# CSE 107: Lab 03: Image Resizing

Jonathan Ballona Sanchez
Lab: W 4:30-7:20pm
Yuxin Tian
October 20, 2022

Abstract:

The intent of this lab was to see the interpolation methods of nearest neighbor and bilinear for resizing images including downsampling then upsampling and upsampling then downsampling. The significance of the effects of using both interpolation methods is to see when one interpolation method would be appropriate for which resizing use case. These results are on the side of visual comparisons which may include bias but in the categories of gradient smoothness, object outlines and textures, it is easier to see that bilinear interpolation is better for downsampling then upsampling and nearest-neighbor interpolation for upsampling and downsampling.

Qualitative Results

Figure 1: Downsampled to size (100, 175) using nearest neighbor interpolation.

Figure 2: Downsampled to size (100, 175) using bilinear interpolation

Figure 3: Upsampled to size (500, 625) using nearest neighbor interpolation.

Figure 4: Upsampled to size (500, 625) using bilinear interpolation.

Quantitative Results

| | Nearest neighbor interpolation | Bilinear interpolation |
|---|---|---|
| Downsample then upsample | 23.027788 | 19.867488 |
| Upsample then downsample | 0.000000 | 9.963456 |

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?

    a. The similarities between both downsampled images are that they have similar colors to images of the twin peaks from UC Merced. There are two main differences between one another. The first difference is the edges of the images, and the edges are more jagged using nearest neighbor interpolation compared to the smoother edges in the bilinear interpolation. The smoothness of the edges is so because the nearest neighbor interp. Take the solid pixel values closest to it so the edge which causes more contrast compared to the bilinear interpolation which has not so sharp edges because of the linearity that is calculated.
    b. The smoothness of the edges is so because of the nearest neighbor interp. Take the solid pixel values closest to it so the edge which causes more contrast compared to the bilinear interpolation which has not so sharp edges because of the linearity that is calculated. The linearity computes average values that will hold texture details to be perceived.
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?
    a. The similarity in both images is that they both generally show the same objects. The difference for the nearest neighbor is that it has higher contrast, looks clearer but it has more jagged edges with little smooth lines compared to bilinear interpolation which looks more blurry but the gradient in objects and the lines are both smoother.
    b. The image with bilinear interpolation looks better overall because with this image the windows are more distinguishable, the outline of the roof is clearer and shadows of the trees are more distinguishable.
    c. The RMSE values agree with me because the values of the down then upsampled images using bilinear interpolation has a lower value compared to the nearest neighbor interpolation.
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?
    a. The images are the same but the texture, the gradient of the image and also the clarity all differ. The nearest neighbor interpolation appears to have clearer and well defined textures, building outlines, and distinguishable shadows compared to the bilinear interpolation approach. The bilinear interpolation approach doesn't have as good contrast, good gradient, but the clarity lacks compared to nearest neighbor.
    b. The nearest neighbor interpolation for the up then downsampled image looks better.

    c. This opinion agrees with my RMSE values because the RMSE of the nearest neighbor interpolation has a RMSE score  0.0 which is lower than the bilinear interpolation RMSE score.
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. The nearest neighbor should have an RMSE value of zero because the RMSE value is supposed to represent the computational difference between the pixel values in the two matrices, which is evident when a 0.0 score due to all the values in the resized image directly gets its values from the original image instead of a computed pixel value that is not even in the original image. There are no gradient pixel values that are averages of surrounding pixels, but pixel values of the neighbors instead.
5. What was the most difficult part of this assignment?
    a. There were two big difficulties for this assignment. The first difficulty was simply putting the concept of nearest neighbor interpolation into code that would not return an out of bounds error. Once the nearest neighbor implementation was done, I was confused on how to represent the pixel values linearly as in a matrix or a system of equations. The systems of equations was the easier route as it was less info to keep track of and easier to follow using the lecture notes.

