2023 Spring CS101 Homework #2

Image Processing

Total point: 50 pts

Out: 04 Apr 2023
Due: 11 Apr 2023

If you have questions regarding Homework 2, use the Homework 2 Q&A board at Elice.

(Korean) 학습도우미 > 게시판 > Homework 2 Q&A
(English) Help Area > Forums > Homework 2 Q&A

Overview

In this programming assignment, you will implement the following set of famous image-processing techniques:

- RGB-to-Gray Image and quantization (Task 1)
- Rotation (Task 2)
- Downsampling and upsampling (Task 3)

In each task, you will need to implement a function that takes an image as input and returns a modified image as output. Please refer to each task for more detailed explanations.

DO NOT USE any additional Python external modules except cs1media. If you do so, you will automatically receive a zero grade.

DO NOT WRITE CODE outside the allowed section. Otherwise, your program may not be properly graded.

Task 1: RGB2GRAY and quantization (15 pts)

Typically, colors in images are represented using integer values ranging from 0 to 255, which means there are 256 possible color levels. However, this exercise will consider a scenario where we can only use a specific number of levels (represented by num_level) to represent colors. This means we will need to change the original pixel values to new ones based on the following scheme.

Suppose that num_level is four. In this case, we will need to change the original pixel values (ranging from 0 to 255) into 0, 64, 128, or 192 according to the following rules:

Original pixel values → Changed pixel colors with four different Levels:

if the original value is in [0 to 63] **then** new value = 0

if the original value is in [64 to 127] **then** new value = 64

if the original value is in [128 to 191] **then** new value = 128

if the original value is in [192 to 255] **then** new value = 192

Before changing the color levels, we need to convert the RGB image into a grayscale image. This can be done by computing a grayscale value for each pixel using the following formula:

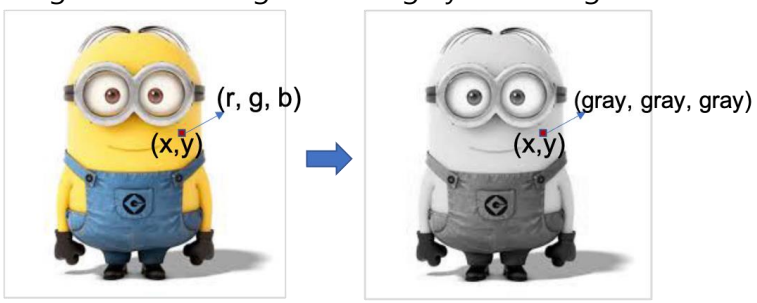
For every pixel (x,y) with color (r, g, b), the corresponding grayscale value can be computed as:

$original\_color_{(x,y)} = (r, g, b)$

$gray_{(x,y)} = int(0.3 * r + 0.59 * g + 0.11 * b)$

$new\_color_{(x,y)} = (gray_{(x,y)}, gray_{(x,y)}, gray_{(x,y)})$

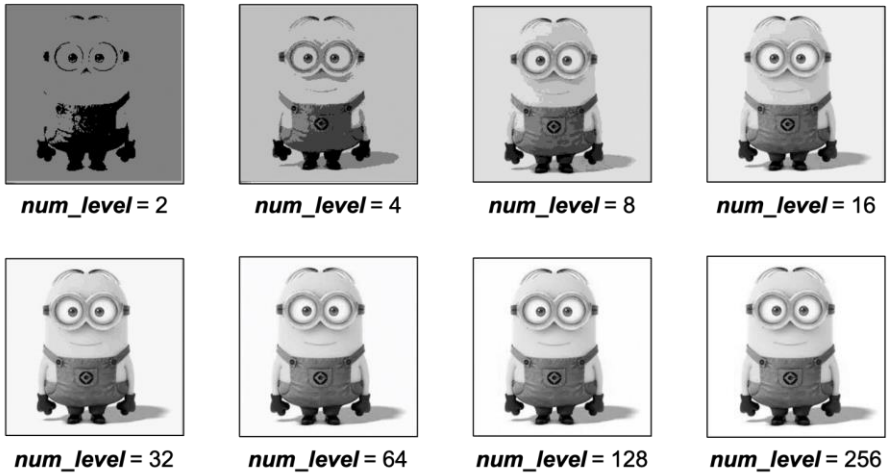
Once we have the grayscale value for each pixel, we can generate the grayscale image by setting the pixel colors to $(gray_{(x,y)}, gray_{(x,y)}, gray_{(x,y)})$.
Then, your RGB image will be changed into a grayscale image as follows:



RGB image Grayscale image

Finally, let's re-generate the image with different level numbers.

