square or circle) to verify the correctness of your implementation.
- **Hints:**
  - Use the Sobel operator with `cv2.Sobel()` to compute the gradients in the x and y directions.
  - Calculate the gradient magnitude using `numpy.sqrt()`.
  - Compute the gradient direction using `numpy.arctan2()`.
  - Use OpenCV's `cv2.arrowedLine()` to draw gradient vectors (arrows) on the image.
  - Use `matplotlib.pyplot.hist()` to plot the histogram of gradient directions.
- **Starter code:**

```
import cv2
import numpy as np

# Load the image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Compute gradients
grad_x =
```

```
grad_y =
magnitude =
direction =

# Draw gradient vectors
image_with_vectors = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
threshold = 100
strong_magnitudes = magnitude > threshold
y_coords, x_coords = np.where(strong_magnitudes)

# Draw gradient vectors (arrows) on the image
for y, x in zip(y_coords, x_coords):
    dx = int(grad_x[y, x] / 10)
    dy = int(grad_y[y, x] / 10)
    start_point = (x, y)
    end_point = (x + dx, y + dy)
    cv2.arrowedLine(image_with_vectors, start_point, end_point, (0, 0, 255), 1,
tipLength=0.3)

# Display the results
cv2.imshow("Image with Gradient Vectors", image_with_vectors)
plt.hist(direction.ravel(), bins=30)
plt.title("Histogram of Gradient Directions")
plt.show()
```

7. Gaussian Derivatives Gradients

- **Task**: Implement separable Gaussian derivative convolution filters to compute the x and y derivatives of an image. For x derivatives, first, convolve the image with a horizontal Gaussian derivative filter in the x direction, and then convolve it with a vertical Gaussian smoothing filter in the y direction. For y derivatives, first, convolve the image with a vertical Gaussian derivative filter in the y direction, and then convolve it with a horizontal Gaussian smoothing filter in the x direction. Display the x and y derivatives of the image as two separate images.
- **Hints:**
  - Define `gaussian_1d` and `gaussian_derivative_1d` functions.
  - Use NumPy's `reshape` function to reshape them to be horizontal or vertical.
  - Use `cv2.filter2D()` to apply these filters to the image.
- **Starter code:**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Function to generate a 1D Gaussian kernel
def gaussian_1d(size, sigma):
    return kernel

# Function to generate a 1D Gaussian derivative kernel
def gaussian_derivative_1d(size, sigma):
    return kernel

# Define parameters for Gaussian filters
kernel_size = 5
sigma = 1.0
```

```
# Compute 1D filters
gaussian_x =
gaussian_y =
gaussian_derivative_x =
gaussian_derivative_y =

# Load the image and convert it to grayscale
image = cv2.imread('your_image_path.jpg', cv2.IMREAD_GRAYSCALE)

# Detect X derivative (Separable convolution)

# Detect Y derivative (Separable convolution)

# Display the results
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Grayscale Image")
plt.subplot(1, 3, 2)
plt.imshow(image_x_derivative, cmap='gray')
plt.title("X Derivative (Gaussian)")
plt.subplot(1, 3, 3)
plt.imshow(image_y_derivative, cmap='gray')
plt.title("Y Derivative (Gaussian)")
plt.show()
```

8. Laplacian of Gaussian (LoG) Filtering

- **Task**: Filter an image using a Laplacian of Gaussian (LoG) filter. Display the zero crossings of the LoG to detect edges. Demonstrate the correctness of the method with test images.
- **Hints:**
  - Compute a Laplacian of Gaussian filter by computing the Laplacian of a Gaussian function and samplig it, or by convolving a discrete Gaussian filter with a discrete 3x3 Laplacian filter.
  - Use `cv2.filter2D()` to apply these filters to the image.
  - To detect zero-crossings in the LOG image, convert it to a binary image (0 for non-positive and 1 for positive values), then detect transitions between 0 and 1.
- **Starter code**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage

# Function to create a Laplacian of Gaussian filter
def laplacian_of_gaussian(size, sigma):
    return kernel

# Apply the LoG filter to the image
def apply_log(image, size, sigma):
    log_filter = laplacian_of_gaussian(size, sigma)
    log_filtered_image = cv2.filter2D(image, -1, log_filter)
    return log_filtered_image
```

```python
# Detect zero crossings
def detect_zero_crossings(log_image):
    # Define the 1D filters for horizontal and vertical derivatives
    horizontal_filter = np.array([[-1, 1]])  # Horizontal 1x2 filter
    vertical_filter = np.array([[-1], [1]])  # Vertical 2x1 filter

    # Convolve the image with the horizontal and vertical filters
    horizontal_edges = cv2.filter2D(log_image, -1, horizontal_filter)
    vertical_edges = cv2.filter2D(log_image, -1, vertical_filter)

    # Detect zero crossings by checking for sign changes between adjacent pixels
    zero_crossings = np.zeros_like(log_image)
    zero_crossings[(horizontal_edges > 0) & (horizontal_edges < 0)] = 1
    zero_crossings[(vertical_edges > 0) & (vertical_edges < 0)] = 1
    zero_crossings[(horizontal_edges == 0) & (vertical_edges == 0)] = 1

    return zero_crossings

# Load the image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Laplacian of Gaussian (LoG)
size = 5  # Filter size
sigma = 1.0  # Standard deviation
log_filtered_image = apply_log(image, size, sigma)
zero_crossings = detect_zero_crossings(log_filtered_image)

# Display the original image, LoG-filtered image, and zero crossings
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.subplot(1, 3, 2)
plt.imshow(log_filtered_image, cmap='gray')
plt.title("LoG Filtered Image")
plt.subplot(1, 3, 3)
plt.imshow(zero_crossings, cmap='gray')
plt.title("Zero Crossings of LoG")
plt.show()
```