CSE 107: Lab 03: Image Resizing.

Valentinno Cruz

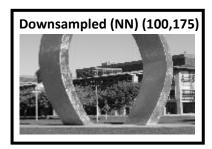
LAB: Thu 4:30-7:20

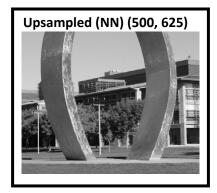
Yuxin Tian
October 21<sup>st</sup>, 2022

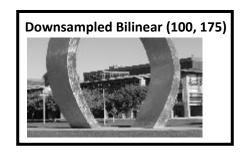
### Abstract:

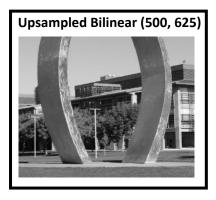
The resizing of Images is a technique used for manipulating an images. This allows the image to be downsized(shrunk) or upsized(enlarged). There are several ways of doing this process, with some being more complicated than others. These methods include Nearest Neighbor, Bilinear, and Bicubic interpolation. These methods essentially all calculate each resized pixel with the use of the locations surrounding the given pixel. However, some methods are much quicker, while others are more accurate. For instance, Nearest Neighbor is processed much quicker and with less power in comparison to Bilinear. This is because Bilinear has to process much more information than that of Nearest Neighbor.

## **Qualitative Results:**









## **Quantitative Results:**

	N.N. interpolation	Bilinear interpolation
Downsample then upsample	30.639599	24.838016
Upsample then downsample	23.793549	12.761159

#### **Questions:**

1. Visually compare the two downsampled images, one using nearest neighbor interpolation and one using bilinear interpolation. How are they different and why based on what you know about the two interpolations?

At first glance it may be difficult to distinguish the two images, but upon further inspection it is evident that the Bilinear photo has a slightly higher contrast. This is due to Bilinear using a lot more resources when selecting each pixel value. This in turn creates a more dynamic picture than that of Nearest Neighbor.

2. Visually compare the two down then upsampled images, one using nearest neighbor interpolation and one using bilinear interpolation. How are they different? Which one looks better to you? Does this agree with the RMSE values?

The Nearest Neighbor upscaled image looks a lot more accurate to the original than that of the Bilinear image. I was really surprised to see this because I was almost certain the Bilinear image would be much more accurate. My RMSE value for Bilinear is smaller, so I think maybe RMSE might not be the best way to gauge images that are restored from a smaller size.

3. Visually compare the two up then downsampled images, one using nearest neighbor interpolation and one using bilinear interpolation. How are they different? Which one looks better to you? Does this agree with the RMSE values?

The Bilinear image looks a little more clear than that of the Nearest Neighbor. This does make sense in comparision to the RMSE values.

4. If your image resizing is implemented correctly, you should get an RMSE value of zero between the original image and the up then downsampled one using nearest neighbor interpolation. Why is this the case?

A RMSE is 0 when there is a perfect fit in the data. This means that the up then downsampled image that used Nearest Neighbor is as accurate to the original image as possible

