Computer Vision PA #1 Report

20195159 Wooseok Jeon

Abstract

This report is about programming assignment #1 : Multi-Image Denoising. Denoising is a technique for digital image processing: removing unwanted noise from a digital picture without losing the important details. The primary objective of PA1 is to address severe noise in multiple images. This is achieved by constructing a cost volume through the disparities between these images. The Semi Global Matching technique is then employed to isolate and extract the noise. Finally, through processes of warping and aggregating, I aim to produce a denoised image.

1. Method & Code Implementation

1.1. Input data

In PA1, a set of seven images possessing a noise level of 25dB was provided. Additionally, image sets with noise levels of 10dB, 12.5dB, and 15dB, each comprising seven images, were also given. Images with the same noise level exhibited disparities horizontally. Using the 4th image as the reference (referred to as the ”reference image”), images 1, 2, and 3 were to its left, while images 5, 6, and 7 were to its right.

1.2. Pixelwise Similarity

To compute the pixelwise similarity when given two images as input, the following methods can be employed.

1. Sum of Absolute Differences(SAD) is defined as the sum of the absolute differences between the corresponding pixels of the two images. It is represented mathematically as:

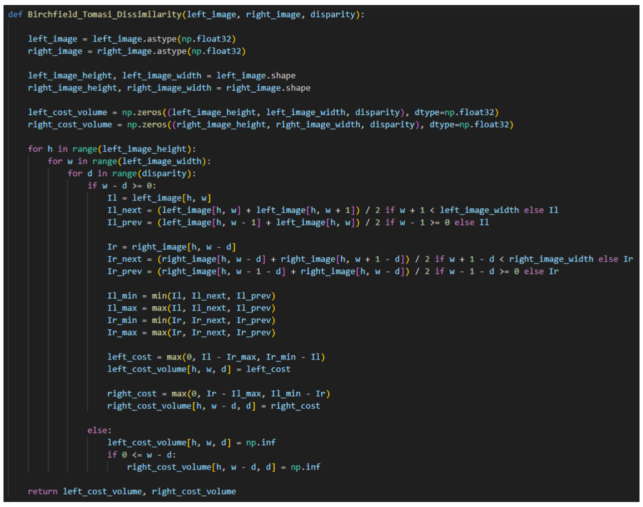
$$D(I_l, I_r) = \sum |I_l(x,y) - I_r(x,y)|$$

2. Sum of Squared Differences(SSD) is the sum of the squared differences between the same. It is represented mathematically as:

$$D(I_l, I_r) = \sum (I_l(x,y) - I_r(x,y))^2$$

3. Birchfield-Tomasi Dissimilarity was introduced by Birchfield and Tomasi as an alternative to the standard pixel matching techniques. It is a pixel dissimilarity measure proposed for stereo matching that considers values within a range that extends half a pixel on either side to minimizes sampling errors, especially when there exists slight misalignment between pixels. It considers the pixel values and their intensities in a specified manner.

I opted for the Birchfield-Tomasi Dissimilarity due to its specific advantages such as its ability to handle illumination changes or its robustness against certain types of noise. Additionally, a significant advantage lies in its capability to facilitate informed choices between the left cost volume and the right cost volume when detailed inter-image information is accessible. Below is the code implementing this:



