Morphology Operators Implementation

May 10, 2025

Abstract

This document demonstrates the implementation of basic morphology operations (erosion and dilation) using NumPy, including image padding and kernel processing. The code shows each step of the morphological operations with clear outputs.

1 Libraries Used

- `numpy`: For numerical operations and array manipulation (core library for image processing)

2 Step-by-Step Process

2.1 Step 1: Import Libraries

```
import numpy as np
```

2.2 Step 2: Define Image with numpy

Create a sample 8x8 grayscale image as a NumPy array for processing.

```
image = np.array([[10, 10, 10, 20, 10, 10, 20, 10],
                  [10, 20, 20, 20, 20, 20, 20, 10],
                  [10, 10, 10, 20, 10, 20, 20, 10],
                  [10, 10, 20, 30, 10, 20, 30, 10],
                  [10, 10, 30, 10, 10, 30, 10, 20],
                  [20, 10, 30, 10, 30, 20, 20, 10],
                  [20, 20, 20, 20, 10, 10, 20, 10],
                  [20, 20, 30, 20, 20, 30, 10, 30]], dtype=np.uint8)
```

2.3 Step 3: Reflect Padding

Add border padding to handle edge pixels during morphological operations.

```
def reflect_padding(image, padding_size):
    height, width = image.shape
    padded_image = np.zeros((height + padding_size*2, width +
       padding_size*2), dtype=image.dtype)

    # Center
    padded_image[padding_size:height+padding_size, padding_size:
       width+padding_size] = image

    # Borders
    for i in range(padding_size):
        padded_image[i, padding_size:width+padding_size] = image[0,
           :]

    for i in range(height+padding_size, height+padding_size*2):
        padded_image[i, padding_size:width+padding_size] = image[-1,
            :]

    for j in range(padding_size):
        padded_image[padding_size:height+padding_size, j] = image[:,
            0]

    for j in range(width+padding_size, width+padding_size*2):
        padded_image[padding_size:height+padding_size, j] = image[:,
            -1]

    # Corners
    padded_image[:padding_size, :padding_size] = image[0, 0]
    padded_image[:padding_size, width+padding_size:] = image[0, -1]
    padded_image[height+padding_size:, :padding_size] = image[-1, 0]
    padded_image[height+padding_size:, width+padding_size:] = image
       [-1, -1]

    return padded_image

padded_image = reflect_padding(image, 1)
```

```
[[10 10 10 10 20 10 10 20 10 10]
 [10 10 10 10 20 10 10 20 10 10]
 [10 10 20 20 20 20 20 20 10 10]
 [10 10 10 10 20 10 20 20 10 10]
 [10 10 10 20 30 10 20 30 10 10]
 [10 10 10 30 10 10 30 10 20 20]
 [20 20 10 30 10 30 20 20 10 10]
 [20 20 20 20 20 10 10 20 10 10]
 [20 20 20 30 20 20 30 10 30 30]
 [20 20 20 30 20 20 30 10 30 30]]
```

Figure 1: Padded image output (10x10 array)

2.4 Step 4: Define Kernel

Create a 3x3 kernel for morphological operations.

```
kernel = np.array([[1, 1, 1],
                   [1, 0, 0],
                   [1, 0, 0]], dtype=np.uint8)
```

2.5 Step 5: Calculate Erosion

Implement erosion operation which shrinks bright regions in the image.

```
def calculate_erosion(image, kernel):
    height, width = image.shape
    kernel_height, kernel_width = kernel.shape
    kernel_height //= 2
    kernel_width //= 2

    result_image = np.zeros((height - 2*kernel_height, width - 2*
       kernel_width), dtype=image.dtype)
    for i in range(kernel_height, height - kernel_height):
        for j in range(kernel_width, width - kernel_width):
            matrix = image[i-kernel_height:i+kernel_height+1, j-
               kernel_width:j+kernel_width+1]
            multiplied = matrix * kernel
            multiplied = multiplied.flatten().tolist()
            while 0 in multiplied:
                multiplied.remove(0)
            result_image[i - kernel_height, j - kernel_width] = min(
               multiplied)
    return result_image

erosion = calculate_erosion(padded_image, kernel)
```

```
[[10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10]
 [10 10 10 10 20 10 10 10]
 [10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10]
 [20 20 20 10 10 10 10 10]]
```

Figure 2: Erosion result (8x8 array)

2.6 Step 6: Calculate Dilation

Implement dilation operation which expands bright regions in the image.

```python
def calculate_dilation(image, kernel):
    height, width = image.shape
    kernel_height, kernel_width = kernel.shape
    kernel_height //= 2
    kernel_width //= 2

    result_image = np.zeros((height - 2*kernel_height, width - 2*
       kernel_width), dtype=image.dtype)
    for i in range(kernel_height, height - kernel_height):
        for j in range(kernel_width, width - kernel_width):
            matrix = image[i-kernel_height:i+kernel_height+1, j-
               kernel_width:j+kernel_width+1]
            multiplied = matrix * kernel
            multiplied = multiplied.flatten().tolist()
            while 0 in multiplied:
                multiplied.remove(0)
            result_image[i - kernel_height, j - kernel_width] = max(
               multiplied)
    return result_image

dilation = calculate_dilation(padded_image, kernel)
```

```
[[10 10 20 20 20 20 20 20]
 [10 10 20 20 20 20 20 20]
 [20 20 20 20 30 20 20 30]
 [10 10 20 30 30 20 30 30]
 [20 20 30 30 30 30 30 30]
 [20 30 30 30 30 30 30 20]
 [20 30 30 30 30 30 30 20]
 [20 20 20 30 20 20 30 20]]
```

Figure 3: Dilation result (8x8 array)

3 Technical Explanations

3.1 Morphological Operations

- **Erosion**: Reduces bright regions by taking the minimum value under the kernel
- **Dilation**: Expands bright regions by taking the maximum value under the kernel
- **Kernel**: A small matrix used to probe the image (structuring element)
- **Padding**: Necessary to handle border pixels during convolution

3.2 Implementation Notes

- The kernel is asymmetric (non-uniform weights)
- Zero values in kernel are ignored in min/max calculations
- Reflect padding helps reduce border artifacts
- Operations are implemented manually for educational purposes

GitHub

https://github.com/AsadiAhmad/Morphology-Operators