Documentation of Template Matching Using SAD and NCC

Contents

1 Introduction

Template matching is a technique in computer vision that involves locating a smaller image (the template) within a larger one (the target image). This documentation covers an implementation that uses two distinct similarity measures: Sum of Absolute Differences (SAD) and Normalized Cross-Correlation (NCC). These metrics serve as quantitative indicators of how well a candidate region in the image matches the template. In many applications including object detection, recognition, and tracking, template matching provides an effective solution when the object of interest is known in advance.

2 Similarity Measures in Template Matching

2.1 Sum of Absolute Differences (SAD)

The Sum of Absolute Differences quantifies the similarity between a template and a candidate image region by computing the sum of the absolute differences of the pixel intensities. The SAD formula is:

$$\mathrm{SAD} = \sum_{i,j} |I(i,j) - T(i,j)| \tag{1}$$

where:

- $I(i,j)$ is the pixel intensity at position (i,j) in the candidate region.
- $T(i,j)$ is the pixel intensity at the corresponding position in the template.

A lower SAD value indicates a better match since it means the candidate region closely resembles the template.

2.2 Normalized Cross-Correlation (NCC)

Normalized Cross-Correlation is a measure that computes the degree of similarity while accounting for differences in brightness and contrast between the template and the candidate region. Its formula is:

$$\mathrm{NCC} = \frac{\sum_{i,j} (I(i,j) - \mu_I)(T(i,j) - \mu_T)}{\sqrt{\sum_{i,j} (I(i,j) - \mu_I)^2 \sum_{i,j} (T(i,j) - \mu_T)^2}} \tag{2}$$

where:

- μ_I is the mean intensity of the candidate region.
- μ_T is the mean intensity of the template.

NCC yields values in the range $[-1,1]$, with values closer to 1 indicating a high degree of similarity. Because it normalizes the pixel values, NCC is robust against local changes in brightness or contrast.

2.3 Role in Template Matching

Both SAD and NCC have their own advantages:

- **SAD** is computationally efficient and works well when lighting conditions are consistent.
- **NCC** provides robustness by normalizing the candidate region and template, allowing for effective matching under varying illumination and contrast conditions.

These measures are critical in template matching as they quantify the similarity between the template and sub-regions of the target image.

3 Code Breakdown and Detailed Explanation

3.1 Function: `my_convolve2d`

Purpose: Performs a 2D convolution between a given image and a template (or kernel) using zero padding. This operation is used for tasks such as filtering or pre-processing.

Key Steps and Quantization:

- **Template Flipping:** The template is flipped horizontally and vertically.
- **Zero Padding:** The image is padded with zeros to allow the template to slide over the borders.
- **Sliding Window (Quantized by Pixels):** Iterates over each discrete pixel position and computes the sum of element-wise multiplications.

Listing 1: Function: my_convolve2d

```python
def my_convolve2d(image, temp):
    """
    Perform a 2D convolution between an image and a template.
    Implements convolution with zero padding.
    """
    # Flip the template (convolution operation)
    temp = np.flipud(np.fliplr(temp))

    # Get image and template dimensions
    iH, iW = image.shape
    kH, kW = temp.shape

    # Calculate padding size (assuming odd kernel size)
    pad_h = kH // 2
    pad_w = kW // 2

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)),
        mode="constant", constant_values=0)

    # Initialize the output image
    output = np.zeros((iH, iW), dtype=np.float64)

    # Perform convolution
    for i in range(iH):
        for j in range(iW):
            # Extract the region of the padded image
            region = padded_image[i:i+kH, j:j+kW]
            # Compute the convolution sum
            output[i, j] = np.sum(region * temp)

    return output
```

3.2 Function: `gaussian_kernel`

Purpose: Generates a Gaussian kernel for smoothing operations. The kernel size is determined by the standard deviation (σ), ensuring approximately 99.7% of the Gaussian's energy is captured.

Key Steps:

- **Kernel Size Calculation (Quantization):**

$$\text{kernel\_size} = 2 \times \text{int}(3 \times \sigma) + 1$$

- **Coordinate Grid Creation:** A grid is centered at zero.
- **Kernel Computation:** The Gaussian function is applied and then normalized.

Listing 2: Function: gaussian$_kernel$

```
def gaussian_kernel(kernel_size, sigma):
    """
    Return a Gaussian kernel of appropriate size.
    The kernel is of size wxw, where w = 2*int(3*sigma)+1.
    Coefficients are normalized to sum to 1.
    """
    # Determine kernel_size: Use 3 std on each side.
    kernel_size = 2 * int(3 * sigma) + 1
    center = kernel_size // 2
    # Create a coordinate grid (centered at zero)
    x = np.arange(kernel_size) - center
    y = np.arange(kernel_size) - center
    xx, yy = np.meshgrid(x, y)
    # Compute the Gaussian function
    kernel = np.exp(-(xx**2 + yy**2) / (2 * sigma**2))
    # Normalize the kernel
    kernel = kernel / kernel.sum()
    return kernel
```

3.3 Function: `temp_sad`

Purpose: Implements template matching using the Sum of Absolute Differences (SAD). The function slides the template over the input image and calculates the SAD at each position.

