# 深度学习实现单目三维重建

统计学 张笑晗

## 一、三维重建与深度学习

三维重建是近期计算机视觉的热门方向，是AR，元宇宙，自动驾驶，摄影测量等应用领域核心技术之一。在 2018 年之前是用传统方法实现的，是通过手工精心的去做出相似的指标，或者设计相关规则，然后计算出高密度的 3D 点云之间的关系。这些方法，在理想的场景下展现出很高的准确率，但是他们都有一个共同的局限性，容易受到低纹理，反光和局部反射的干扰，导致匹配很艰难，从而造成重构的 3D 立体或者深度图不完整。虽然现在有的算法，在准确率上已经很高了，但是在重构完整性这个方面，还有很大的改善空间。

在2018年之后出现了深度学习的方法，加强了重建的纹理特征。理论上来说，深度学习的方法带有全局的语义信息，让它学习到高光，或者反射的信息，以至于更好的去进行二维到三维的匹配。这种方法展示出令人满意的结果，并且慢慢的超出了传统的方法。本文选择了几个在三维重建中表现优异的网络结构和方法进行叙述。

## 二、三维重建的评价指标[1]

三维重建首先估计每个视角图片的深度图，再将所有的深度图投影到空间形成点云，重建点云与真实点云相比较评价三维重建的好坏，评价指标如下：

### 准确性

设 $\mathcal{G}$ 为真实点云，$\mathcal{R}$ 为重建点云，对于重建点云中的一个点 $\boldsymbol{r} \in \mathcal{R}$，与真实点云的距离定义为：

$$\boldsymbol{e}_{r\to\mathcal{G}} = \underset{\boldsymbol{g}\in\mathcal{G}}{min} \parallel \boldsymbol{r} - \boldsymbol{g} \parallel \tag{1}$$

准确性定义为：

$$P(d) = \frac{100}{|\mathcal{R}|}\sum_{\boldsymbol{r}\in\mathcal{R}}[\boldsymbol{e}_{r\to\mathcal{G}} < d] \tag{2}$$

$[\cdot]$ 表示满足括号内要求时为1，否则为0。$d$ 是阈值。该式的意义是统计重建点云和真实点云距离不超过 $d$ 的重建点云的点数占重建点云点数的百分比。

### 完整性

同样地，设真实点云中的一点 $g \in \mathcal{G}$，定义其到重建点云的距离：

$$\boldsymbol{e}_{g\to\mathcal{R}} = \underset{\boldsymbol{r}\in\mathcal{R}}{min} \parallel \boldsymbol{g} - \boldsymbol{r} \parallel \tag{3}$$

完整性定义为：

$$R(d) = \frac{100}{|\mathcal{G}|}\sum_{\boldsymbol{g}\in\mathcal{G}}[\boldsymbol{e}_{g\to\mathcal{R}} < d] \tag{4}$$

