EEE 515 : Machine Vision

Yongbaek Cho

1. Problem 1. Setting up Coding Environment

2. Problem 2. Messing around with OpenCV and Images

   1. Import all the dependencies.

```
import cv2
import matplotlib.pyplot as plt
import ipdb
import numpy as np

path = r'C:\Users\danny\PycharmProjects\pythonProject1\pythonProject\pythonProject'
'''
Problem #2 : Messing around with OpenCV and Images
'''
```

   2. Loading and Displaying images

      a. Read in the image.

      Code:

```
################## section 2 ######################
# a. Show the image "elephant.jpeg"
image = cv2.imread('elephant.jpeg') #load image
plt.axis('off') # remove axis
plt.imshow(image) # display the image
plt.show() # display image
filename = 'elephant_opencv.png' # filename of image
cv2.imwrite(filename, image) # save/write the image
```

Output1: display with image using opencv

Comment: OpenCV provides the order of the BGR color channels.

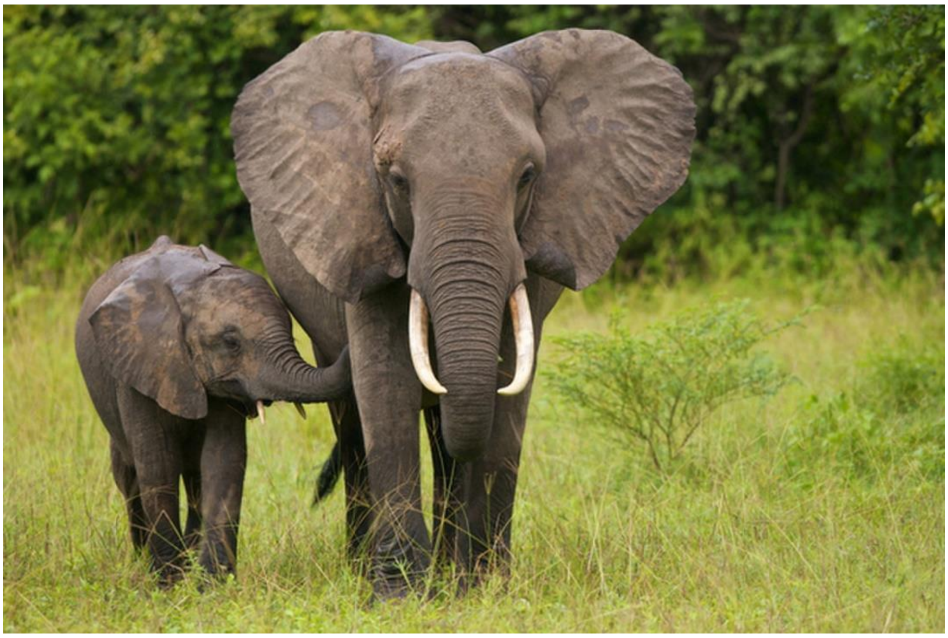
b.

Code:

```
# b. BGR to RGB
src = cv2.imread('elephant_opencv.png') # load image
image2RGB = cv2.cvtColor(src, cv2.COLOR_BGR2RGB) # convert image(BGR to RGB)
filename2 = 'elephant_matplotlib.png' #file name
cv2.imwrite(filename2,image2RGB) # save/write the image
```

Output1: with plt.imshow()

Output2 : with cv2.imwrite() "elephant_opencv.png"



c. Grady scale

Code:

```
# C. Gray scale
src1 = cv2.imread('elephant_matplotlib.png') #load image
img2gray = cv2.imread('elephant_matplotlib.png', 0) #load image
plt.imshow(img2gray, cmap='gray') # gray scale with using matplotlib parameter "cmap = gray"
filename3 = 'elephant_gray.png'
cv2.imwrite(filename3, img2gray)
```

Output:

3. Problem 3. Cropping the image

Code:

```
'''
Problem #3 : Cropping Image
'''
# Crop the image
crop_img = image[400:900, 100:600] #crop the image based on baby elephant
plt.imshow(crop_img) #display the image
plt.show() #display the img
filename4 = 'babyelephant.png' #file name
cv2.imwrite(filename4,crop_img) # save / write the image
```

Output: “babayelephant.png”

