Messing around with OpenCV and Images

CHINMAY AMRUTKAR
chinmay.amrutkar@asu.edu

1. Image Display and Color Correction

To display, we have chosen an image that was captured on a mobile near Hole in the Rock in Phoenix, Arizona. It features a small water body reflecting the clear sky, surrounded by desert plants and tall palm trees.

```
        #Load an image
        img = cv2.imread('A1_I1.jpg')
        #Display an image
        plt.imshow(img)
        plt.show()

        # Convert the image from BGR to
         ↪  RGB
        image_rgb = cv2.cvtColor(img,
         ↪  cv2.COLOR_BGR2RGB)
        plt.imshow(image_rgb)
        plt.show()
```

We first begin by loading the image using `cv2.imread` and display it using `plt.imshow` and `plt.show`. On observation, we can see that the colors are inverted as a result of the default format BGR of `cv2.imread`. So, in order to view the image in RGB format, we use `cv2.cvtColor(img, cv2.COLOR_BGR2RGB)` as shown in the code.

You can see that there is a clear difference in the output of both images, as shown in Figure 1 and Figure 2.

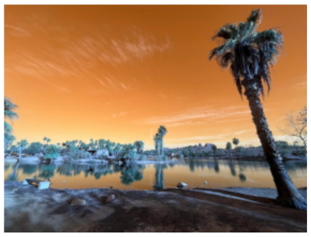
Figure 1. BGR Image

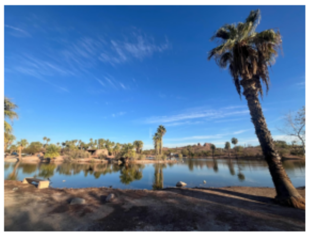
Figure 2. RGB Image

2. Image Display and Color Correction

We first start by zooming in on the image by defining the coordinates in the code below. We have focused on the center of the image, specifically at the top of two palm trees, as shown in Figure 3.

```
# Define the cropping coordinates (x,
 ↪  y, width, height)
x, y, w, h = 560, 460, 70, 70
cropped_image = image_rgb[y:y+h, x:x+w]
```

The zoomed image's texture appears grainy and pixelated, the color displays blue background, green leaves in the middle from the palm tree leaves, and yellowish-brown trunk reflecting the sunlight. This is because Digital images are captured at a fixed resolution that consists of a limited number of pixels that represent the scene. When we magnify image beyond its original size, individual pixels become more visible, leading to loss of detail, increased graininess, and blurriness.

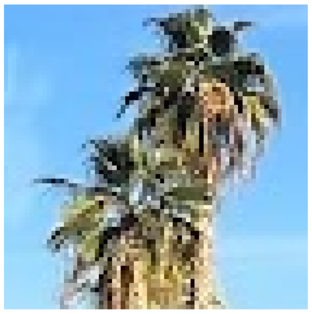
Figure 3. Zoomed Image

3. Image Resizing (Downsampling)

To downsample i.e. process of reducing its resolution by decreasing the number of pixels, we used `cv2.resize` shown in code below.

```
img = cv2.imread('A1_I2.jpg')
image_rgb = cv2.cvtColor(img,
 ↪  cv2.COLOR_BGR2RGB)
downsampled = cv2.resize(image_rgb,
 ↪  (None, None), fx=0.1, fy=0.1)
```

The original image Figure 4 and downsized image Figure 5 are shown below.

