Three Common Ways of Image Interpolation

Mai Bo 12011727

Abstract—Digital image processing involves acquiring, processing, analyzing, and interpreting images using digital computers. It employs a range of techniques and methods to extract useful information from digital images, improve image quality, and accomplish specific image processing tasks with precision and expertise. Scaling, rotating, and distorting images involve image interpolation, which is the process of estimating pixel values at other locations using known pixel values in the image.This article discusses the implementation and application of Nearest Neighbor Interpolation, Bilinear Interpolation, and Bicubic Interpolation in Python code, and discusses the advantages and disadvantages of each of the three interpolation methods, and advocates the purposeful application of one of the methods for image interpolation in different application environments and needs.

1 Introduction and Background

Digital image processing is a discipline that involves the use of computers to acquire, process, analyze, and interpret images. With the development of computer technology, digital image processing techniques have been widely used in medical imaging and other fields, with the aim of extracting useful information from images, improving image quality, and realizing specific image processing tasks. In digital image processing, image interpolation is an important technique used to scale, rotate, and deform images to meet the needs of different application scenarios. Through interpolation methods, the missing pixel values in the new image can be estimated to maintain the accuracy and quality of the image.Three of the more common methods in image interpolation are nearest neighbor interpolation, bilinear interpolation and bicubic interpolation.This paper discusses the similarities and differences of these three methods and compares their effects from the aspects of algorithm theory to code implementation, and gives suggestions for their use in specific scenarios.

2 Experiment and Results

2.1 Nearest neighbor interpolation

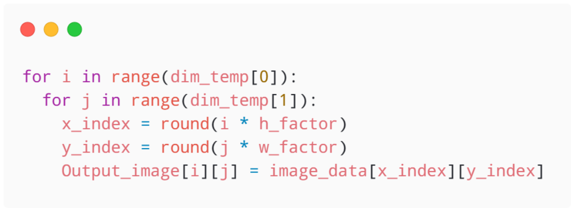
Nearest neighbor interpolation is the simplest interpolation of gray values. It is also called zero-order interpolation. It makes the gray value of the transformed pixel equal to the gray value of the input pixel nearest to it.

Known that the size of initial image $f(x,y)$ and the output image $g(m,n)$ are $f_x * f_y$ and $g_m * g_n$ respectively, the value $g(m_p, n_p)$ of a particular point P is the value of the nearest neighbour of that pixel point in the original image after a coordinate transformation.

The Pseudo code of the algorithm is shown in $Algorithm1$ and the corresponding Python code is shown in Fig.1.

Algorithm 1: Nearest neighbor interpolation

```
Data: Initial image $f(x,y)$, Input size $a*b$, Output size $m*n$
Result: Output image $g(m,n)$
1 for each point i in the output image do
2     Do coordinate transformation: $x_i = \frac{m_i}{m} * a$, $y_i = \frac{n_i}{n} * b$
3     Find the nearest neighbor point $f(x_p, y_p)$
4     Assign $f(x_p, y_p)$ to $g(m_i, n_i)$
5 end
```



```python
for i in range(dim_temp[0]):
  for j in range(dim_temp[1]):
    x_index = round(i * h_factor)
    y_index = round(j * w_factor)
    Output_image[i][j] = image_data[x_index][y_index]
```

Fig. 1: The Python code of nearest neighbor interpolation

2.2 Bilinear interpolation

The core idea of bilinear interpolation is to perform a single linear interpolation in each of the two directions of the image.

Known that the four nearest neighbors of a pixel point P (x_p, y_p) are $Q_{11}(x_1, y_1)$, $Q_{12}(x_1, y_2)$,$Q_{21}(x_2, y_1)$ and $Q_{22}(x_2, y_2)$, two linear interpolations in the x-direction and y-direction are derviated by:
x-directions:

$$f(m_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{1}$$

$$f(m_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \tag{2}$$

y-directions:

$$f(x_p, y_p) = \frac{y_2 - y_p}{y_2 - y_1} f(m_1) + \frac{y_p - y_1}{y_2 - y_1} f(m_2) \tag{3}$$

Since it is clear that $x_2 = x_1 + 1$, $y_2 = y_1 + 1$, the derviation can be simplified to:

$$\begin{aligned} f(x_p, y_p) = & f(Q_{11})(x_2 - x_p)(y_2 - y_p) + && (4)\\ & f(Q_{21})(x_p - x1)(y_2 - y_p) + && (5)\\ & f(Q12)(x_2 - x_p)(y_p - y_1) + && (6)\\ & f(Q_{22})(x_p - x_1)(y_p - y_1) && (7) \end{aligned}$$