Key Steps:

- **Sliding Window:** The template is moved over every possible pixel position.
- **SAD Calculation:** For each position, the sum of absolute differences between the template and candidate region is computed.
- **Best Match Selection:** Records the location with the minimum SAD as the best match.

Listing 3: Function: temp$_sad$

```
def temp_sad(image, temp):
    """
    Template Matching using Sum of Absolute Differences (SAD).

    Inputs:
        image: Main image (grayscale).
        temp: Template image (grayscale).

    Output:
        (tx, ty): Coordinates of the top-left corner of the best match.
    """
    H, W = image.shape
    h, w = temp.shape
    best_cost = np.inf  # Lower cost indicates a better match
    best_x, best_y = 0, 0

    # Slide the template over every possible position in the image
    for i in range(H - h + 1):
        for j in range(W - w + 1):
            # Extract candidate region
            candidate = image[i:i+h, j:j+w]
            # Compute SAD
            cost = np.sum(np.abs(candidate - temp))
            if cost < best_cost:
                best_cost = cost
                best_x, best_y = i, j
                print('Found␣template␣at␣(' + repr(best_x) + ',' +
                    repr(best_y) +
                      ')␣with␣cost␣' + repr(best_cost))
    return best_x, best_y
```

3.4 Function: `temp_ncc`

Purpose: Performs template matching using Normalized Cross-Correlation (NCC). It evaluates similarity by accounting for local intensity variations.

Key Steps:

- **Sliding Window:** The template is moved over every pixel position.
- **Local Statistics Computation:** Computes mean and standard deviation for the candidate patch and template.
- **NCC Calculation:** Uses the formula:

$$\text{NCC} = \frac{\sum_{i,j} (T(i,j) - \mu_T)\,(I(i,j) - \mu_I)}{\sqrt{\sum_{i,j} (T(i,j) - \mu_T)^2 \ \sum_{i,j} (I(i,j) - \mu_I)^2}}$$

- **Best Match Selection:** Chooses the candidate with the highest NCC score.

Listing 4: Function: temp$_{n}cc$

```
def temp_ncc(image, temp):
    """
    Template matching using Normalized Cross-Correlation (NCC).

    Inputs:
        image: Main image (grayscale).
        temp: Template image (grayscale).

    Output:
        (tx, ty): Coordinates of the top-left corner of the best match.
    """
    H, W = image.shape
    h, w = temp.shape
    best_score = -np.inf
    best_x, best_y = 0, 0

    # Precompute template statistics
    T_mean = np.mean(temp)
    T_std = np.std(temp)
    if T_std == 0:
        T_std = 1e-10

    # Slide the template over every possible position
    for i in range(H - h + 1):
        for j in range(W - w + 1):
            candidate = image[i:i+h, j:j+w]
            I_mean = np.mean(candidate)
            I_std = np.std(candidate)
            if I_std == 0:
                I_std = 1e-10
            # Compute normalized cross-correlation
            numerator = np.sum((temp - T_mean) * (candidate - I_mean))
            denominator = np.sqrt(np.sum((temp - T_mean)**2) *
               np.sum((candidate - I_mean)**2))
            score = numerator / denominator
            if score > best_score:
                best_score = score
                best_x, best_y = j, i  % (column, row) ordering
                print('Found␣template␣at␣(' + repr(best_x) + ',' +
                   repr(best_y) +
                      ')␣with␣score␣' + repr(best_score))
    return best_x, best_y
```

3.5 Function: `imshow`

Purpose: Displays an image using Matplotlib after converting it to the RGB color space for proper display.

Key Steps:

- Converts the image from BGR or grayscale to RGB.
- Uses flexible argument passing (`*args` and `**kwargs`) for customization.
- Disables axis markers for clarity.

Listing 5: Function: imshow

```
def imshow(image, *args, **kwargs):
    """
    Display an image using Matplotlib.
    Converts BGR images (or grayscale) to RGB for correct display.
    """
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    else:
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)
    plt.imshow(image, *args, **kwargs)
    plt.axis('off')
    plt.show()
```

3.6 Main Workflow and Multi-Scale Template Matching

The main body of the code performs the following operations:

1. **Image Loading and Preprocessing:**

- The template and input images are loaded and converted to grayscale.

2. **Single-Scale Template Matching:**
   - Template matching is performed using both SAD and NCC to locate the best match in the input image.
3. **Multi-Scale Template Matching:**
   - A small template and a corresponding multi-scale background image are loaded.
   - The small template is resized to create medium and large templates.
   - A filtered image is obtained by convolving the image with a Gaussian kernel.
   - Template matching is applied at different scales to detect the object at various resolutions.
4. **Visualization:**
   - Detected matches are highlighted by drawing colored rectangles over the matched regions.