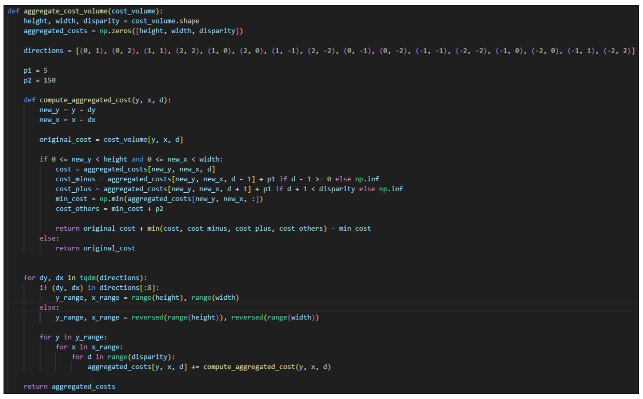
```python
def Birchfield_Tomasi_Dissimilarity(left_image, right_image, disparity):

    left_image = left_image.astype(np.float32)
    right_image = right_image.astype(np.float32)

    left_image_height, left_image_width = left_image.shape
    right_image_height, right_image_width = right_image.shape

    left_cost_volume = np.zeros((left_image_height, left_image_width, disparity), dtype=np.float32)
    right_cost_volume = np.zeros((right_image_height, right_image_width, disparity), dtype=np.float32)

    for h in range(left_image_height):
        for w in range(left_image_width):
            for d in range(disparity):
                if w - d >= 0:
                    Il = left_image[h, w]
                    Il_next = (left_image[h, w] + left_image[h, w + 1]) / 2 if w + 1 < left_image_width else Il
                    Il_prev = (left_image[h, w - 1] + left_image[h, w]) / 2 if w - 1 >= 0 else Il

                    Ir = right_image[h, w - d]
                    Ir_next = (right_image[h, w - d] + right_image[h, w + 1 - d]) / 2 if w + 1 - d < right_image_width else Ir
                    Ir_prev = (right_image[h, w - 1 - d] + right_image[h, w - d]) / 2 if w - 1 - d >= 0 else Ir

                    Il_min = min(Il, Il_next, Il_prev)
                    Il_max = max(Il, Il_next, Il_prev)
                    Ir_min = min(Ir, Ir_next, Ir_prev)
                    Ir_max = max(Ir, Ir_next, Ir_prev)

                    left_cost = max(0, Il - Ir_max, Ir_min - Il)
                    left_cost_volume[h, w, d] = left_cost

                    right_cost = max(0, Ir - Il_max, Il_min - Ir)
                    right_cost_volume[h, w - d, d] = right_cost

                else:
                    left_cost_volume[h, w, d] = np.inf
                    if 0 <= w - d:
                        right_cost_volume[h, w - d, d] = np.inf

    return left_cost_volume, right_cost_volume
```

1.3. Semi-Global Matching

In the subsequent step, the Semi-Global Matching (SGM) algorithm was applied to aggregate the cost volume. SGM is a renowned technique for calculating disparity maps, achieved by summing matching costs across various directions in an image. The following is the equation for how SGM was performed:

$$\begin{aligned}
L_r(p,d) = D(p,d) \\
&+ \min\big(L_r(p-r,d), L_r(p-r,d-1) + p_1, \\
&L_r(p-r,d+1) + p_1, \min_i\{L_r(p-r,i)\} + P_2\big) \\
&- \min_i\{L_r(p-r,i)\}
\end{aligned}$$

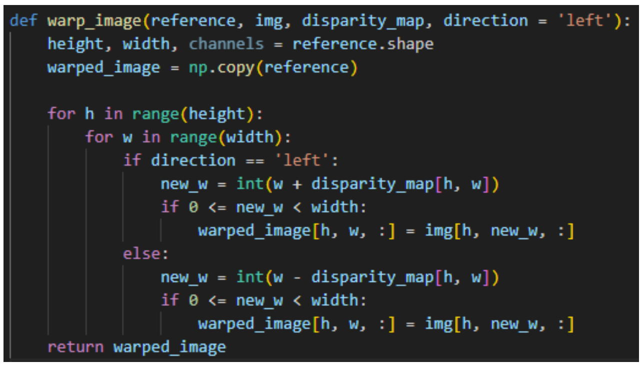
Initially conceived with 8 directions, I have expanded this to incorporate 16, effectively doubling the directional input to refine the outcome. The first 8 directions are employed during the forward pass and the remaining 8 in the backward pass, ensuring a comprehensive and detailed mapping. Hyperparameters(p1, p2) were initialized to 5 and 150 respectively, providing a balanced approach to disparity computation. Also, I used dynamic programming to calculate disparity map, avoiding redundant computations by storing and reusing processed data. The process culminates in the derivation of a disparity map via the SGM method, as illustrated in the following code:



```python
def aggregate_cost_volume(cost_volume):
    height, width, disparity = cost_volume.shape
    aggregated_costs = np.zeros([height, width, disparity])

    directions = [(0, 1), (0, 2), (1, 1), (2, 2), (1, 0), (2, 0), (1, -1), (2, -2), (0, -1), (0, -2), (-1, -1), (-2, -2), (-1, 0), (-2, 0), (-1, 1), (-2, 2)]

    p1 = 5
    p2 = 150

    def compute_aggregated_cost(y, x, d):
        new_y = y - dy
        new_x = x - dx

        original_cost = cost_volume[y, x, d]

        if 0 <= new_y < height and 0 <= new_x < width:
            cost = aggregated_costs[new_y, new_x, d]
            cost_minus = aggregated_costs[new_y, new_x, d - 1] + p1 if d - 1 >= 0 else np.inf
            cost_plus = aggregated_costs[new_y, new_x, d + 1] + p1 if d + 1 < disparity else np.inf
            min_cost = np.min(aggregated_costs[new_y, new_x, :])
            cost_others = min_cost + p2

            return original_cost + min(cost, cost_minus, cost_plus, cost_others) - min_cost
        else:
            return original_cost


    for dy, dx in tqdm(directions):
        if (dy, dx) in directions[:8]:
            y_range, x_range = range(height), range(width)
        else:
            y_range, x_range = reversed(range(height)), reversed(range(width))

        for y in y_range:
            for x in x_range:
                for d in range(disparity):
                    aggregated_costs[y, x, d] += compute_aggregated_cost(y, x, d)

    return aggregated_costs
```