MyImageFunctions.py

```
import math

from PIL import Image, ImageOps

# Import numpy
import numpy as np
from numpy import asarray


def myImageResize(numpyMatrix, rows, cols, interpolType):

   M = rows
   N = cols

   # create an empty matrix for resize image
   new_matrix = np.zeros(shape=(M, N))
   oldRows, oldCols = numpyMatrix.shape

   # ratios to multiply y5 and x5 values
   verticalRatio = oldRows / M
   horizontalRatio = oldCols / N

   for m in range(0, M):
       for n in range(0, N):

           # find x5 and y5 based on new image size
           p_x = (horizontalRatio * (n - 0.5)) + 0.5
           p_y = (verticalRatio * (m - 0.5)) + 0.5

           # check to see what rendering method
           if interpolType == 'bilinear':
               # check for out of bounds
               coordX = myTupleCompute(p_x, oldCols)
               coordY = myTupleCompute(p_y, oldRows)

               # get pixel values for 4 surrounding pixels of x5 and y5
               p_1 = numpyMatrix[int(coordY[0]), int(coordX[0])]
               p_2 = numpyMatrix[int(coordY[1]), int(coordX[1])]
```

```
                p_3 = numpyMatrix[int(coordY[0]), int(coordX[1])]
                p_4 = numpyMatrix[int(coordY[1]), int(coordX[0])]
                # retrieve the p5 value from the interpolated values
                p5 = mybilinear(
                    # x1, y1, p1
                    coordX[0], coordY[0], p_1,
                    # x2, y2, p2
                    coordX[0], coordY[1], p_2,
                    # x3, y3, p3
                    coordX[1], coordX[0], p_3,
                    # x4, y4, p4
                    coordX[1], coordY[1], p_4,
                    # x5, y5
                    p_x, p_y)
                # assign p5
                new_matrix[m, n] = p5
            else:
                # logic for nearest neighbor
                p_x = round((horizontalRatio * (n + 0.5)) - 0.5)
                p_y = round((verticalRatio * (m + 0.5)) - 0.5)
                # assign new image pixel value from nearest neighbor
                new_matrix[m, n] = numpyMatrix[int(p_y), int(p_x)]

    return new_matrix


def mybilinear(x1, y1, p1, x2, y2, p2, x3, y3, p3, x4, y4, p4, x5, y5):
    # use system of equations to compute p5
    p5_i = (((p3 - p1) * (x5 - x1)) / (x3 - x1)) + p1

    p5_ii = (((p4 - p2) * (x5 - x2)) / (x4-x2)) + p2

    p5 = (((p5_ii - p5_i) * (y5 - y1)) / (y2-y1)) + p5_i

    return p5


def myTupleCompute(x, M):
    # If x is an integer
    if x % 1 == 0:
        m1 = 0
        m2 = 0
```

```
        return [m1, m2]
    else:
        # if x is a decimal
        if x < 1.0:

            m1 = 0.0
            m2 = 1.0
            return [m1, m2]
        # if x is greater than the bounds of the image
        elif x > M-1:
            m1 = M-2
            m2 = M-1
            return [m1, m2]
        else:
            m1 = np.floor(x)
            m2 = np.ceil(x)
            return [m1, m2]


def myRMSE(origIm, newIm):
    orig_im_pixels = asarray(origIm, dtype=np.float32)
    new_im_pixels = asarray(newIm, dtype=np.float32)
    rmseScore = 0

    rows, cols = orig_im_pixels.shape
    # for loop to compute difference in pixel value for two seperate matrices
    for i in range(0, rows):
        for j in range(0, cols):
            pixelDif = orig_im_pixels[i, j] - new_im_pixels[i, j]
            squaredPixelDif = pixelDif * pixelDif
            rmseScore += squaredPixelDif
    # sqrt = ^0.5
    rmseScore = (1/(rows * cols)) * rmseScore
    rmseScore = (rmseScore**(0.5))
    return rmseScore
```