Image Processing Report

Phan Lâm Anh

August 4, 2023

1 Introduction

This report provides an overview of various image processing functions implemented in Python using the PIL (Python Imaging Library), NumPy, and Matplotlib libraries. Each function is described in detail below.

2 Function: change_brightness

```
def change_brightness(image, value):
    image = image.astype("int16")
    image = image + value
    image = np.clip(image, 0, 255)
    return image.astype("uint8")
```

2.1 Description

The `change_brightness` function adjusts the brightness of an image by adding a specified value to each pixel's intensity. The function takes an image and a value as inputs. The image is first converted to a signed integer type (`int16`) to allow for negative values. The specified value is then added to each pixel's intensity. The resulting image is clipped to ensure that all pixel values are within the valid range of 0 to 255. Finally, the image is converted back to an unsigned integer type (`uint8`) and returned.

2.2 Idea

The code snippet performs the following steps to adjust the brightness of an image: Convert the image to the int16 data type: The line image = image.astype("int16") converts the image data type to int16. This conversion allows for the addition of both positive and negative values to the image. Add the specified value to the image: The line image = image + value adds the specified value to each pixel's intensity in the image. This step increases or decreases the brightness of the image based on the value provided. Clip the

image pixel values to the valid range: The line image = np.clip(image, 0, 255) ensures that all pixel values in the image are within the valid range of 0 to 255. This step prevents any pixel values from exceeding the valid range. Convert the image back to the uint8 data type: The line return image.astype("uint8") converts the image data type back to uint8 before returning the adjusted image. This conversion ensures that the image is returned in the expected data type.

Figure 1: Brightness 100

3 Function: change_contrast

```
def change_contrast(image, contrast):
    # Calculate the contrast factor
    factor = (259 * (contrast + 255)) / (255 * (259 - contrast))

    # Apply the contrast adjustment using a Look Up Table (LUT)
    lut = np.arange(256) * factor
    lut = np.clip(lut, 0, 255).astype("uint8")
    contrast_image = lut[image]

    return contrast_image
```

3.1 Description

The `change_contrast` function adjusts the contrast of an image by applying a contrast factor to each pixel's intensity. The function takes an image and a contrast value as inputs.

3.2 Idea

1. Calculate the contrast factor:
   - The contrast factor is calculated using the formula: $\frac{259 \times (contrast + 255)}{255 \times (259 - contrast)}$.
   - This formula ensures that the contrast factor is within the range of -255 to \$255.
2. Apply the contrast adjustment using a Look Up Table (LUT):
   - A Look Up Table (LUT) is created using `np.arange(256)` $\times factor, where$`factor`$is the contrast facto$ `using np.clip(lut, 0, 255).`
   - `The LUT is converted to an unsigned integer type (uint8) using astype("uint8").`
3. Apply the contrast adjustment to the image:
   - The LUT is used to map each pixel's intensity value to its corresponding adjusted value.
   - The contrast-adjusted image is obtained by indexing the LUT with the original image: `contrast_image = lut[image]`.
4. Return the contrast-adjusted image.

Figure 2: Contrast 100

4 Function: flip_image

```
def flip_image(image, direction):
    if direction == "horizontal":
        return np.flip(image, axis=1)
    elif direction == "vertical":
        return np.flip(image, axis=0)
    else:
        return image
```

4.0.1 Description

The `flip_image` function flips an image either horizontally or vertically based on the specified direction. The function takes an image and a direction as inputs.

4.0.2 Idea

1. Check the specified direction:
    - If the direction is "horizontal", the function uses `np.flip(image, axis=1)` to flip the image horizontally.
    - If the direction is "vertical", the function uses `np.flip(image, axis=0)` to flip the image vertically.
    - If the direction is neither "horizontal" nor "vertical", the function returns the original image.

Figure 3: Flip horizontal

5 Function: rgb_to_grayscale

```
def rgb_to_grayscale(image):
    grayscale_image = np.mean(image, axis=2).astype(np.uint8)

    return grayscale_image
```

5.0.1 Description

The `rgb_to_grayscale` function converts an RGB image to grayscale. The function takes an RGB image as input and returns the corresponding grayscale image.

5.0.2 Idea

1. Compute the grayscale image:
   - The function uses `np.mean(image, axis=2)` to calculate the mean across the color channels (axis=2) of the RGB image.
   - This operation collapses the three color channels into a single channel, resulting in a grayscale image.
2. Convert the grayscale image to the `uint8` data type:
   - The grayscale image is then cast to the `uint8` data type using `astype(np.uint8)`.
   - This step ensures that the pixel values of the grayscale image are within the range of 0 to 255.
3. Return the grayscale image.

