Image Resizing and Interpolation

Introduction

Image resizing is an essential process in digital image processing that alters an image's dimensions while aiming to preserve quality. It is commonly used in applications such as web design and photography, where fitting various resolutions is necessary. Interpolation methods estimate new pixel values to reduce artifacts during resizing. This study compares four techniques—nearest neighbor, bilinear, bicubic, and biquadratic—to evaluate their effectiveness on different images.

Problem Statement

Resizing images can degrade quality, and different interpolation methods produce varying results. This paper aims to determine which methods—nearest neighbor, bilinear, bicubic, or biquadratic—are most effective under specific conditions and why some perform better than others.

Interpolation Methods

Interpolation is a mathematical technique used to estimate new pixel values when resizing images. Different interpolation methods vary in their approach and effectiveness, impacting the quality of the resized image. This section will explore four commonly used methods—nearest neighbor, bilinear, bicubic, and biquadratic—highlighting their principles, advantages, and limitations in preserving image quality during the resizing process. For this comparison I will be choosing nearest neighbor and bicubic interpolation methods.

Nearest Neighbor

Nearest neighbor interpolation is the simplest method for resizing images. It works by assigning the value of the nearest pixel in the original image to the new pixel in the resized image. This technique is fast and easy to implement but can result in blocky or pixelated images, especially when enlarging, as it does not consider the values of surrounding pixels. While it

may be suitable for certain applications where speed is prioritized over quality, it often lacks the smoothness and detail provided by more advanced interpolation methods.

Example

$$\text{Original 2x2 Matrix} = \begin{bmatrix} 100 & 1 \\ 0 & 255 \end{bmatrix} \text{ Resized 4x4 Matrix} = \begin{bmatrix} 100 & 100 & 1 & 1 \\ 100 & 100 & 1 & 1 \\ 0 & 0 & 255 & 255 \\ 0 & 0 & 255 & 255 \end{bmatrix}$$

Code

Inputs:

`image`: array of pixels

`scale`: float representing rescaling factor

```
import numpy as np

def nearest_neighbor_interpolation(input_image, scale_factor):
    # Get the dimensions of the original image (height, width, number of channels)
    original_height, original_width, num_channels = input_image.shape

    # Compute the dimensions of the new scaled image
    new_height = int(original_height * scale_factor)
    new_width = int(original_width * scale_factor)

    # Create an empty array for the output image with the new dimensions
    # The dtype is kept the same as the input image to ensure consistency
    output_image = np.zeros((new_height, new_width, num_channels), dtype=input_image.dtype)

    # Loop over each pixel in the new (output) image
    for i in range(new_height):
        for j in range(new_width):
            # Find the nearest pixel in the original image
            # Scale the indices back to the original image by dividing by the scale factor
            # Use 'min' to avoid out-of-bounds indices (clipping at the edge of the original
image)
            original_i = min(int(i / scale_factor), original_height - 1)
            original_j = min(int(j / scale_factor), original_width - 1)

            # Copy the value of the nearest pixel from the original image to the new image
            output_image[i, j] = input_image[original_i, original_j]

    # Return the scaled output image
    return output_image
```

Bicubic Interpolation

Bicubic interpolation is a method used to estimate unknown values within a 2D grid by using the values of 16 surrounding points (in a 4×4 grid). It is an extension of cubic interpolation and is commonly applied in image processing to produce smoother and more detailed results compared to simpler methods like nearest neighbor and bilinear interpolation. By considering more neighboring pixels, bicubic interpolation provides better accuracy and smoother transitions, making it ideal for tasks like image resizing and transformation.

Code

Inputs:

`image`: array of pixels

`scale`: float representing rescaling factor

```
import numpy as np


def cubic_interpolate(p, x):
    # Cubic interpolation formula using 4 points
    return (
            p[1] +
            0.5 * x * (p[2] - p[0] +
                       x * (2.0 * p[0] - 5.0 * p[1] + 4.0 * p[2] - p[3] +
                            x * (3.0 * (p[1] - p[2]) + p[3] - p[0])))
    )

def bicubic_interpolation(image, scale):
    orig_height, orig_width, channels = image.shape
    new_height = int(orig_height * scale)
    new_width = int(orig_width * scale)

    # Output image
    new_image = np.zeros((new_height, new_width, channels), dtype=np.uint8)

    # Scale ratios
    row_scale = orig_height / new_height
    col_scale = orig_width / new_width

    for i in range(new_height):
        for j in range(new_width):
            # Corresponding position in the original image
            x = i * row_scale
            y = j * col_scale

            # Calculate indices of the top-left corner of the 4x4 neighborhood
            x0 = int(np.floor(x)) - 1
            y0 = int(np.floor(y)) - 1

            # Distance from the original pixel position
            dx = x - np.floor(x)
            dy = y - np.floor(y)

            # Iterate over channels (e.g., RGB)
            for c in range(channels):
                # 4x4 neighborhood
                neighborhood = np.zeros((4, 4), dtype=np.float32)

                for m in range(4):
                    for n in range(4):
                        # Get the pixel values, handling boundary conditions with clamping
                        xn = np.clip(x0 + m, 0, orig_height - 1)
                        yn = np.clip(y0 + n, 0, orig_width - 1)
                        neighborhood[m, n] = image[int(xn), int(yn), c]

                # Interpolate along x for each row
                interpolated_rows = np.array([cubic_interpolate(neighborhood[m, :], dy) for m
in range(4)])
```

```
                # Interpolate along y to get the final value
                pixel_value = cubic_interpolate(interpolated_rows, dx)

                # Clip the value to the range [0, 255] and assign it to the new image
                new_image[i, j, c] = np.clip(pixel_value, 0, 255)

    return new_image
```

Visualization Code

Following is a method to run

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


def plot():
    # Convert BGR to RGB for displaying with matplotlib
    input_image_rgb = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
    resized_image_rgb = cv2.cvtColor(resized_image_neighbor, cv2.COLOR_BGR2RGB)
    resized_image_quad_rgb = cv2.cvtColor(resized_image_biquad, cv2.COLOR_BGR2RGB)
    # Enable interactive mode
    plt.ion()
    # Display the original image in a separate window
    plt.figure(figsize=(6, 6))
    plt.imshow(input_image_rgb)
    plt.axis('off')
    plt.show()
    # Display the nearest neighbor resized image in a separate window
    plt.figure(figsize=(6, 6))
    plt.imshow(resized_image_rgb)
    plt.axis('off')
    plt.show()
    # Display the bicubic resized image in a separate window
    plt.figure(figsize=(6, 6))
    plt.imshow(resized_image_quad_rgb)
    plt.axis('off')
    plt.show()
    # Keep the plots open
    plt.ioff()  # Turn off interactive mode
    plt.show()  # This ensures that the last window stays open until closed


if __name__ == '__main__':
    image = 'image.jpg'
    input_image = cv2.imread(image)
    scale_factor = 5

    # Print dimensions of the input image
    print(f"Input Image Dimensions: {input_image.shape[1]}x{input_image.shape[0]} (Width x He
ight)")

        # Resize the input image
    resized_image_neighbor = nearest_neighbor_interpolation(input_image, scale_factor)
    resized_image_biquad = bicubic_interpolation(input_image, scale_factor)
```

```
    # Print dimensions of the resized image
    print(f"Resized Image Dimensions: {resized_image_neighbor.shape[1]}x{resized_image_neighb
or.shape[0]} (Width x Height)")

