IT416 Computer Vision

LAB ASSIGNMNET #1

Submitted by:

Sreenivasapur Adnan Raqeeb

201IT162

Code Link: [Github Link with code implemented in python]

I. Zooming Methods

A. Pixel Replication or Nearest Neighbour

Since we can only replicate the neighbouring pixels when using pixel replication, it is also referred to as nearest neighbour interpolation. With the pixel replication method, the input image's already-given or existing pixels are used to create new ones. The input image is zoomed in and each pixel is repeated n times in both row and column directions.

Note: We use this method in two steps: row-wise and column-wise.

Pixel Replication

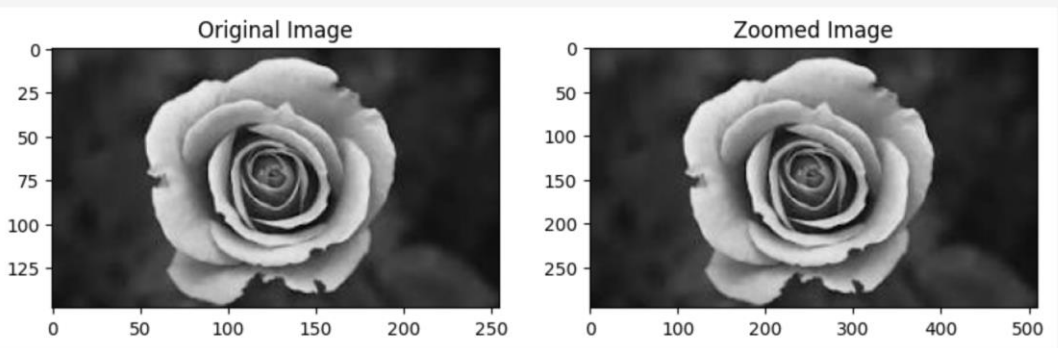
NewImage=(InputRows)(zoomFactor),(InputCols)(zoomFactor)

```
def pixel_replication_zoom(original_image, zoom_factor):

    original_height, original_width, _ = original_image.shape
    new_width = int(original_width * zoom_factor)
    new_height = int(original_height * zoom_factor)
    zoomed_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)

    # Apply pixel replication zooming
    for x in range(new_width):
        for y in range(new_height):
            original_x = int(x / zoom_factor)
            original_y = int(y / zoom_factor)

            # Get the pixel value from the original image
            pixel = original_image[original_y, original_x]

            # Set the pixel value in the zoomed image
            zoomed_image[y, x] = pixel

    return zoomed_image
```

✓ 0.0s

```
# Example for Pixel Replication
image_path = "Image BW.jpg"
zoom_factor = 2
```

✓ 0.0s

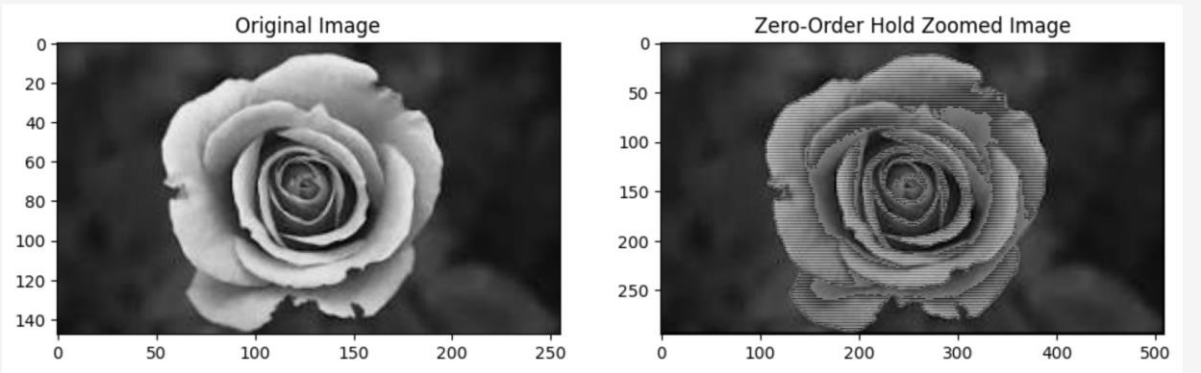


B. Zero-Order Hold

We can only zoom twice using this method, which is why the zero-order hold is also called zoom twice. By using the zero-order hold method, we select two neighbouring elements from a row. These components are then added. We split the outcome in half after the addition. Next, the number that is produced is positioned in between the neighbouring items that we previously selected.