Figure 4: Grayscale

6 Function: rgb_to_sepia

```
def rgb_to_sepia(img_2d: np.ndarray) -> np.ndarray:
    h, w, c = img_2d.shape
    new_img_1d = img_2d[:, :, :3].reshape((h * w, 3)).astype(float)
    formula = [[0.393, 0.349, 0.272],
    [0.769, 0.686, 0.534],
    [0.189, 0.168, 0.131]]
    new_img_1d = np.clip(np.matmul(new_img_1d, formula), 0, 255)

    return new_img_1d.reshape((h, w, 3)).round(0).astype(np.uint8)
```

6.0.1 Description

The `rgb_to_sepia` function converts an RGB image to a sepia-toned image. The function takes a 2D numpy array representing the RGB image as input and returns a new numpy array representing the sepia-toned image.

6.0.2 Idea

1. Obtain the dimensions of the input image:
    - The line `h, w, c = img_2d.shape` extracts the height (`h`), width (`w`), and number of color channels (`c`) from the input image.
2. Reshape the input image:
    - The line `new_img_1d = img_2d[:, :, :3].reshape((h * w, 3)).astype(float)` reshapes the input image to a 1D array of shape (`h * w`, 3) containing the RGB values of each pixel.
    - The `astype(float)` conversion ensures that the pixel values are treated as floating-point numbers for subsequent calculations.
3. Define the sepia transformation formula:
    - The `formula` variable represents the sepia transformation formula as a 2D list.
4. Apply the sepia transformation:
    - The line `new_img_1d = np.clip(np.matmul(new_img_1d, formula), 0, 255)` applies the sepia transformation to the reshaped image array using matrix multiplication (`np.matmul`).
    - The `np.clip` function ensures that the pixel values are within the range of 0 to 255.
5. Reshape the transformed image:

- The line `return new_img_1d.reshape((h, w, 3)).round(0).astype(np.uint8)` reshapes the transformed image array back to its original dimensions (`h`, `w`, 3) and rounds the pixel values to the nearest integer.
- The `astype(np.uint8)` conversion ensures that the pixel values are represented as unsigned 8-bit integers.

Figure 5: Sepia

7 Function: rgb_to_sharpen

```
def rgb_to_sharpen(image, factor):
    image_array = np.array(image)
    # separate the channels
    r = image_array[:, :, 0]
    g = image_array[:, :, 1]
    b = image_array[:, :, 2]
    # sharpen each channel
    sharpen_r = sharpening_image(r)
    sharpen_g = sharpening_image(g)
    sharpen_b = sharpening_image(b)
    # combine the channels
    sharpen_image_array = np.dstack((sharpen_r, sharpen_g, sharpen_b))
    sharpen_image_array = sharpen_image_array.astype(np.uint8)
    sharpen_image = Image.fromarray(sharpen_image_array)
    return sharpen_image  # Return the Image object
```

7.0.1 Description

The `rgb_to_sharpen` function applies a sharpening filter to each channel of an RGB image. The function takes an image and a sharpening factor as inputs and returns the sharpened image.

7.0.2 Idea

1. Convert the input image to a numpy array:
   - The line `image_array = np.array(image)` converts the input image to a numpy array for further processing.
2. Separate the RGB channels:
   - The lines `r = image_array[:, :, 0]`, `g = image_array[:, :, 1]`, and `b = image_array[:, :, 2]` separate the red, green, and blue channels, respectively.
3. Apply sharpening to each channel:
   - The lines `sharpen_r = sharpening_image(r)`, `sharpen_g = sharpening_image(g)`, and `sharpen_b = sharpening_image(b)` apply the `sharpening_image` function to each channel individually.
4. Combine the sharpened channels:
   - The line `sharpen_image_array = np.dstack((sharpen_r, sharpen_g, sharpen_b))` stacks the sharpened channels back together to form the RGB image.

5. Convert the sharpened image array to an Image object:

    - The line `sharpen_image = Image.fromarray(sharpen_image_array)` converts the sharpened image array to an Image object.