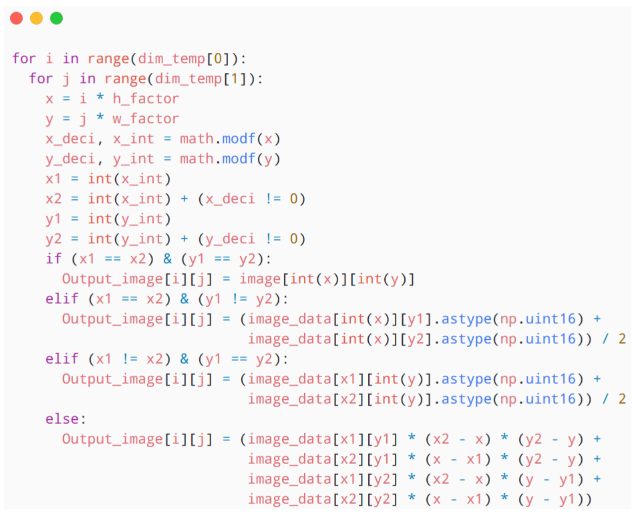
According to this formula, the Pseudo code of the algorithm is shown in $Algorithm2$ and the corresponding Python code is shown in Fig.2. Some special cases that may result in zero are also handled in Python code.

To make it easier to understand, the equation can be further simplified to :

$$f(x_p, y_p) = a_1 * f(Q_{11}) + a_2 * f(Q_{12}) + \tag{8}$$
$$a_3 * f(Q_{21}) + a_4 * f(Q_{22}) \tag{9}$$

From this equation it can be seen that bilinear interpolation is a weighted summation of the pixel values from its four nearest neighbouring points and the weights are related to the distance between that point and the four neighbouring points.

```
Algorithm 2: Bilinear interpolation
  Data: Initial image $f(x, y)$, Input size $a * b$, Output size $m * n$
  Result: Output image $g(m, n)$
1 for each point i in the output image do
2     Do coordinate transformation: $x_i = \frac{m_i}{m} * a$, $y_i = \frac{n_i}{n} * b$
3     Find the four nearest neighbor point $Q_{11}$,$Q_{12}$,$Q_{21}$,$Q_{22}$
4     Apply the formula and assign the value to $g(m_i, n_i)$
5 end
```



```
for i in range(dim_temp[0]):
  for j in range(dim_temp[1]):
    x = i * h_factor
    y = j * w_factor
    x_deci, x_int = math.modf(x)
    y_deci, y_int = math.modf(y)
    x1 = int(x_int)
    x2 = int(x_int) + (x_deci != 0)
    y1 = int(y_int)
    y2 = int(y_int) + (y_deci != 0)
    if (x1 == x2) & (y1 == y2):
      Output_image[i][j] = image[int(x)][int(y)]
    elif (x1 == x2) & (y1 != y2):
      Output_image[i][j] = (image_data[int(x)][y1].astype(np.uint16) +
                            image_data[int(x)][y2].astype(np.uint16)) / 2
    elif (x1 != x2) & (y1 == y2):
      Output_image[i][j] = (image_data[x1][int(y)].astype(np.uint16) +
                            image_data[x2][int(y)].astype(np.uint16)) / 2
    else:
      Output_image[i][j] = (image_data[x1][y1] * (x2 - x) * (y2 - y) +
                            image_data[x2][y1] * (x - x1) * (y2 - y) +
                            image_data[x1][y2] * (x2 - x) * (y - y1) +
                            image_data[x2][y2] * (x - x1) * (y - y1))
```

Fig. 2: The Python code of bilinear interpolation

2.3 Bicubic interpolation

Like the bilinear interpolation method, this method also uses mapping to obtain the pixel values in the enlarged image by weighting them in the neighbourhood of the mapped points. The difference is that the bicubic interpolation method requires 16 points in the immediate neighbourhood of the P point to be weighted. We first construct a BiCubic function, which is a function used to compute the weights in front of a nearest neighbour point based on the relative position of that point to the P-point:

$$W(x) = \begin{cases} x = & (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \leq 1 \\ y = & a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2 \\ z = & 0 & \text{otherwise} \end{cases} \tag{10}$$

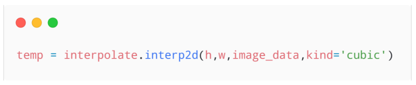
Here a is generally taken as $a = -0.5$.And $|x|$ is the distance to the P-point.After obtaining the weights, it is only necessary to weight up the pixel values of these 16 points, and the formula for interpolation calculation is as follows:

$$f(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} f(x_i, y_j) W(x - x_i) W(y - y_j) \tag{11}$$

As for the selection of point P, it is obtained according to the mapping formula of the nearest neighbour interpolation method, and the 16 points of the nearest neighbours of point P are also selected according to their relative positions.