    plot()
```

Experimental Setup

Image Selection

When selecting images for testing interpolation methods, it's essential to consider their characteristics. For instance, images with smooth gradients, such as landscapes or soft portraits, are ideal for evaluating `bicubic interpolation`, as this method excels in producing smooth transitions and preserving fine details. Conversely, `nearest neighbor interpolation` performs well on images with distinct edges and flat areas of color, such as pixel art or logos, where sharpness is paramount, but still gets outperformed. Choosing a low-resolution image for significant upscaling can highlight the strengths and weaknesses of both methods, allowing for a clear comparison of how each interpolation technique handles various types of content. By analyzing these differences, we can better understand the applications and limitations of each method.

Testing

Bicubic Interpolation Works Better

The displayed images highlight the differences between the original and the resized versions created using `nearest neighbor interpolation` and `bicubic interpolation`. The original image features 8×8 grid of random pixel values. `Nearest neighbor interpolation`, while simple, produces exact replica of the original image, but each original pixel gets split into 4 identical new pixels of same RGB value. This method is less suitable for photographs with soft gradients. In contrast, `bicubic interpolation` delivers a much smoother resized image by considering the values of surrounding pixels to create new pixel values, resulting in fluid transitions and better color blending. Although it may introduce slight blurriness, bicubic interpolation generally preserves image integrity far better than `nearest neighbor interpolation`.

▼ `scale`: 2

Original Image | Nearest Neighbor Interpolation | Bicubic Interpolation

▼ `scale`: 3

Original Image | Nearest Neighbor Interpolation | Bicubic Interpolation

▼ `scale`: 0.8

Original Image | Nearest Neighbor Interpolation | Bicubic Interpolation

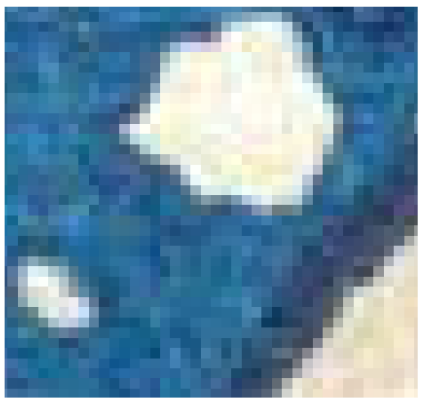
Both Methods Struggle

- Original Image

- Nearest Neighbor Interpolation
- Bicubic Interpolation

At first glance visually there might be no difference at all, but if we zoom in.

- Original Image
- Nearest Neighbor Interpolation
- Bicubic Interpolation



Both `nearest neighbor` and `bicubic interpolation` were applied to a vibrant image with distinct edges. While both methods faced challenges in accurately reproducing finer details, it was clear that `bicubic interpolation` outperformed its counterpart. Although `nearest neighbor` struggled to maintain the image's sharpness and exhibited noticeable pixelation, bicubic interpolation demonstrated a superior ability to smooth out transitions and preserve edge integrity, resulting in a more visually appealing and detailed resized image. Thus, despite the difficulties encountered by both methods, bicubic interpolation clearly emerged as the more effective choice for this particular image.

Error Estimation

For error estimation we can upscale image first by some factor and then descale it with same factor. Optimally resulting image should be same with the original one.

```
import numpy as np

def calculate_error(original_image, resized_image):
    # Ensure the images are the same shape
    if original_image.shape != resized_image.shape:
        raise ValueError("Original and resized images must have the same dimensions.")

    # Calculate the error matrix
    error_matrix = np.abs(original_image - resized_image)

    # Calculate matrix norms
    l1_norm = np.sum(error_matrix)  # L1 norm
    l2_norm = np.sqrt(np.sum(error_matrix ** 2))  # L2 norm
    infinity_norm = np.max(error_matrix)  # Infinity norm

    return l1_norm, l2_norm, infinity_norm
```

Result of downscaling and upscaling by a `scale` of 5

- Original Image
- Nearest Neighbor Interpolation
- Bicubic Interpolation

Zoomed In

- Original Image
- Nearest Neighbor Interpolation
- Bicubic Interpolation

	Nearest Neighbor	Bicubic
L1 Norm	3490930339	3446444783
L2 Norm	40580.28	38058.06
Infinity	255	255

The experiment demonstrated that downscaling an image by a `scale` of 5, followed by upscaling it by the same factor, highlighted the strengths and weaknesses of the `nearest neighbor` and `bicubic interpolation` methods. The `nearest neighbor interpolation` struggled to preserve the original pixel values, resulting in a noticeable loss of detail and quality in the resized image. In contrast, the `bicubic interpolation` method managed to approximate the original pixel values more effectively, leading to a more visually coherent result. Numerical evaluations further confirmed this observation, as the matrix norms indicated that bicubic interpolation produced significantly lower difference values compared to nearest neighbor interpolation. This suggests that bicubic interpolation is superior for maintaining image fidelity during resizing operations, especially when the goal is to preserve the nuances of the original image.

Conclusion

This study has compared `nearest neighbor` and `bicubic interpolation` methods in image resizing, focusing on their effectiveness in preserving image quality across various scaling scenarios. While `nearest neighbor interpolation` is efficient and produces clear pixelated images, it tends to struggle with smooth gradients and detail preservation, resulting in blockiness during enlargement. In contrast, `bicubic interpolation`, despite introducing slight blurriness, offers significantly smoother transitions and better color blending, maintaining finer details and edge integrity in resized images.

The experiments demonstrated that bicubic interpolation outperforms `nearest neighbor interpolation` in scenarios involving photographs and images with subtle gradients, while still facing challenges with highly detailed and vibrant images. Overall, the findings suggest that for most applications where image quality is paramount, `bicubic interpolation` is the preferred choice. However, the selection of an interpolation method should ultimately depend on the specific requirements

of the task at hand, balancing speed, quality, and the characteristics of the images being processed. Further research could explore hybrid approaches or more advanced methods to optimize image resizing in various contexts.