6. Return the sharpened image.

Figure 6: Sharpened, kernel = 10

8 Function : convolve

```
def convolve(channel, kernel):
    k_rows, k_cols = kernel.shape
    padded_channel = np.pad(
        channel, ((k_rows // 2, k_rows // 2),
        (k_cols // 2, k_cols // 2)), mode="edge"
    )
    result = np.zeros_like(channel, dtype=np.float32)

    for i in range(channel.shape[0]):
        for j in range(channel.shape[1]):
            result[i, j] = np.sum(
                padded_channel[i : i + k_rows, j : j + k_cols] * kernel
            )

    return result
```

8.0.1 Description

The `convolve` function performs a convolution operation between a channel and a kernel. The function takes a channel (2D numpy array) and a kernel (2D numpy array) as inputs and returns the result of the convolution operation.

8.0.2 Idea

1. Obtain the dimensions of the kernel:
    - The line `k_rows, k_cols = kernel.shape` extracts the number of rows (`k_rows`) and columns (`k_cols`) of the kernel.
2. Pad the channel with zeros:
    - The line `padded_channel = np.pad(channel, ((k_rows // 2, k_rows // 2), (k_cols // 2, k_cols // 2)), mode="edge")` pads the channel with zeros on all sides using the `np.pad` function.
    - The padding size is determined by dividing the number of rows and columns of the kernel by 2.
    - The `mode="edge"` argument ensures that the edge pixels of the channel are extended to fill the padded regions.
3. Initialize an empty result array:
    - The line `result = np.zeros_like(channel, dtype=np.float32)` creates an array of the same shape as the channel, filled with zeros and with the same data type as the channel.

4. Perform the convolution operation:
    - The nested `for` loops iterate over each pixel in the channel.
    - The line `result[i, j] = np.sum(padded_channel[i : i + k_rows, j : j + k_cols] * kernel)` computes the dot product between the kernel and the corresponding region of the padded channel, and assigns the result to the corresponding pixel in the result array.
5. Return the result array.

9 Function: rgb_to_blur

```
def rgb_to_blur(image, kernel_size):
    # Convert the image to a numpy array
    image_array = np.array(image)

    # Check the number of color channels in the image
    num_channels = image_array.shape[2] if len(image_array.shape)
    == 3 else 1

    # Define the blurring kernel
    kernel = np.ones((kernel_size, kernel_size)) / (kernel_size**2)

    # Apply the blurring kernel to each color channel
    (or the grayscale channel)
    blurred_image_array = np.zeros_like(image_array, dtype=np.float32)

    for channel in range(num_channels):
        blurred_image_array[..., channel]
        = convolve(image_array[..., channel], kernel)

    # Clip values to ensure they are within the valid range
    blurred_image_array = np.clip(blurred_image_array, 0, 255)

    # Convert the blurred image array to an 8-bit integer array
    blurred_image_array = blurred_image_array.astype(np.uint8)

    # Convert the numpy array to an Image object
    blurred_image = Image.fromarray(blurred_image_array)

    return blurred_image  # Return the Image object
```

9.0.1 Description

The `rgb_to_blur` function applies a blurring filter to an RGB image. The function takes an image and a kernel size as inputs and returns the blurred

image.

9.0.2 Idea

1. Convert the input image to a numpy array:
   - The line `image_array = np.array(image)` converts the input image to a numpy array for further processing.
2. Check the number of color channels in the image:
   - The line `num_channels = image_array.shape[2] if len(image_array.shape) == 3 else 1` checks if the image has three dimensions (indicating RGB channels) or two dimensions (indicating a grayscale image).
3. Define the blurring kernel:
   - The line `kernel = np.ones((kernel_size, kernel_size)) / (kernel_size**2)` creates a kernel of size `kernel_size` with all elements set to 1, and then normalizes the kernel by dividing it by the square of `kernel_size`.
4. Apply the blurring kernel to each color channel (or the grayscale channel):
   - The line `blurred_image_array = np.zeros_like(image_array, dtype=np.float32)` creates an empty array of the same shape as the image array, filled with zeros and with the data type set to `np.float32`.
   - The `for` loop iterates over each channel, and the line `blurred_image_array[..., channel] = convolve(image_array[..., channel], kernel)` applies the `convolve` function to each channel individually, and assigns the result to the corresponding channel in the blurred image array.
5. Clip values to ensure they are within the valid range:
   - The line `blurred_image_array = np.clip(blurred_image_array, 0, 255)` ensures that all values in the blurred image array are between 0 and 255.
6. Convert the blurred image array to an 8-bit integer array:
   - The line `blurred_image_array = blurred_image_array.astype(np.uint8)` converts the blurred image array to an 8-bit integer array.
7. Convert the numpy array to an Image object:
   - The line `blurred_image = Image.fromarray(blurred_image_array)` converts the blurred image array to an Image object.
8. Return the blurred image.

