Tutorial 2: Image Handling

Samuel Smith

September 14, 2023

1 Introduction

This tutorial provides an overview of image handling using Python and OpenCV, covering reading images, colour space transformations, pixel manipulation, and visualization.

2 Prerequisites

- Python
- OpenCV
- Matplotlib
- numpy

3 Tutorial

3.1 Reading an Image

We shall start with a recap of the basics from last week. The first thing we need to do when dealing with images is to make sure we have access to the image processing functions. To do this we will be using the OpenCV-python module. This is what you installed last week. To use it within our python code we need to import the opencv module. To do this we use `import cv2`. This enables use to use all the functions, class types and methods contained within the module. Use the following code to test the `imread` function. This

function takes a filepath as an input argument,then stores the pixel information in a numpy array variable called image, but you can supply the image variable with a name of your choosing. Numpy Arrays are like regular lists of integers or floats, but are optimised for faster array processing by utilising a c++ backend.

```
import cv2
image = cv2.imread('image.jpg')
```

Now we have our image data stored in a variable called image we can determine whether or not it is a colour image or grayscale image based on the number of dimensions. An RGB colour image has 3 channels, whereas a grayscale or binary image has 1 channel. Binary images have boolean data or 0s and 1s, whereas 8bit int grayscale images have a pixel range of 0-255. To check the number of dimensions we can apply the `shape` method to the numpy array. The `print` function then sends the output of shape to the console.

```
import cv2
import numpy as np
image = cv2.imread('image.jpg')
print(np.shape(image))
```

Question 1: What is the height and width of your image measured in pixels?

3.2 Converting to Grayscale

```
import cv2
import matplotlib.pyplot as plt
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow('Image', gray_image)
```

Grayscale images are used when colour information isn't required or isn't available. Many imaging technologies such as MRI, CAT, Ultrasound, radar and lidar don't use colour information. The `cvtColor` method allows us to convert our images to a number of different colour spaces and formats.
Let's look at the input arguments, the first parameter is our image variable, the second parameter defines the conversion method. Traditionally, colour

images are described with the channels in a specific order, 1.Red, 2.Green, 3.Blue (RGB). However, one quirk of openCV is that it imports the image with the channels in reverse order (BGR). Therefore we must make sure we use the correct conversion method. We want to go from a BGR image to a grayscale, so we must use the `cv2.COLOR_BGR2GRAY` conversion method.

Question 2: Plot the BGR image using both `cv2.imshow` and `plt.imshow` functions from the cv2 and matplotlib modules. What differences do you observe?

3.3 Converting to Binary

Binary images are extremely important for a wide range of tasks. The pixel values of a binary image are either 0 or 1. This means that when you convert an image to binary, you are dividing the image into two distinct regions: those with a value above a threshold, represented by a 1, and those with a value below a threshold, represented by a 0. This grouping allows us to do many great things, such as creating an image mask. If we multiply the image pixels by a corresponding binary image mask, anything multiplied by 0 will become 0, and anything multiplied by 1 will remain the same. This enables us to hide unimportant parts of an image and focus on regions of interest. z

```
1       _, binary_image = cv2.threshold(gray_image, 127, 255, cv2.
            ↪ THRESH_BINARY)
```

The above code shows how to apply a threshold to a grayscale image in order to convert it to binary. Note the outputs of the function, there are 2. The first return value is ignored by using the _ syntax. The first value returned by the cv2.threshold function is the threshold value that was applied. However in this example, we specified a threshold value of 127 (the second input parameter/argument), so we don't need to know it. The second return value is called binary_image and that contains our binary image output from the threshold function. There are many other methods that can be applied to determine a threshold value, the most popular is known as Otsu's method. For further information on the different approaches utilised in OpenCV please review the following documentation: `https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html`

3.4 Accessing Pixels, Rows, Columns, and Channels

One of the most fundamental aspects of image processing is understanding the concept of indexing. In a list, or an array, each value within the variable is given an index. When we modify the value of a pixel, we access it through its index location. Python uses 0 indexing, meaning the first location in the variable is assigned with a 0. Some other programming languages such as Matlab and Octave begin their index with a 1. Images are typically two-dimensional (sometimes three-dimensional), and RGB images contain three channels. We can index the various locations using the following convention: rows, columns, and layers. Rows run horizontally, so the first horizontal row has the index 0, the second row has 1, the third row has 2, and so on. Columns run vertically; therefore, the horizontal location of an image is described by the column. It is crucial to understand this convention, as it may seem counter-intuitive to some.

Px(0,0) denotes the pixel in the top left corner of a 2-D image, Px(1,0) is the pixel underneath. Px(0,1) is the second pixel along from the top left corner on the top row. The layer or colour channel is the third index. For a RGB image the Red layer is 0, Green layer is 1 and the blue layer is 2. We access pixels by indexing with [ ] brackets. We can use the colon : operator to access more than one pixel at a time. When used by itself, the colon operator essentially tells the program to take everything in the row, or everything in the column, or all the channels.

```
# Access a single pixel
pixel_value = image[50, 50]
# Access a row
row_values = image[50, :]
# Access a channel
red_channel = image[:, :, 0]
```

Question 3: How would you index to select the all of the second row from the green channel?

```
# Increase the intensity of the red channel by 50
image[:, :, 0] += 50
```

Question 4: The line of code above increases the value of all the pixels in the red channel of an RGB image by 50. Modify the code to subtract 25 from the blue channel.

3.5 Plots and Subplots

Often in image processing tasks, we would like to view multiple images side by side. Or an image with some data. We can achieve this using subplots.

```
import matplotlib.pyplot as plt

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(thresholded, cv2.
    ↪ COLOR_BGR2RGB))
plt.axis('off')
plt.title('Thresholded Image')

plt.show()
```

Question 4: Modify the above code to display 4 images in a 2x2 grid

4 Questions

1. How can you determine the number of dimensions of an image?
2. What is the indexing order for accessing pixel values in images?
3. How can you copy mutable variables in Python

5 Coding Tasks

1. Create a function that reads an image and returns its grayscale and binary versions. Test the function with different images.
2. Modify the colour intensity of the blue and green channels of an image by 30 and 60, respectively. Then, visualize the original and modified images side-by-side.
3. Apply two different types of thresholding techniques and visualize them in a 1x3 subplot along with the original grayscale image.

4. Take an RGB image and separate its colour channels. Then, create a new image that swaps the red and blue channels. Finally, display the original and modified images side by side