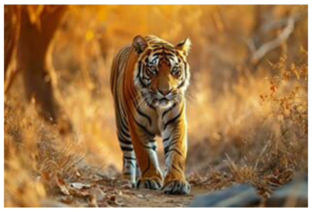
Figure 4. Original Image

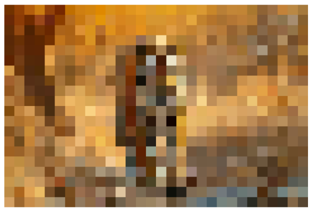
Figure 5. Downsampled Image

4. Image Resizing (Upsampling)

To upsample the same downsampled image, by 10x i.e. its original resolution, we have used two interpolation methods- nearest neighbor and bicubic methods, using `cv2.resize` and changing the arguments to the functions as shown in code below.

```
# Upsample the image using nearest
↪  neighbor interpolation
nearest_neighbor_image =
↪  cv2.resize(downsampled, (None,
↪  None), fx=10, fy=10,
↪  interpolation=cv2.INTER_NEAREST)
# Upsample the image using bicubic
↪  interpolation
bicubic_image = cv2.resize(downsampled,
↪  (None, None), fx=10, fy=10,
↪  interpolation=cv2.INTER_CUBIC)
```

The upsampled image using nearest neighbor interpolation appears blocky and pixelated due to the replication of the nearest pixel values, whereas, upsampled image using bicubic interpolation appears smoother and more visually pleasing as it uses a weighted average of pixels in a 4x4 neighborhood to calculate the new pixel values, as shown in Figure 6 and Figure 7 respectively.

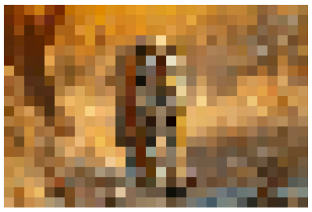
Figure 6. Nearest Neighbor

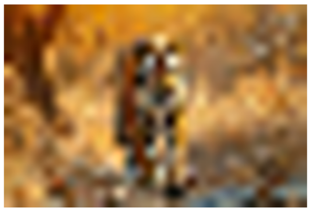
Figure 7. Bicubic

From the comparison, we can conclude that the bicubic interpolation generally provides better visual quality for upsampling images, whereas the nearest-neighbor interpolation may be useful for certain applications where blocky, pixelated effects are desired.

5. Calculating Absolute Difference and Summing Pixel Values

To calculate the absolute difference between the ground truth image and the two upsampled images, we use the command `cv2.absdiff` as shown in code below, and get difference image for nearest-neighbor method in Figure 8 and bicubic method in Figure 9.

```
    # Calculate the absolute difference
    ↪  between the ground truth image
    ↪  and the upsampled images
    difference_nearest =
    ↪  cv2.absdiff(image_rgb,
    ↪  nearest_neighbor_image)
    difference_bicubic =
    ↪  cv2.absdiff(image_rgb,
    ↪  bicubic_image)
    # Sum all the pixels in the
    ↪  difference images
    sum_difference_nearest =
    ↪  np.sum(difference_nearest)
    sum_difference_bicubic =
    ↪  np.sum(difference_bicubic)
```

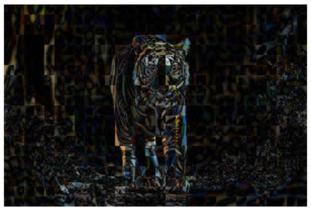
Figure 8. Nearest Neighbor

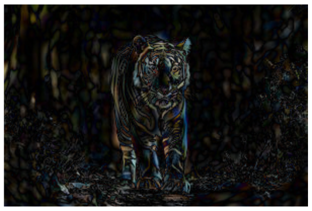
Figure 9. Bicubic Difference

We then sum all pixels in the difference images and analyze the results to determine which method causes less error. We found that the sum of pixel differences for Nearest-Neighbor 3446690, whereas the sum of pixel differences for Bicubic was 3137288. From this data, we can conclude that the bicubic method caused less error during upsampling, as it had lower pixel difference than nearest-neighbor method.

For designing retro video games or for artistic pixelated effects or blocky effect Nearest-Neighbor Interpolation would be more useful whereas for applications requiring smooth and visually pleasing images like photo editing or image printing, bicubic method would be more useful.

6. Acknowledgments

List of resources used:

- https://www.geeksforgeeks.org/image-resizing-using-opencv-python/
- https://www.youtube.com/watch?v=ZihsQU7h9-E&t=205s