4. Problem 4. Pixel-wise Arithmetic Operations:

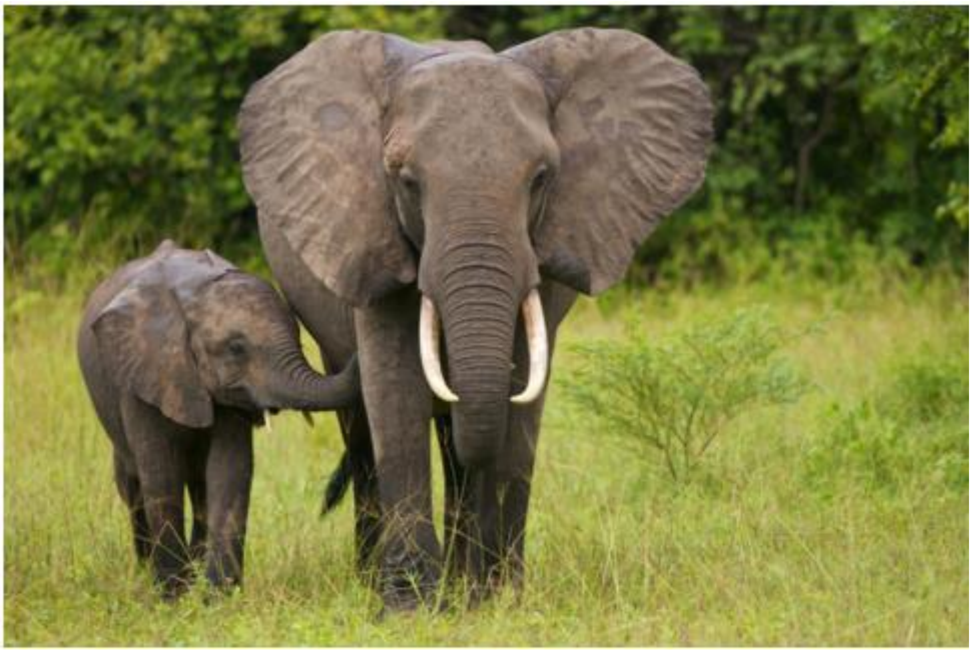
   a & b.

   Code:

```
'''
Problem #4 : Pixel-wise Arithmetic Operation
'''
# a & b add 256 (image = image + 256)
add_image = image2RGB + 256 # add 256
print(add_image.dtype) # print the image's datatype
img2uint8 = np.uint8(add_image) #convert back to unit8
print(img2uint8)
plt.axis('off')
plt.imshow(img2uint8)
plt.show()
```

Output:

After adding 256, the image's datatype is `uint16`



Comment: It looks like there is no change in the image after casting it into np.unit8()

The main difference between an 8-bit image and a 16-bit image is the amount of tones available for a given color. In terms of color, 8-bit image can hold 16,000,000 colors but 16-bit image can hold 28,000,000,000. However, the human eye cannot distinguish if it the image exceeds over 16 million colors. That is why there is no change in the image.

b. Split R,G,B separately and add 256.

Code:

```
# c split RGB channels seperately
#image2split = cv2.imread('elephant_matplotlib.png')
R,G,B= cv2.split(image2RGB) # split the image with R,G,B

print(R)
print(G)
print(B)
R_256 = cv2.add(R,256) # ADD 256
print(R_256)
G_256 = cv2.add(G, 256) # ADD 256
print(G_256)
B_256 = cv2.add(B, 256) # ADD 256
print(B_256)

inversebgr = cv2.merge((R, G, B)) # Merge the channels back together
print(inversebgr)
plt.imshow(inversebgr)
plt.show()
```

Output:

Comment: It looks likt BGR channel and the reason for the difference from step b is that OpcnCV.add is a saturated operation while Numpy.add is a modulo operation.

5. Problem 5. Resizing images

   a & b. Read image in RGB and 10x downsampling

Code:

```python
'''
Problem #5 : Resizing images
'''
# a read in the image in color again and convert it to RGB space
src4 = cv2.imread('elephant.jpeg')
convert_img = cv2.cvtColor(src4, cv2.COLOR_BGR2RGB)
# b downsample the image by 10x in width and height
image3 = cv2.resize(convert_img, None, fx = 0.1, fy = 0.1) #downsampling 10x
filename4 = 'elephant_10xdown.png' #filename
cv2.imwrite(filename4,image3) #save / write the image
plt.imshow(image3)
plt.show()
```

Output:

c. 10x Upsampling from downsampled image with 2 different interpolations

Code:

```python
# c back to original resolution with 2 different interpolation: nearest neightbor and bicubic
src5  = cv2.imread('elephant_10xdown.png')
#convert_img_2 = cv2.cvtColor(src5, cv2.COLOR_BGR2RGB)
image4 = cv2.resize(src5, None, fx = 10, fy = 10, interpolation= cv2.INTER_NEAREST) #upsampling 10x nearest neighbor
image5 = cv2.resize(src5, None, fx = 10, fy = 10, interpolation= cv2.INTER_CUBIC) #upsampling 10x bicubic
filename5 = 'elephant_10xup_method_nn.png'
filename6 = 'elephant_10xup_method_bicubic.png'
cv2.imwrite(filename5,image4)
cv2.imwrite(filename6,image5)
plt.imshow(image4)
plt.imshow(image5)
plt.show()
```

Output:

1. Nearest Neighbor

2. Bicubic

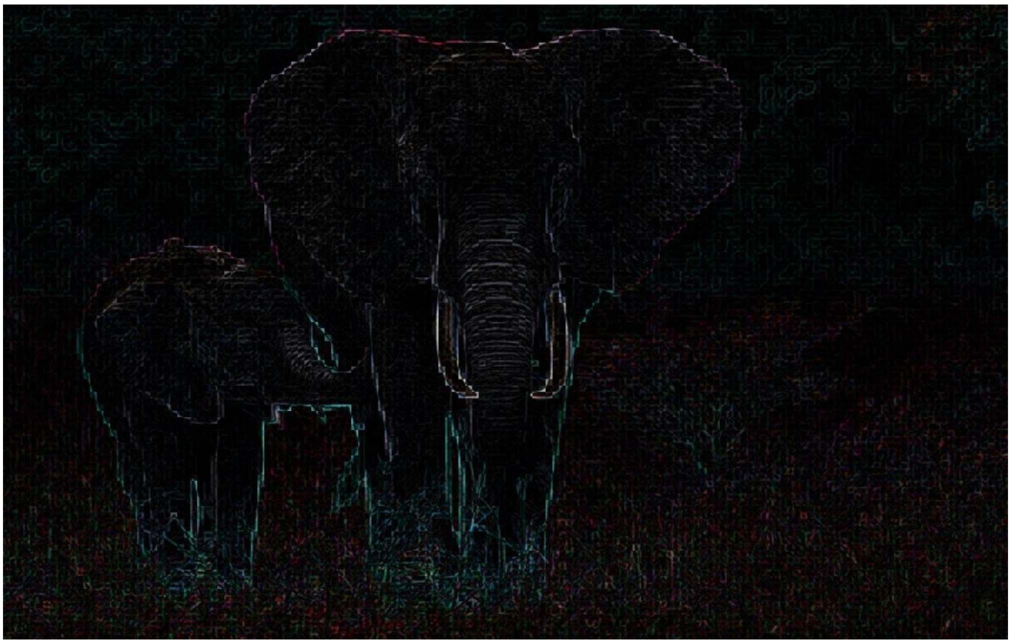
d. Calculate the absolute difference between the ground truth image and the 2 unsampled images with the two methods.

   Code:

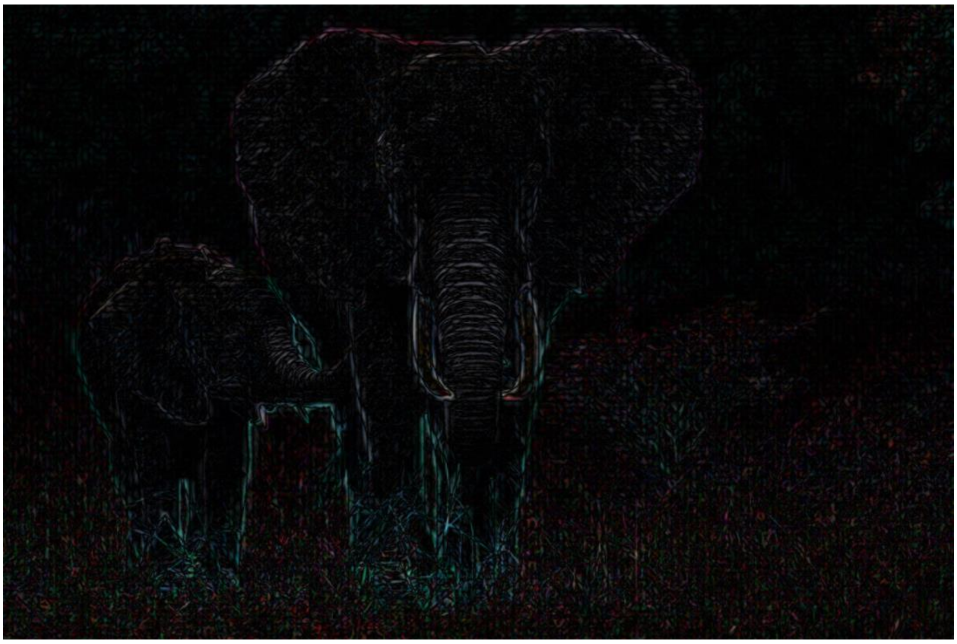
```
# d calculate the absolute difference between the ground truth image and the two upsampled images
original_image = image2RGB #original image
upsample_nn = cv2.imread('elephant_10xup_method_nn.png') #load image
upsample_bicubic = cv2.imread('elephant_10xup_method_bicubic.png') #load image
diff_1 = cv2.absdiff(original_image, upsample_nn) #compute absolute difference
diff_2 = cv2.absdiff(original_image, upsample_bicubic) #compute absolute difference
cv2.imwrite('elephant_diff_near.png', diff_1) #save
cv2.imwrite('elephant_diff_bicub.png', diff_2) #save

print(diff_1)
print(diff_2)
print(diff_1.sum())
print(diff_2.sum())
```

Output:

1. Nearest Neighbor with absolute difference



2. Bicubic



Following number is the absolute difference.

First: nearest neighbor

Second : bicubic

```
46599821
40261755
```

Questions: Which method caused less error in upsampling?

Answer: Bicubic Interpolation shows less error in upsampling