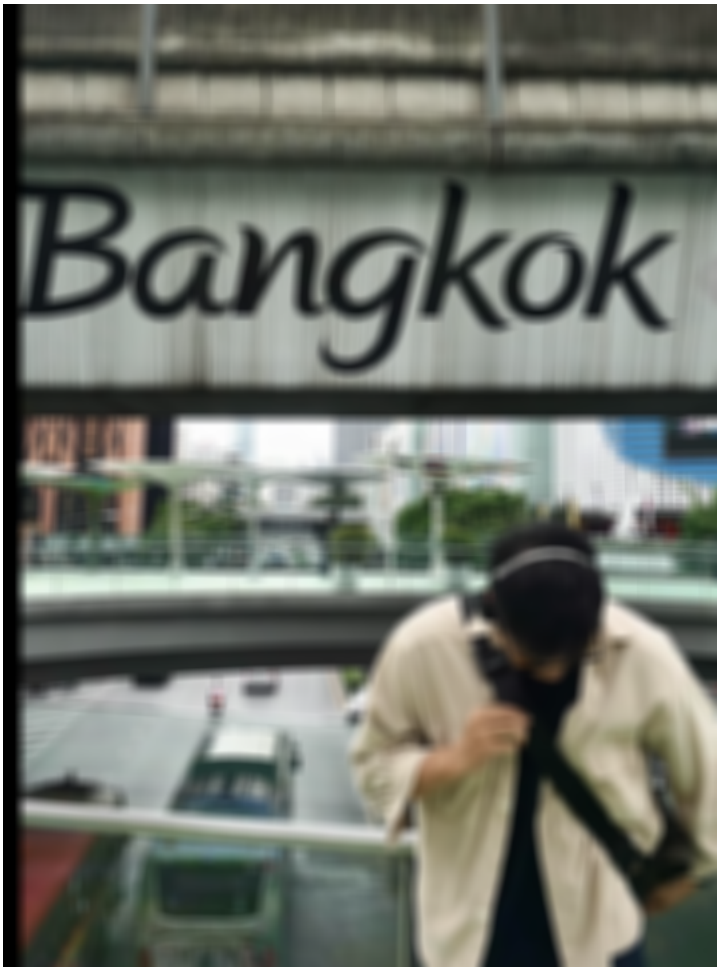

Figure 7: Blurred, kernel = 10

10 Function: crop_image

```
def crop_image(image_path, crop_height, crop_width, shape):
    # Open the image and convert it into a numpy array
    img = Image.open(image_path)
    img_np = np.array(img)

    # Calculate the center of the image and the coordinates
    of the rectangle you want to crop
    h, w, _ = img_np.shape
    ch, cw = crop_height, crop_width
    x1, y1 = w // 2 - cw // 2, h // 2 - ch // 2

    x2, y2 = w // 2 + cw // 2, h // 2 + ch // 2

    # Crop the image
    cropped_np = img_np[y1:y2, x1:x2]

    # Create a mask based on the shape
    center_y, center_x = cropped_np.shape[0] // 2, cropped_np.shape[1] // 2
    Y, X = np.ogrid[:ch, :cw]
    if shape == "circle":
        dist_from_center = np.sqrt((X - center_x) ** 2 + (Y - center_y) ** 2)
        radius = min(center_x, center_y)
        mask = dist_from_center <= radius
    elif shape == "rectangle":
        mask = np.ones((ch, cw), dtype=bool)
    elif shape == "ellipse":
        rx, ry = cw / 2, ch / 2
        mask1 = ((X - center_x) / rx) ** 2 + ((Y - center_y) / ry) ** 2 <= 1
        mask2 = ((X - center_x) / ry) ** 2 + ((Y - center_y) / rx) ** 2 <= 1
        mask = np.logical_or(mask1, mask2)
    else:
        raise ValueError("Invalid shape")

    num_channels = cropped_np.shape[2]
    mask = np.stack([mask] * num_channels, axis=-1)

    # Apply the mask to the image
    masked_img_np = np.where(mask, cropped_np, 0)  # Use 0 for black

    # Convert the masked image back to a PIL image
    masked_img = Image.fromarray(masked_img_np.astype(np.uint8))

    return masked_img
```

10.1 Description

The ‘crop_image‘ function crops an image based on the specified crop height, crop width, and shape. It returns a new image with the cropped region.

