"Heaven's Light is Our Guide"

Department of Computer Science & Engineering

Rajshahi University of Engineering & Technology

Lab Report-2

Digital Image Processing Sessional

Course Code: CSE 4106

Submitted By:	Submitted To:
Name: Sajidur Rahman Tarafder Roll: 2003154 Section: C Session: 2020-21 Department: CSE	Khaled Zinnurine Lecturer Department of CSE, RUET

Contents

Bilinear Interpolation for Image Resizing

Objective: Implement bilinear interpolation from scratch to resize images and compare with OpenCV's built-in implementation.

1 OpenCV Bilinear Interpolation

1.1 Code Snippet

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

# Original 3x3 image
image = np.array([[50, 120, 200],
                  [80, 150, 250],
                  [30, 100, 180]], dtype=np.uint8)

# Resize using OpenCV's bilinear interpolation
interpolated_image_lf = cv2.resize(image, (6, 6),
                                   interpolation=cv2.INTER_LINEAR)

# Print both arrays
print("Original 3x3 Image:")
print(image)
print("\n6x6 Interpolated Image:")
print(interpolated_image_lf)
```

Listing 1: Using OpenCV's Built-in Bilinear Interpolation

1.2 Implementation Approach

In this first part, I used OpenCV's built-in `cv2.resize()` function with the `INTER_LINEAR` flag to perform bilinear interpolation. The function takes a 3×3 image and resizes it to 6×6. Bilinear interpolation works by taking a weighted average of the four nearest pixel values in the original image to calculate each new pixel value. OpenCV handles all the mathematical computations internally, making it very fast and efficient. This serves as our reference implementation to verify our manual implementation later.

1.3 Output

The original 3×3 image contains pixel values ranging from 30 to 250. After bilinear interpolation to 6×6, we get smoothly interpolated values that fill in the gaps between the original pixels, creating a larger version of the image with gradually changing intensities.

2 Manual Bilinear Interpolation Implementation

2.1 Code Snippet

```python
import numpy as np
import cv2

def manual_bilinear_resize(image, Hout, Wout):
    Hin, Win = image.shape
    manually_interpolated_image = np.zeros((Hout, Wout),
                                           dtype=image.dtype)

    for i in range(Hout):
        for j in range(Wout):
            # Step 1: Map output coordinates to input coordinates
            x = (j + 0.5) * Win / Wout - 0.5
            y = (i + 0.5) * Hin / Hout - 0.5

            # Step 2: Find the four surrounding pixels
            x0 = int(np.floor(x))
            y0 = int(np.floor(y))
            x1 = min(x0 + 1, Win - 1)
            y1 = min(y0 + 1, Hin - 1)

            # Step 3: Handle boundary cases
            x0 = max(x0, 0)
            y0 = max(y0, 0)

            # Step 4: Calculate interpolation weights
            dx = x - x0
            dy = y - y0

            # Get the four corner pixels
            A = image[y0, x0]
            B = image[y0, x1]
            C = image[y1, x0]
            D = image[y1, x1]

            # Horizontal interpolation
            Top = A * (1 - dx) + B * dx
            Bottom = C * (1 - dx) + D * dx

            # Vertical interpolation
            value = Top * (1 - dy) + Bottom * dy

            manually_interpolated_image[i, j] = int(value)

    return manually_interpolated_image

# Test the manual implementation
Hout = 6
```

```
Wout = 6

manually_interpolated_image = manual_bilinear_resize(image, Hout,
    Wout)

print("Original 3x3 Image:")
print(image,'\n')
print("Difference between OpenCV and Manual:")
print(interpolated_image_lf - manually_interpolated_image)
```

Listing 2: Manual Bilinear Interpolation from Scratch

2.2 Implementation Approach

I implemented bilinear interpolation completely from scratch to understand how it works internally. The algorithm has four main steps:

Step 1 - Coordinate Mapping: For each pixel in the output image, I calculate where it would be located in the original image. The formula `(j + 0.5) * Win / Wout - 0.5` maps the output coordinates back to input coordinates, using the 0.5 offset to ensure pixel centers align correctly.

Step 2 - Find Neighbors: I identify the four pixels in the original image that surround this mapped location. These are found by taking the floor of the coordinates (x0, y0) and the next pixel over (x1, y1).

Step 3 - Boundary Handling: I make sure the coordinates don't go outside the image boundaries by clamping them between 0 and the image dimensions.

Step 4 - Bilinear Interpolation: This is the heart of the algorithm. I first interpolate horizontally between the top two pixels (A and B) and between the bottom two pixels (C and D), using the weights based on how far the mapped point is between them. Then I interpolate vertically between these two horizontal results to get the final pixel value.

The weights `dx` and `dy` represent how far the mapped point is from the top-left corner, ranging from 0 to 1. When dx=0, we're exactly at the left pixel; when dx=1, we're at the right pixel; and values in between give proportional weights.

2.3 Verification

To verify my implementation is correct, I compare it with OpenCV's result by subtracting the two images. If the difference is zero (or very close to zero due to rounding), it means my manual implementation produces the same results as OpenCV's optimized version. This confirms that I've correctly implemented the bilinear interpolation algorithm.

Discussion and Conclusion

In this lab, I implemented bilinear interpolation from scratch and learned how image resizing actually works under the hood.

Bilinear interpolation is called "bilinear" because it performs linear interpolation twice - once horizontally and once vertically. It's much better than nearest-neighbor interpolation because it produces smooth results without blocky artifacts. Instead of just copying the nearest pixel value, it blends the values of the four surrounding pixels based on their distances.

The key insight I gained is understanding the coordinate mapping. When resizing from 3×3 to 6×6, each output pixel doesn't directly correspond to an input pixel. Instead, it maps to a fractional position between input pixels. The 0.5 offsets in the coordinate calculation are crucial - they ensure we're measuring from pixel centers rather than edges, which gives more accurate results.

Implementing this manually was challenging but rewarding. I had to think carefully about:

- How to map coordinates between different image sizes
- How to handle edge cases at image boundaries
- The order of interpolation operations (horizontal first, then vertical)
- Converting floating-point calculations back to integer pixel values

When I compared my manual implementation with OpenCV's built-in function, they produced identical (or nearly identical) results, which validated that I understood the algorithm correctly. The small differences, if any, are just due to rounding at different stages of the calculation.

This exercise showed me that even though libraries like OpenCV make image processing easy, understanding the underlying mathematics gives you much better control and helps when you need to implement custom variations or optimize for specific use cases. Bilinear interpolation is fundamental to many image processing operations, including image rotation, warping, and computer graphics transformations.