For the given num_level, the grayscale image is changed as follows:



With the following requirements, implement a Python function quant_img(input_image, num_level) that converts the given RGB image to a grayscale image and then changes the color representation according to the given level number:

Input:

- input_image [cs1media Picture object]
- num_level [int]
  (Note that num_level can take a value in [2, 4, 8, 16, 32, 64, 128, 256].)

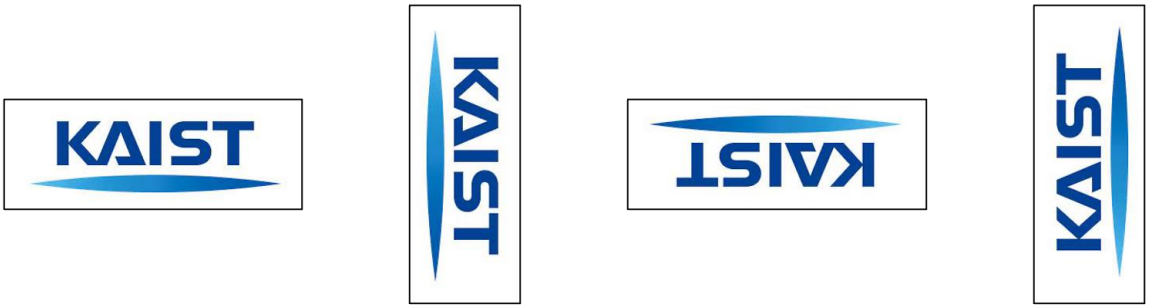
Output:

- Grayscale image with num_level color levels [cs1media Picture object]

Task 2: Image rotation (15 pts)

In this task, you will rotate an input image by the given angle in the clockwise direction.

Note: The width and height of the image are NOT always the same. Therefore, when the image is rotated by 90 or 270 degrees, your rotated image could have a different width and height from the original image. You can generate your own image with the ***create_picture(width, height)*** function in the cs1media module.
An example of rotation is demonstrated with the KAIST logo image.



With the following requirements, implement a Python function rotate_img(input_image, angle) that rotates the input image by the specified angle in the clockwise direction:

Input:

- input_image [cs1media Picture object]
- angle [int]
  (Note that angle can take value in [0, 90, 180, 270].)

Output:

- rotated_image [cs1media Picture object]

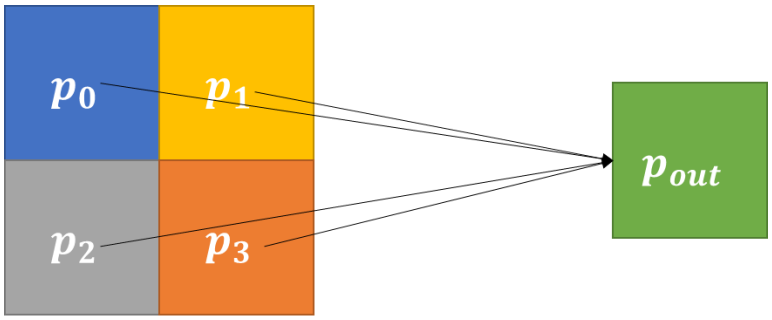
Task 3: Image resizing (20 pts)

In this task, you will implement two functions: image downsampling and image upsampling.

Task 3.1: Image downsampling (10 pts)

In this subtask, you will resize a given image to a smaller size (= downsampling) by a given factor while preserving the image ratio. Given an input image, your goal is to create a replica of the image with a reduced size.

One way to downsample an image without loss of visual information is by taking an average of the neighboring pixel values. For example, if we want to downsample an image by a factor of 2, we need to average the values of 2 x 2 adjacent pixels from the input image to fill out the corresponding pixel in the downsampled image. Refer to the diagram below:



$$p_{out} = \frac{1}{4}(p_0 + p_1 + p_2 + p_3)$$

The diagram above shows a grid of 2 x 2 pixels, each with four unique pixel values p_0, p_1, p_2, p_3. To fill out the target pixel p_{out} of the downsampled image, we need to take an average of the four-pixel values p_0, p_1, p_2, p_3.

Note that this is a case for downsampling by a factor of 2. For any arbitrary factor *n*, you will need to compute the downsampled pixel values by using *n x n* grid of adjacent pixels in the input image.