1.4. Warping

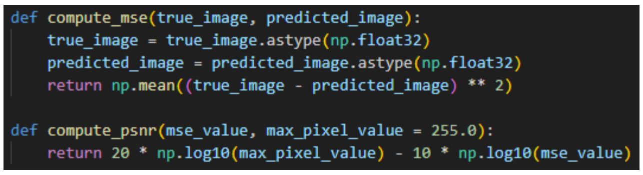
Warping is the process of transforming or distorting an image, typically through geometric changes like stretching, rotating, or skewing. Warping with disparity involves adjusting the position of pixels in an image based on calculated disparity values. The warping process was conducted using the extracted 7 disparity maps, fetching information from the 7 original images based on the disparity map information of the reference image. This warping method, which is based on a single image, is much more efficient in terms of information loss compared to warping each individual image. At this point, the warping method was varied based on whether it was a left cost volume or a right cost volume, utilizing the directional information accordingly. The warped images were averaged based on the RGB axis. Also, due to the loss of information of d pixels at the cost volume borders, d pixels were cropped from all edges.



```python
def warp_image(reference, img, disparity_map, direction = 'left'):
    height, width, channels = reference.shape
    warped_image = np.copy(reference)

    for h in range(height):
        for w in range(width):
            if direction == 'left':
                new_w = int(w + disparity_map[h, w])
                if 0 <= new_w < width:
                    warped_image[h, w, :] = img[h, new_w, :]
            else:
                new_w = int(w - disparity_map[h, w])
                if 0 <= new_w < width:
                    warped_image[h, w, :] = img[h, new_w, :]
    return warped_image
```

1.5. Metric

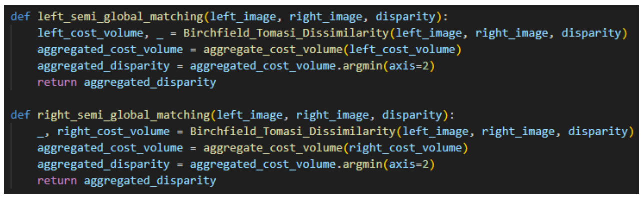
The denoising process's performance is evaluated using two common metrics: Mean Squared Error (MSE) and Peak Signal to Noise Ratio (PSNR). Pixel values of images loaded using cv2 are in unsigned int format, which can lead to overflow during calculations. Therefore, I have implemented a step to convert them to float type within the function.



```python
def compute_mse(true_image, predicted_image):
    true_image = true_image.astype(np.float32)
    predicted_image = predicted_image.astype(np.float32)
    return np.mean((true_image - predicted_image) ** 2)

def compute_psnr(mse_value, max_pixel_value = 255.0):
    return 20 * np.log10(max_pixel_value) - 10 * np.log10(mse_value)
```

