Assignment 2: Canny Edge Detector

Due Date: 18 October 2024, 23:59

Extension granted until: 21 October 2024, 23:59

Introduction

In this assignment you will implemented the Canny Edge Detector, which is a seminal work in Computer Vision and particular in edge detection. You will get a first glance of important techniques in CV, such as filtering through the use of 2D convolution. This will enable you to reduce the "noise" of an image and to calculate its image gradient. By using the latter, you will extract the important edges from the image, following the steps of non-maximum suppression and hysteresis thresholding.

Preliminary step

Load the image *building.jpg* using OpenCV. For all checks that are needed for the following questions, you should use the `assert()` function. All intermediate and final results should be plotted using the `matplotlib` package.

1 2D Convolution (20 points)

In this question you need to write a function that implements a generic 2D convolution of 2D kernel over a 2D array (or 3D for multi-channel images). Your function should use the following function signature:

```
def convolution_2D(arr, kernel, border_type):
    """
       Calculate the 2D convolution kernel*arr
    :param arr: numpy.array(float), input array
    :param kernel: numpy.array(float), convolution kernel of nxn size (only odd dimensions are allowed)
    :param border_type: int, padding method (OpenCV)
    :return:
        conv_arr: numpy.array(float), convolution output
    """
```

First, you should check if the kernel is of square odd size and and then flip the kernel, as defined by the convolution operation. Next, you need to pad the input array according to the kernel size and then the padded array should be convolved with the kernel. The convolution should take place for all channels of the input array. The output array should have the same dimensions as the input array. You cannot use any OpenCV function (or any other library) to convolve the input array with the convolution kernel. For padding, you can use the OpenCV's corresponding function.

2 Noise Reduction (10 points)

In order to reduce noise on the input image, you need to first create a function with the following function signature:

```
def gaussian_kernel_2D(ksize, sigma):
    """
        Calculate a 2D Gaussian kernel
    :param ksize: int, size of 2d kernel, always needs to be an odd number
    :param sigma: float, standard deviation of gaussian
    :return:
        kernel: numpy.array(float), ksize x ksize gaussian kernel with mean=0
    """
```

You will need to check that the input size is an odd positive non-zero number and that sigma in a non-zero positive number. The function should return a 2D Gaussian kernel, where all of its elements sum to 1. You cannot use any OpenCV function (or any other library) to create the Gaussian kernel.

For reducing the image's noise you need to convolve the Gaussian kernel with the input image, using your 2D convolution implementation. Plot your results in a 1×2 grid (original and blurred image).

3 Image Gradient (20 points)

For this step, first you need to implement two functions that convolve the image using the Sobel operators, with the following function signatures:

```
def sobel_x(arr):
    """
        Calculate the 1st order partial derivatives along x-axis
    :param arr: numpy.array(float), input image
    :return:
        dx: numpy.array(float), output partial derivative
    """

def sobel_y(arr):
    """
        Calculate 1st the order partial derivatives along y-axis
    :param arr: numpy.array(float), input image
    :return:
        dy: numpy.array(float), output partial derivatives
    """
```

Each function will use the corresponding Sobel kernel to convolve the image (in grayscale), using your 2D convolution implementation. The output images should be normalized in the range $(-1, +1]$ or $[-1, +1)$, thus retaining the negative and positive partial derivatives (hint: *use only the absolute maximum value of each partial derivative image*). Plot the I_x and I_y images in a 1×2 grid.

Finally, calculate the image gradient magnitude (renomralize to $[0, 1]$) and the direction (in degrees) and plot them in a 1×2 grid. For the direction image, first represent it in the HSV format (magnitude and direction) and then convert it to RGB for visualization purposes. Keep in mind that OpenCV's HSV color space supports hue range in [0,179], saturation range in [0,255] and value range in [0,255].

4 Non-maximum Suppression (20 points)

In this question you need to implement the non-maximum suppression step of the Canny Edge Detector algorithm. You need to create a function with the following function signature:

```
def non_maximum_suppression(arr_mag, arr_dir):
    """
        Find all local maxima along image gradient direction
    :param arr_mag: numpy.array(float), input image gradient magnitude
    :param arr_dir: numpy.array(float), input image gradient direction
    :return:
        arr_local_maxima: numpy.array(float)
    """
```

First you should bin the directions in *8 principal directions* and then find local maxima according to the image gradient magnitude and direction. Plot the resulting thinned edges.

5 Hysteresis Thresholding (30 points)

As the final step of Canny algorithm, you need to implement the hysteresis thresholding procedure using the following function signature:

```
def hysteresis_thresholding(arr, low_ratio, high_ratio):
    """
        Use the low and high ratios to threshold the non-maximum suppression image and then link
        non-weak edges
    :param arr: numpy.array(float), input non-maximum suppression image
    :param low_ratio: float, low threshold ratio
    :param high_ratio: float, high threshold ratio
    :return:
        edges: numpy.array(uint8), output edges
    """
```

This function should first calculate the min and max thresholds using the low and high ratios w.r.t. the non-maximum suppression result. Next, the thresholds need to be applied to the image. Finally, for pixels that are between the two thresholds (non-weak edges) an iterative process should follow, until most of them are connected to strong ones, based on their 1-ring neighborhood proximity to strong edges.

Plot the resulted edges from your implementation and from OpenCV's Canny Edge Detector implementation, in a 1×2 grid. For OpenCV's results you need to use:

```
edges = cv2.Canny(image=blurred_image, threshold1=100, threshold2=200)
```

where the `blurred image` should be the same as the one you used for your implementation.

6 Extra Credits: "Flattened" Convolution (10 points)

For this **extra credits** question, implement the `convolution_2D()` function (Question 1), by using the "flattened" version of the convolution operation.

7 Instructions

- The assignment is due by **Friday October 18th, at 23:59** (see the course's GitHub repository for more details regarding the late assignment policy).
- The assignment should be submitted **only** through Moodle. Compress all the needed **source code** files, e.g. all *.py files into a .zip file and name it as follows "**Lab_Assignment_2_<UC-ID-Number>.zip**".
- For further questions you can create an issue on the course's GitHub repository.
- All lab assignments should be conducted **independently**.
- The Moss system will be used to detect code similarity.
- Plagiarism and/or cheating will be punished with a grade of zero for the current assignment. A second offense will carry additional penalties.