5. What was the most difficult part of this assignment?

Coming up with a way to code the split Bilinear and Nearest Neighbor depending on the string that's called is what took me the longest to implement.

```
Test myimresize.py
 rom PIL import Image, ImageOps
 import <mark>numpy</mark> as np
 from numpy import asarray
orig_im = Image.open('LAB 3\Lab_03 image.tif')
orig im.show()
orig_im_pixels = asarray(orig_im, dtype=np.float32)
from MyImageFunctions import myImageResize
downsampled_im_NN_pixels = myImageResize(orig_im_pixels, 100, 175, 'nearest')
downsampled_im_NN = Image.fromarray(np.uint8(downsampled_im_NN_pixels.round()))
downsampled_im_NN.show()
downsampled_im_NN.save('downsampled_NN.tif');
down_up_sampled_im_NN_pixels = myImageResize(downsampled_im_NN_pixels, 400, 400, 'nearest')
down_up_sampled_im_NN = Image.fromarray(np.uint8(down_up_sampled_im_NN_pixels.round()))
 own_up_sampled_im_NN.show()
Import myRMSE from MyImageFunction rom MyImageFunctions import myRMSE
own_up_NN_RMSE = myRMSE( orig_im_pixels, down_up_sampled_im_NN_pixels)
 rint('\nDownsample/upsample with mvimresize using nearest neighbor interpolation = %f' % down up NN RMSE'
ownsampled im bilinear pixels = myImageResize(orig im pixels, 100, 175, 'bilinear')
 wnsampled_im_bilinear = Image.fromarray(np.uint8(downsampled_im_bilinear_pixels.round()))
 ownsampled_im_bilinear.show()
ownsampled im bilinear.save('downsampled bilinear.tif');
 own_up_sampled_im_bilinear_pixels = myImageResize(downsampled_im_bilinear_pixels, 400, 400, 'bilinear')
 own_up_sampled_im_bilinear = Image.fromarray(np.uint8(down_up_sampled_im_bilinear_pixels.round()))
lown up sampled im bilinear.show()
 own_up_bilinear_RMSE = myRMSE( orig_im_pixels, down_up_sampled_im_bilinear_pixels)
 rint('Downsample/upsample with myimresize using bilinear interpolation = %f' % down_up_bilinear_RMSE)
```

```
upsampled_im_NN_pixels = myImageResize(orig_im_pixels, 500, 625, 'nearest')
upsampled_im_NN = Image.fromarray(np.uint8(upsampled_im_NN_pixels.round()))
 osampled_im_NN.show()
upsampled_im_NN.save('upsampled_NN.tif');
 Downsample the numpy matrix to the original size using nearest neighbor interpolation.
p_down_sampled_im_NN_pixels = myImageResize(upsampled_im_NN_pixels, 400, 400, 'nearest')
up_down_sampled_im_NN = Image.fromarray(np.uint8(up_down_sampled_im_NN_pixels.round()))
up_down_sampled_im_NN.show()
# Compute RMSE between original numpy matrix and down then upsampled nea
up_down_NN_RMSE = myRMSE( orig_im_pixels, up_down_sampled_im_NN_pixels)
 rint('\nUpsample/downsample with myimresize using nearest neighbor interpolation = %f' % up_down_NN_RMSE)
 osampled_im_bilinear_pixels = myImageResize(orig_im_pixels, 500, 625, 'bilinear')
upsampled_im_bilinear = Image.fromarray(np.uint8(upsampled_im_bilinear_pixels.round()))
 osampled im bilinear.show()
upsampled im bilinear.save('upsampled bilinear.tif');
up_down_sampled_im_bilinear_pixels = myImageResize(upsampled_im_bilinear_pixels, 400, 400, 'bilinear')
up_down_sampled_im_bilinear = Image.fromarray(np.uint8(up_down_sampled_im_bilinear_pixels.round()))
up_down_sampled_im_bilinear.show()
up_down_bilinear_RMSE = myRMSE( orig_im_pixels, up_down_sampled_im_bilinear_pixels)
orint('Upsample/downsample with myimresize using bilinear interpolation = %f' % up_down_bilinear_RMSE)
```

```
import numpy as np
import math
def myImageResize(inImage_pixels, M, N, interpolation_method):
    ImageOut = np.zeros((M, N))
   r, c = inImage_pixels.shape[:2]
   cRat = c / N
   rRat = r / M
    if interpolation_method == 'nearest':
       for i in range(M):
            for j in range(N):
                y, x = int(j * cRat), int(i * rRat)
                ImageOut[i][j] = inImage_pixels[x][y]
    elif interpolation_method == 'bilinear':
        for i in range(M):
            for j in range(N):
                y, x = j * cRat, i * rRat
                x_floor, y_floor = int(x), int(y)
                x_{ceil}, y_{ceil} = min(r - 1, int(math.ceil(x))), min(c - 1, int(math.ceil(y)))
                p1 = inImage_pixels[x_floor, y_floor]
                p2 = inImage_pixels[x_floor, y_ceil]
                p3 = inImage_pixels[x_ceil, y_floor]
                p4 = inImage_pixels[x_ceil, y_ceil]
                ImageOut[i, j] = mybilinear(x_floor, y_floor, p1,
                                            x_floor, y_ceil, p2,
                                            x_ceil, y_floor, p3,
                                            x_ceil, y_ceil, p4,
                                            x, y)
   return ImageOut
def myRMSE(first_im_pixels, second_im_pixels):
    M, N = np.shape(first_im_pixels)
   rmse = 0
   for m in range(M):
        for n in range(N):
            rmse += (first_im_pixels[m,n] - second_im_pixels[m,n])**2
    rmse = np.sqrt(rmse/(M*N))
    return rmse
```

```
def mybilinear(x1,y1,p1,x2,y2,p2,x3,y3,p3,x4,y4,p4,x5,y5):
    Min_x = min(np.floor(x1),np.floor(x2),np.floor(x3),np.floor(x4))
    Max_x = max(np.ceil(x1),np.floor(x2),np.floor(x3),np.floor(x4))
    Min_y = min(np.floor(y1),np.floor(y2),np.floor(y3),np.floor(y4))
    Max_y = max(np.ceil(y1),np.floor(y2),np.floor(y3),np.floor(y4))
    if (Min_x == Max_x):
       return p1
    if (Min_y == Max_y):
       return p1
    if (Min_x == Max_x):
        totx = (Max_y - y5) * p1 + (y5 - Min_y) * p4
        return totx
    if (Min_y == Max_y):
       min_max_y = (Max_x - x5) * p1 + (x5 - Min_x) * p4
        return min_max_y
    val_1 = (Max_x - x5) * p1 + (x5 - Min_x) * p2

val_2 = (Max_x - x5) * p3 + (x5 - Min_x) * p4
    Bilinear = (Max_y - y5) * val_1 + (y5 - Min_y) * val_2
    return Bilinear
```

# Task 3: Creating a gradient grayscale image. Computing the image average.



Figure 3. The Gradient Image

## Questions for task 3:

- 1. What is the average pixel value in your gradient image?
  - a. 127.5
- 2. Why did you expect to get this value from the gradient image?
  - a. Because the average between 0 & 255 is 127.5
- 3. What was the most difficult part of this task?
  - a. Figuring out how to code the average

```
from PIL import Image, ImageOps
    import numpy as np
    from numpy import asarray
    import math
    rows = 100
     cols = 256
     im_pixels = np.zeros(shape=(rows, cols))
18
    x = np.linspace(255, 0, 256)
19
20
    image2_grad = np.tile(x, (100, 1)).T
24
    a = np.rot90(image2_grad, 1, (1,0))
    image_pix = asarray(a) # set as an array
26
    newImage = Image.fromarray(np.uint8(image_pix))
29
     x = np.size(image_pix, 0)
     y = np.size(image_pix, 1)
    total =0
    for i in range(x):
         for j in range(y):
             total+=image_pix[(i,j)]
         b=x*y
     mean=(total)/b
     print("Mean Value is: ",np.mean(mean))
    newImage.show()
    newImage.save('image.tif')
```