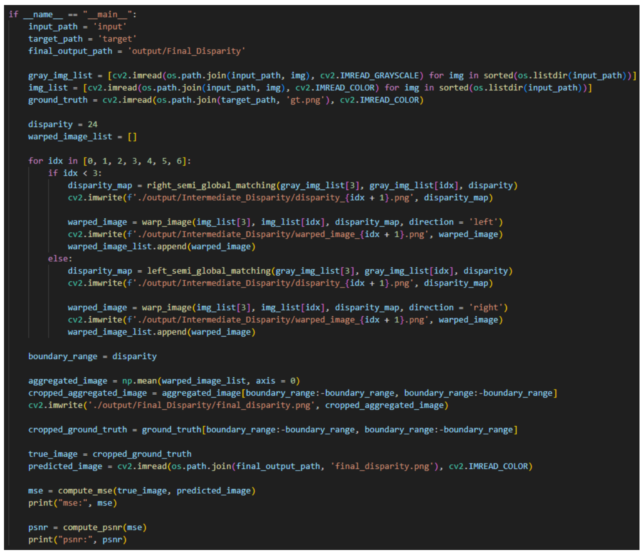
2. Result & Discussion

As previously mentioned, considering image 4 as the reference, images 1, 2, and 3 are located to the left, while images 5, 6, and 7 are to the right. Therefore, when extracting the Cost Volume using the Birchfield-Tomasi Dissimilarity, different cost volumes must be extracted for each. For the cost volume between image 4 and images 1, 2, and 3, the Right Cost Volume needs to be extracted. On the other hand, for the remaining images, the Left Cost Volume should be extracted before deriving the disparity map. Accordingly, two distinct functions were defined for these purposes.



```python
def left_semi_global_matching(left_image, right_image, disparity):
    left_cost_volume, _ = Birchfield_Tomasi_Dissimilarity(left_image, right_image, disparity)
    aggregated_cost_volume = aggregate_cost_volume(left_cost_volume)
    aggregated_disparity = aggregated_cost_volume.argmin(axis=2)
    return aggregated_disparity

def right_semi_global_matching(left_image, right_image, disparity):
    _, right_cost_volume = Birchfield_Tomasi_Dissimilarity(left_image, right_image, disparity)
    aggregated_cost_volume = aggregate_cost_volume(right_cost_volume)
    aggregated_disparity = aggregated_cost_volume.argmin(axis=2)
    return aggregated_disparity
```

Now that all the required functions have been implemented, the input images were loaded separately in grayscale and color. The disparity value was initialized to 24, and then, appropriate SGM functions were applied to the reference image and each of the other images, including the reference image. After processing, they were warped and saved. Subsequently, the average of the warped set of 7 images was computed, cropped to yield the final image, and both MSE and PSNR values were calculated. The following is the code that encompasses all these steps.



```python
if __name__ == "__main__":
    input_path = 'input'
    target_path = 'target'
    final_output_path = 'output/Final_Disparity'

    gray_img_list = [cv2.imread(os.path.join(input_path, img), cv2.IMREAD_GRAYSCALE) for img in sorted(os.listdir(input_path))]
    img_list = [cv2.imread(os.path.join(input_path, img), cv2.IMREAD_COLOR) for img in sorted(os.listdir(input_path))]
    ground_truth = cv2.imread(os.path.join(target_path, 'gt.png'), cv2.IMREAD_COLOR)

    disparity = 24
    warped_image_list = []

    for idx in [0, 1, 2, 3, 4, 5, 6]:
        if idx < 3:
            disparity_map = right_semi_global_matching(gray_img_list[3], gray_img_list[idx], disparity)
            cv2.imwrite(f'./output/Intermediate_Disparity/disparity_{idx + 1}.png', disparity_map)

            warped_image = warp_image(img_list[3], img_list[idx], disparity_map, direction = 'left')
            cv2.imwrite(f'./output/Intermediate_Disparity/warped_image_{idx + 1}.png', warped_image)
            warped_image_list.append(warped_image)
        else:
            disparity_map = left_semi_global_matching(gray_img_list[3], gray_img_list[idx], disparity)
            cv2.imwrite(f'./output/Intermediate_Disparity/disparity_{idx + 1}.png', disparity_map)

            warped_image = warp_image(img_list[3], img_list[idx], disparity_map, direction = 'right')
            cv2.imwrite(f'./output/Intermediate_Disparity/warped_image_{idx + 1}.png', warped_image)
            warped_image_list.append(warped_image)

    boundary_range = disparity

    aggregated_image = np.mean(warped_image_list, axis = 0)
    cropped_aggregated_image = aggregated_image[boundary_range:-boundary_range, boundary_range:-boundary_range]
    cv2.imwrite('./output/Final_Disparity/final_disparity.png', cropped_aggregated_image)

    cropped_ground_truth = ground_truth[boundary_range:-boundary_range, boundary_range:-boundary_range]

    true_image = cropped_ground_truth
    predicted_image = cv2.imread(os.path.join(final_output_path, 'final_disparity.png'), cv2.IMREAD_COLOR)

    mse = compute_mse(true_image, predicted_image)
    print("mse:", mse)

    psnr = compute_psnr(mse)
    print("psnr:", psnr)
```

In the evaluation of denoising process, I assessed the quality and accuracy of the resulted images through the Mean Squared Error (MSE) and the Peak Signal-to-Noise Ratio (PSNR). The computations yielded a PSNR value of approximately 24.7. Although this falls slightly short of the benchmark 25 as mentioned in the PA guidelines, it shows the efficacy of my denoising technique.

Below is the final result image, obtained after the denoising and cropping processes.