Note: We apply this method row-wise first and then column-wise.

```
def zero_order_hold_zoom_image(image):
    # Get the image dimensions
    height, width, channels = image.shape

    # Create a new image with increased dimensions
    zoomed_image = np.zeros((2 * height - 1, 2 * width - 1, channels), dtype=image.dtype)

    # Apply zero-order hold row-wise
    for y in range(height-1):
        for x in range(width - 1):
            zoomed_image[2*y, 2*x, :] = image[y, x, :]

        for x in range(width - 1):
            zoomed_image[2*y, 2*x+1, :] = image[y, x, :]


    # Apply zero-order hold column-wise
    for y in range(height - 1):
        for x in range(2 * width - 1):
            zoomed_image[2*y + 1, x, :] = (zoomed_image[2*y, x, :] + zoomed_image[2*y + 2, x, :]) // 2


    # # Copy the last row as is
    # zoomed_image[-1, :, :] = image[-1, :, :]

    return zoomed_image
```

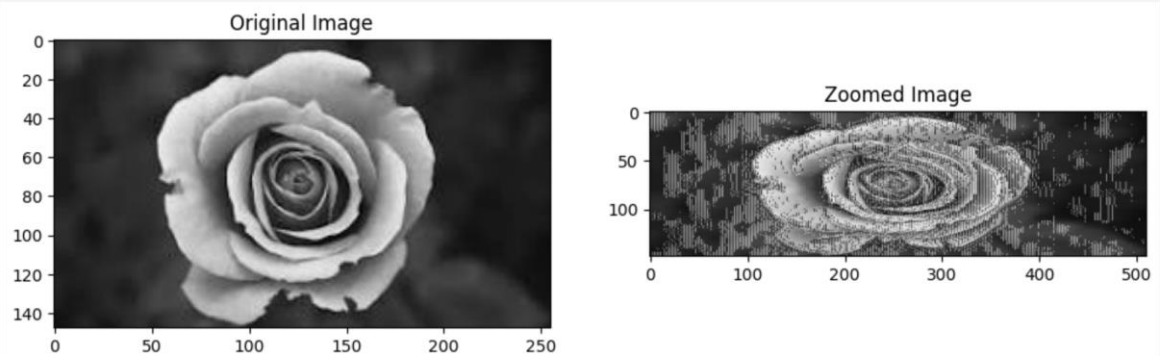


C. K-Times Zooming Method

The most popular and accurate technique is K-times zooming, which addresses the drawbacks of both zero-order hold zooming and pixel replication. This approach will work as follows

- The smaller of two adjacent pixels is subtracted from the larger one. Another name for the outcome of subtraction is OP.
- Next, we add the result to the smaller pixel that was previously selected, divide the output (OP) by k (the zooming factor represented by the alphabet k in the name), and place this value between the neighbouring pixels.
- Once more, add the value (OP) to the value obtained in the previous step, then place the new value next to the value that was previously put.
- This process will be repeated until k−1 values are successfully added to the current image.

```
def k_times_zoom(image_path, k):

    original_image = cv2.imread(image_path)
    image_array = np.array(original_image)
    height, width, _ = image_array.shape
    zoomed_image = np.zeros((height, width * 2, 3), dtype=np.uint8)

    for i in range(height):
        for j in range(0, width * 2 - 1, 2):
            pixel1 = image_array[i, j // 2]
            pixel2 = image_array[i, min(j // 2 + 1, width - 1)]

            op = np.abs(np.subtract(pixel1, pixel2))
            op = op.astype(np.float64) / k

            zoomed_pixel = (pixel1.astype(np.float64) + op).astype(np.uint8)
            zoomed_image[i, j] = pixel1
            zoomed_image[i, j + 1] = zoomed_pixel
```



