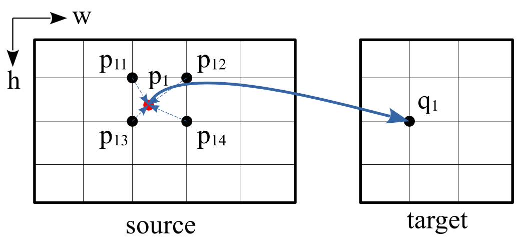
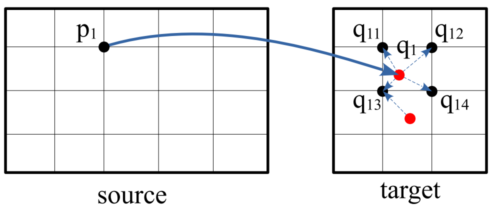
Implementation Notes: Differentiable Image Warping

Zhiwei Xu
zwxu064@gmail.com

originate: 16 August 2021
modify: 6 September 2024



(a) Case 1



(b) Case 2

Figure 1: Image warping. (a) The target image is sourced from weighted sum of neighbouring pixels in the source image, which can be achieved using torch.linspace() and torch.grid_sample() following Eq. (1). (b) The target image is sourced from weighted sum of neighbouring pixels projected from the source image, see Eq. (4). Disparity maps of these two cases can be learned for either the source image or the target image. Generally, the target image in Case 2 has less unassigned/void pixels than that in Case 1, and thus, Case 2 is preferred for the main implementation. Differentiation of the disparity map is enabled by the coordinate weight $g(p_i, p_{ij})$ in Eq. (2), instead of rounding p_i to its nearest pixel.

Case 1. Given Fig. (1a), the forward and backward propagation are

$$f_t(q_i) = f_s(q_i + d_i) = f_s(p_i) = \sum_{j \in \mathcal{E}_i} g(p_i, p_{ij}) f_s(p_{ij}) \ , \tag{1}$$

$$g(p_i, p_{ij}) = g_h(p_i, p_{ij}) g_w(p_i, p_{ij}) = \left(1 - \left|h_{p_i} - h_{p_{ij}}\right|\right)\left(1 - \left|w_{p_i} - w_{p_{ij}}\right|\right) \ , \tag{2}$$

where $f_s(\cdot)$ and $f_t(\cdot) : \mathbb{R}^2 \mapsto \mathbb{R}$ are mapping functions for pixel intensity of the source image and the target image, respectively, pixel coordinate $q_i \in \mathbb{R}^2$, and disparity $d_i \in \mathbb{R}^2$. With loss L, the gradients of d_i follows the chain rule,

$$\frac{\nabla L}{\nabla d_i} = \frac{\partial L}{\partial f_t(q_i)} \frac{\partial f_t(q_i)}{\partial d_i} = \frac{\partial L}{\partial f_t(q_i)} \frac{\partial f_t(q_i)}{\partial f_s(p_i)} \frac{\partial f_s(p_i)}{\partial d_i} = \sum_{j \in \mathcal{E}_i} f_s(p_{ij}) \frac{\partial L}{\partial f_t(q_i)} \frac{\partial f_t(q_i)}{\partial f_s(p_i)} \frac{\partial g(p_i, p_{ij})}{\partial d_i} \ , \tag{3}$$

where $\partial g(p_i, p_{ij}) / \partial d_i \in \mathbb{R}^2$ can be easily computed.

Case 2. Instead of projecting weighted sum of pixel intensities from the source image to the target image, Case 2 projects the source image to the target image. Applying Eq. (1) to Fig. (1b) with the disparity map for the source image,

$$f_t(q_{ij}) = \frac{1}{\sum_{i\in\mathcal{E}_j} g(q_i, q_{ij})} \sum_{i\in\mathcal{E}_j} g(q_i, q_{ij}) f_t(q_i) \ . \tag{4}$$

Then,

$$\begin{aligned}\frac{\nabla L}{\nabla d_i} &= \sum_{j\in\mathcal{E}_i} \frac{\partial L}{\partial f_t(q_{ij})} \frac{\partial f_t(q_{ij})}{\partial d_i} = \sum_{j\in\mathcal{E}_i} \frac{\partial L}{\partial f_t(q_{ij})} \frac{\partial f_t(q_{ij})}{\partial g(q_i, q_{ij})} \frac{\partial g(q_i, q_{ij})}{\partial d_i} \\ &= \sum_{j\in\mathcal{E}_i} \frac{\partial L}{\partial f_t(q_{ij})} \frac{f_t(q_i) - f_t(q_{ij})}{\sum_{i\in\mathcal{E}_j} g(q_i, q_{ij})} \frac{\partial g(q_i, q_{ij})}{\partial d_i} \ .\end{aligned} \tag{5}$$

One can decompose $f_t(q_{ij})$ into $\sum_{i\in\mathcal{E}_j} g(q_i, q_{ij})$ and $\sum_{i\in\mathcal{E}_j} g(q_i, q_{ij}) f_t(q_i)$ to calculate partial derivatives on d_i separately, followed by an accumulation.

To simplify this equation, one can average over the number of neighbouring pixels instead of their coordinate weights, giving

$$f_t(q_{ij}) = \frac{1}{|\mathcal{E}_j|} \sum_{i\in\mathcal{E}_j} g(q_i, q_{ij}) f_t(q_i) \ . \tag{6}$$

Consequently, $\nabla L/\nabla d_i$ can also be simplified.

Remark. Differentiating the image warping on d_i is primarily infeasible because d_i represents real-valued coordinate shifts, which forms the index of pixel after rounding yet directly affecting the pixel intensity. However, imposing coordinate weights using $g(\cdot, \cdot)$ in Eq. (2) associates the real-valued disparity values with the projected pixel intensities, and thus, can be differentiated.

Implementation in CUDA. The computation of pixel projection from the source image to the target image and the coordinate weights can be parallelized. After that, for Eq. (6), the averaging can be performed after using the thread synchronization in CUDA programming. The backward propagation can also be parallelized in the same approach.