The Pseudo code of the bicubic interpolation algorithm is shown in $Algorithm3$ and the corresponding Python code is shown in Fig.3.

```
Algorithm 3: Bicubic interpolation
  Data: Initial image $f(x, y)$, Input size $a * b$, Output size $m * n$
  Result: Output image $g(m, n)$
1 for each point i in the output image do
2     Do coordinate transformation: $x_i = \frac{m_i}{m} * a$, $y_i = \frac{n_i}{n} * b$
3     Find the 16 nearest neighbor points $Q_0 - Q_{15}$
4     Calculate the distance and weights
5     Apply the formula and assign the value to $g(m_i, n_i)$
6 end
```



```
temp = interpolate.interp2d(h,w,image_data,kind='cubic')
```

Fig. 3: The Python code of bicubic interpolation

2.4 Comparison results

In order to compare the effect of the three interpolation methods as well as their respective advantages and disadvantages, this paper uses a picture of rice as input and zooms in and out of each of the three interpolation methods and records their running speeds, the results of which are shown in Fig.4-Fig.10 and Table 1.

Firstly, from the aspect of image quality comparison and analysis, the image edge obtained by nearest interpolation is seriously serrated and the quality is poor; The image quality obtained by bicubic interpolation is the best, that is, the image effect is $bicubic > bilinear >> nearest$. In addition, it should be $bicubic >> bilinear > nearest$ in computational complexity and theoretical speed. But since bicubic interpolation calls the Python function "interp2d" from packet "scipy" directly, the speed is optimized to present the results in Table 1. Considering the cost of

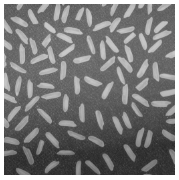


Fig. 4: Original rice picture

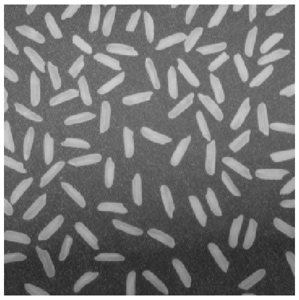


Fig. 5: Enlarged_nearest

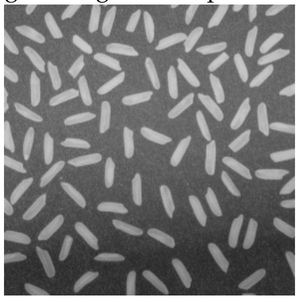


Fig. 6: Enlarged_bilinear

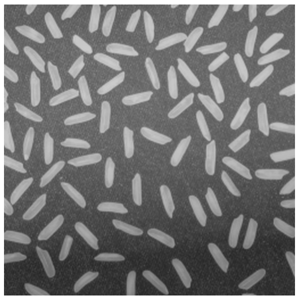


Fig. 7: Enlarged_bicubic

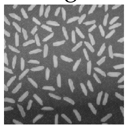


Fig. 8: Shrunk_nearest

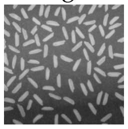


Fig. 9: Shrunk_bilinear

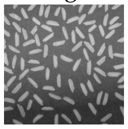


Fig. 10: Shrunk_bicubic

TABLE 1: The running speeds of three methods

Method	Nearest	Bilinear	Bicubic
Time(s)	0.1176	1.4523	0.1346

various aspects of practical application scenarios, bilinear interpolation method is generally adopted as the default image interpolation method because of its low computational complexity and the expected output image effect. In the case of high precision requirements, bicubic method or some other algorithms with higher precision will be used, while in the case of poor hardware computing power, nearest neighbor interpolation can only be used.

3 Conclusion

In general, the nearest neighbor interpolation is the simplest interpolation method, the calculation speed is fast but the image quality is low. Bilinear interpolation achieves a balance between speed and quality. Bicubic interpolation provides higher quality image magnification, but it is more expensive to compute. In practical applications, the appropriate interpolation method can be selected according to the different needs of image quality and the actual amount of hardware computing power.