This is a result of downsampling an image by a factor of 2:

With the following requirements, implement a Python function downsample(input_image, rescaling_factor) that performs downsampling:

Input:

- input_image [cs1media Picture object]
- rescaling_factor [int]

output:

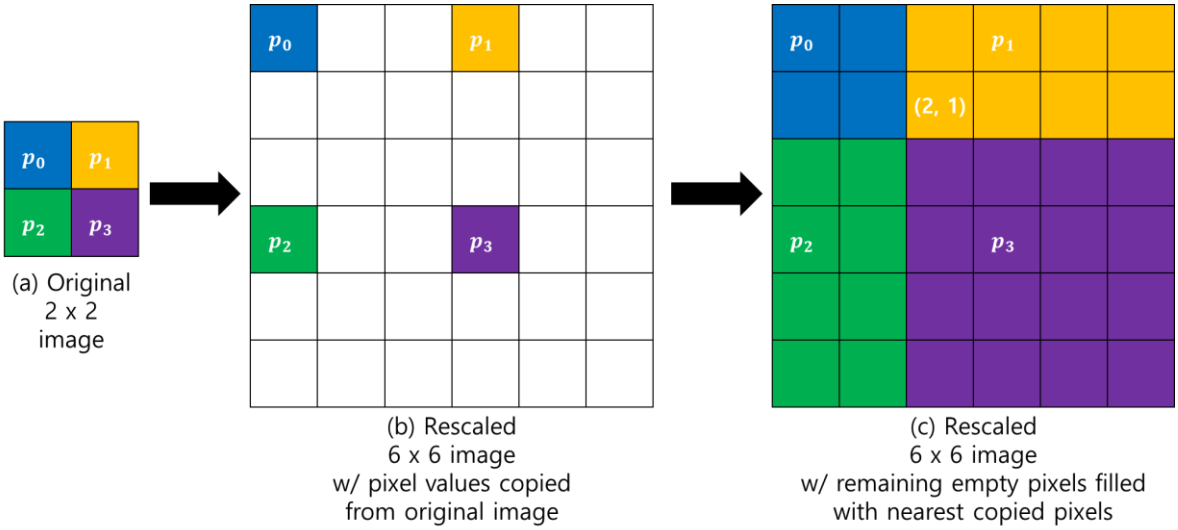
- downsampled_image [cs1media Picture object]

For ease of implementation, the rescaling_factor will be constrained such that the downsampled image ALWAYS results in integer image sizes (widths and heights). For example, there would be no case of downsampling a 32 x 32 image by a factor of 3.

Task 3.2: Image upsampling (10 pts)

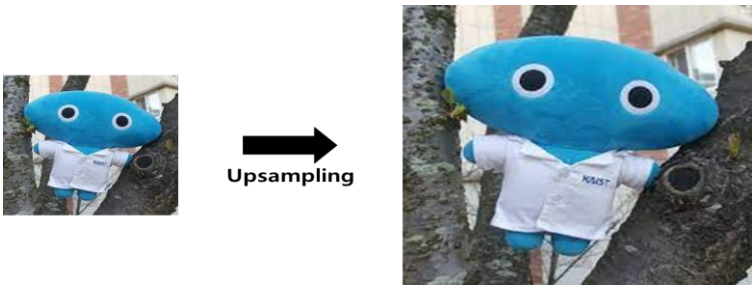
In this subtask, you will now resize a given image to a bigger size (= upsampling) by a given factor. Given an input image, your goal is to create a replica of the image with a enlarged size.

One way to upsample an image is using the nearest neighbor method. In upsampling, we need to fill out the empty values in the upsampled pixel grid. Nearest neighbor method does this by taking the value of the nearest pixel of the original image. Refer to the diagram below for an example of upsampling a 2 x 2 sized image by a factor of 3:



First, we rescale the original 2 x 2 image ((a) in the figure above) by a factor of 3 to create a 6 x 6 upsampled image. We then have to fill in the values of the pixels of the upsampled image. First, we copy the pixel values of the original unresized image to the corresponding pixels of the resized image. For example, the value of pixel p_3 at position $(1,1)$ of the original image will be copied to position $(3,3)$ of the upsampled image ((b) in the figure above). To fill in the remaining empty pixels, we refer to the values of the nearest ***copied*** pixels. For example, since the copied pixel that is nearest to the position $(2,1)$ in the upsampled image is p_1, we fill in the value of $(2,1)$ with the value of p_1 ((c) in the figure above).

This is the result of upsampling an image:



With the following requirements, implement a Python function upsample(input_image, rescaling_factor) that performs upsampling:

Input:

- input_image [cs1media Picture object]

- rescaling_factor [int]

output:

- upsampled_image [cs1media Picture object]

Note that there could be a case where an empty pixel has multiple nearest colored/original pixels. For ease of implementation, you don't have to worry about such cases. In other words, the rescaling factor will always be an ODD number. Also, for ease of implementation, the rescaling_factor will ALWAYS be an integer value.