Listing 6: Drawing rectangles and visualizing results

```
def draw_rectangle(image, top_left, template_size, color=(255, 0, 0)):
    """
    Draws a rectangle around the matched template in the image.
    """
    h, w = template_size
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv2.rectangle(image, top_left, bottom_right, color, 3)

# Convert image to RGB for visualization
output_image = cv2.cvtColor(map_image, cv2.COLOR_GRAY2BGR)

% Draw matches with different colors indicating scale:
% Green for small-scale, Blue for medium-scale, Red for large-scale.
draw_rectangle(output_image, best_match, smalltempl_image.shape, (0,
    255, 0))
draw_rectangle(output_image, medium_match, medium_template.shape,
    (255, 0, 0))
draw_rectangle(output_image, large_match, large_template.shape, (0, 0,
    255))

% Display the result
plt.figure(figsize=(6, 6))
plt.imshow(output_image, cmap='gray')
plt.title("Detected␣Templates")
plt.axis("off")
plt.show()
```

4 Results and Visualizations

The following figures show the images used in the template matching process:

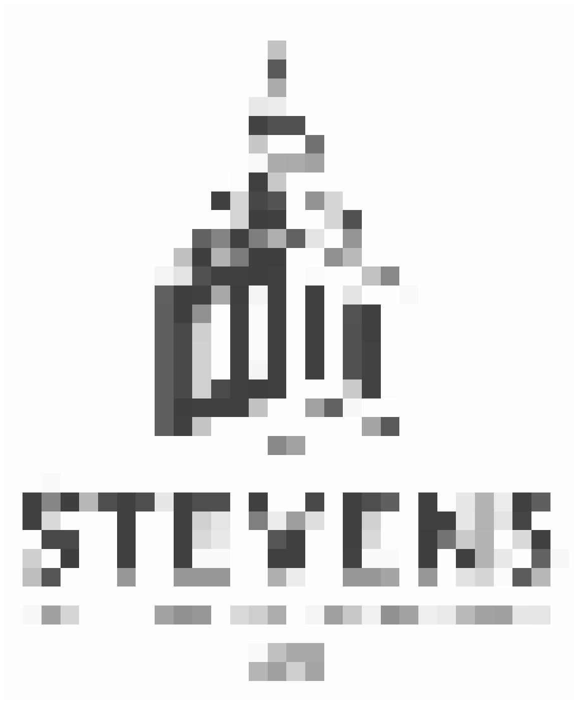

Figure 1: Template Image

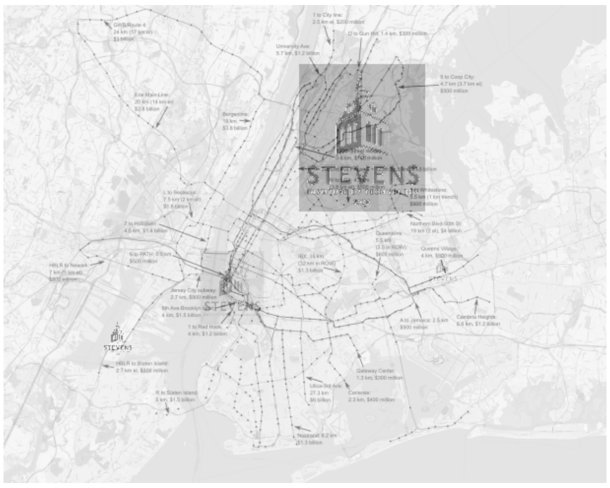

Figure 2: Input Image

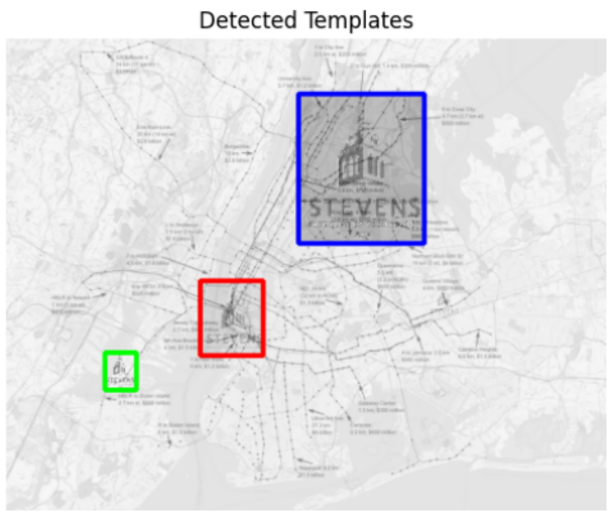

Figure 3: Output Image with Detected Matches

5 Conclusion

This documentation has provided a detailed overview of the implementation of template matching using SAD and NCC. It includes a theoretical explanation of the similarity measures with their respective formulas, a step-by-step breakdown of the code, and visual results demonstrating the template, input, and output images. The approach exhibits robustness by accommodating multi-scale matching and normalization techniques.