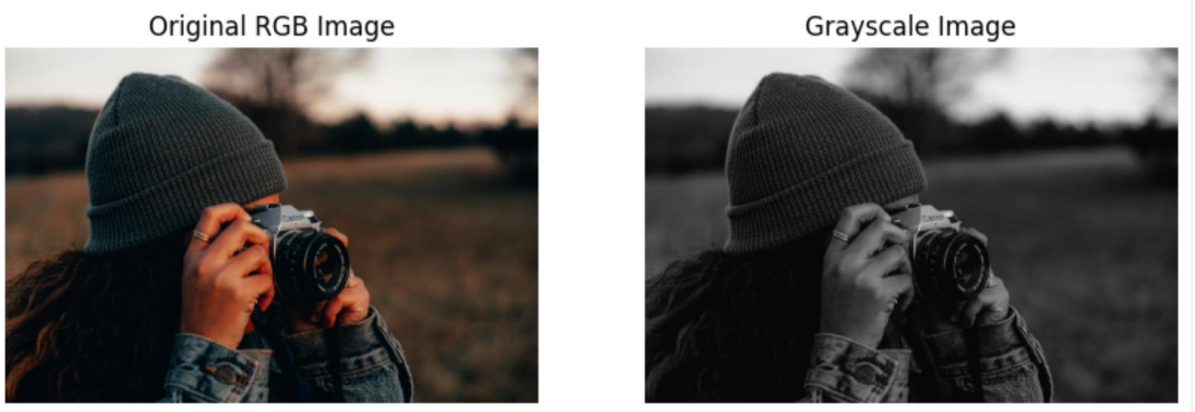
II. RGB to Grayscale conversion

The luminosity method is the best method to convert RGB image to grayscale image

It is expected that green will contribute more to the final value and blue will contribute less. Following a number of tests and a more thorough investigation, scientists came to the following conclusion:

$$grayscale = 0.3 * R + 0.59 * G + 0.11 * B$$

```
def rgbToGray(img):
    new_img = np.zeros_like(img)
    shape = img.shape
    for i in range(shape[0]):
        for j in range(shape[1]):
            r = img[i][j][0]
            g = img[i][j][1]
            b = img[i][j][2]
            grayscale = 0.299 * r + 0.587 * g + 0.114 * b
            for k in range(3):
                new_img[i][j][k] = grayscale

    return new_img.astype(np.uint8)
```



III. Brightness Manipulation

If we take an 8-bit RGB color space and assume that the histogram has two sides, one being (0, 0, 0) [black] and the other (255, 255, 255) [white], we can make the image darker or lighter by shifting its contents left or right. Moving the histogram to the right increases the frequency of values in the image that are closer to white, making it brighter; moving it to the left brings more values towards black, darkening the image.

```
def boundedPixelValue(color, brightnessFactor):
    scaledValue = float(color * (1 + brightnessFactor))
    if scaledValue < 0:
        return 0
    elif scaledValue > 255:
        return 255

    return int(scaledValue)

im = Image.open("Image.jpg")
out = Image.new('RGB', im.size, 0xffffff)

#constant brightness factor of 0.5
brightnessFactor = 0.5

width, height = im.size
for x in range(width):
    for y in range(height):
        r, g, b = im.getpixel((x, y))

        updatedR = boundedPixelValue(r, brightnessFactor)
        updatedG = boundedPixelValue(g, brightnessFactor)
        updatedB = boundedPixelValue(b, brightnessFactor)

        out.putpixel((x, y), (updatedR, updatedG, updatedB))

processed_image = out.copy()
```