10.2 Idea

1. Open the image and convert it into a numpy array:
    - The line ‘img = Image.open(image_path)‘ opens the image using the provided image path.
    - The line ‘img_np = np.array(img)‘ converts the opened image to a numpy array for further processing.
2. Calculate the center of the image and the coordinates of the rectangle you want to crop:
    - The lines ‘h, w, _ = img_np.shape‘ and ‘ch, cw = crop_height, crop_width‘ store the height and width of the image and the desired crop height and width, respectively.
    - The lines ‘x1, y1 = w // 2 - cw // 2, h // 2 - ch // 2‘ and ‘x2, y2 = w // 2 + cw // 2, h // 2 + ch // 2‘ calculate the coordinates of the top-left and bottom-right corners of the rectangle to be cropped.
3. Crop the image:
    - The line ‘cropped_np = img_np[y1:y2, x1:x2]‘ extracts the desired region from the image using the calculated coordinates.
4. Create a mask based on the shape:
    - The lines ‘center_y, center_x = cropped_np.shape[0] // 2, cropped_np.shape[1] // 2‘ calculate the coordinates of the center of the cropped region.
    - The lines ‘Y, X = np.ogrid[:ch, :cw]‘ create coordinate grids for the cropped region.
    - Depending on the specified shape, different masks are created:
        - For a circle shape, the line ‘dist_from_center = np.sqrt((X - center_x) ** 2 + (Y - center_y) ** 2)‘ calculates the distance of each pixel in the cropped“‘

Figure 8: Rectangle , 400x400

Figure 9: Circle, 400x400

Figure 10: Ellipse, 400x500

11 Process_image

process_image(*image_name*, *function_id*)
  Read the image
  Convert image to numpy array
  if *function_id* = 1: *Change Brightness*
    Prompt user to enter brightness value
    Call **change_brightness**(*image_array*, *brightness*)
    Convert the numpy array back to an Image object
    Save the processed image
  elif *function_id* = 2: *Change Contrast*
    Prompt user to enter contrast value
    Call **change_contrast**(*image_array*, *contrast*)
    Convert the numpy array back to an Image object
    Save the processed image
  elif *function_id* = 3: *Flip Image*

Prompt user to enter flip direction
Call **flip_image**(*image_array*, *direction*)
Convert the numpy array back to an Image object
Save the processed image
elif *function_id* = 4: *Convert to Grayscale/Sepia Tone*
Prompt user to enter conversion type
if *conversion_type* = "grayscale":
Call **rgb_to_grayscale**(*image_array*)
Set processed_image_name = "grayscale.png"
elif *conversion_type* = "sepia":
Call **rgb_to_sepia**(*image_array*)
Set processed_image_name = "sepia.png"
Convert the numpy array back to an Image object
Save the processed image
elif *function_id* = 5: *Apply Blur/Sharpen*
Prompt user to enter filter type
if *filter_type* = "blur":
Prompt user to enter blur radius
Call **rgb_to_blur**(*image_array*, *radius*)
elif *filter_type* = "sharpen":
Prompt user to enter sharpen factor
Call **rgb_to_sharpen**(*image_array*, *factor*)
Save the processed image
elif *function_id* = 6: *Crop Image*
Prompt user to enter width, height, and shape
Call **crop_image**(*image_name*, *height*, *width*, *shape*)
Convert the numpy array back to an Image object
Save the processed image
else:
Print "Invalid function ID"
Return

12 Conclusion

In this report, we have discussed various image processing functions implemented in Python. Each function serves a specific purpose, such as adjusting brightness, contrast, flipping images, converting to grayscale, sepia tone, sharpening, and blurring. These functions can be used to enhance and modify images for different applications.

References

[1] Pillow Documentation,
https://pillow.readthedocs.io/en/stable/

[2] NumPy Documentation,
https://numpy.org/doc/

[3] Change Contrast Function,
https://hackernoon.com/image-processing-algorithms-adjusting-contrast-and-image-brig

[4] RGB to Grayscale Conversion Function,
https://e2eml.school/convert$_r gb_t o_g rayscale.html$

[5] RGB to Sepia Conversion Function,
https://yabirgb.com/sepia$_f ilter/$

[6] RGB to Sharpen Filter Function,
https://medium0.com/spinor/a-straightforward-introduction-to-image-blurring-smoothin