如果重建点云是残缺的，满足 $[\boldsymbol{e}_{g\to\mathcal{R}} < d] = 1$ 的点就会少，导致完整性下降。

### 综合指标

希望准确性和完整性越大越好，定义 $F$：

$$F(d) = \frac{2P(d)R(d)}{P(d) + R(d)} \tag{5}$$

$F$ 可以保证不会出现一个指标很大而另一个指标很小的情况，如果 $P(d) \to 0$ 或 $R(d) \to 0$，则 $F \to 0$

## 三、深度学习方法的最新成果

### MVSNet--深度学习方法的理论奠基[2]

MVSNet 是第一个完全用深度学习实现端到端的三维重建的网络，奠定了后续方法的框架基础，其结构如下：

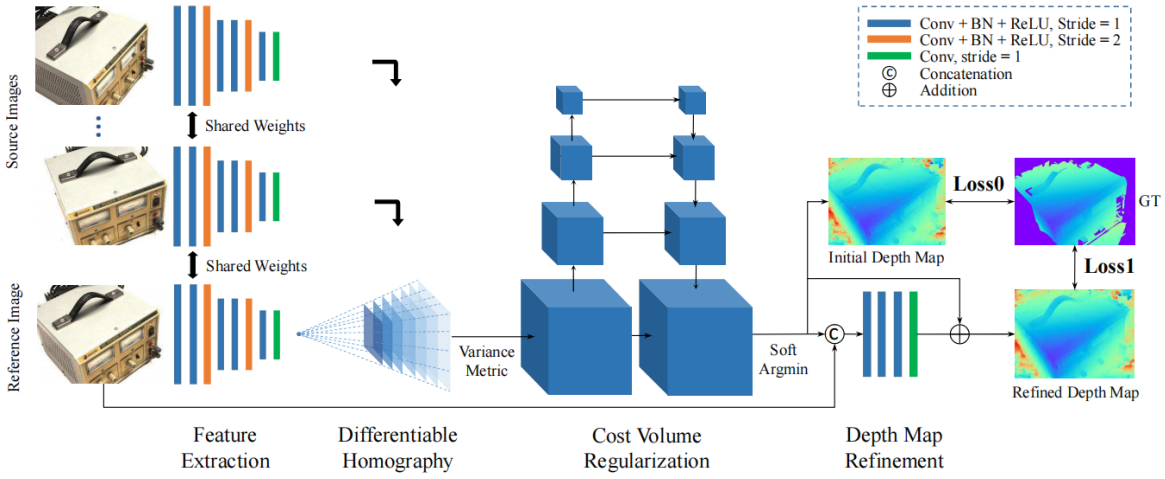


图1

网络的输入为一个场景多个视角的图片（因为单个视角会丢失图像的深度，需要用多个视角才可以恢复出深度），分为一个参考帧和其余的邻域帧。网络的输出为参考帧的深度图。多个视角的图片经过共享权重的 2D CNN 进行特征提取（充分融合邻域的信息），将原来三通道的图像转换为多通道的深度图。在空间中划分深度，在每一层上对邻域帧每个像素像素做单应变换（$Differentiable\ Homography$）后转换到参考帧的对应像素上，单应变换推导如下：

单应矩阵通常描述处于共同平面上的一些点在两张图像之间的变换关系。设图像 $I_1$ 和 $I_2$ 有一对匹配好的特征点，这个特征点落在平面 $P$ 上，设这个平面满足方程：

$$\boldsymbol{n}^\top \boldsymbol{P} + d = 0 \tag{6}$$

整理得

$$-\frac{\boldsymbol{n}^\top \boldsymbol{P}}{d} = 1 \tag{7}$$

设 $I_1$ 到 $I_2$ 的坐标变换为旋转矩阵 $R$ 和平移矩阵 $t$，相机内参为 $K$，特征点在 $I_1$ 和 $I_2$ 下的深度为 $d_1$，$d_2$，有

$$\begin{aligned} d_2\boldsymbol{p_2} &= \boldsymbol{K}(\boldsymbol{RP} + \boldsymbol{t}) \\ &= \frac{1}{d_2}\boldsymbol{K}(\boldsymbol{RP} + \boldsymbol{t}(-\frac{\boldsymbol{n}^\top \boldsymbol{P}}{d}) \\ &= \frac{1}{d_2}\boldsymbol{K}(\boldsymbol{R} - \frac{\boldsymbol{tn}^\top}{d})\boldsymbol{P} \\ &= \frac{d_1}{d_2}\boldsymbol{K}(\boldsymbol{R} - \frac{\boldsymbol{tn}^\top}{d})\boldsymbol{K}^{-1}\boldsymbol{p_1} \end{aligned} \tag{8}$$

于是得到了描述像素 $p_1$，$p_2$ 之间得变换，记为 $H$，于是：

$$\boldsymbol{p_2} = \frac{d_1}{d_2}\boldsymbol{H}\boldsymbol{p_1} \tag{9}$$

代码实现：

```python
def differentiable_warping(
    src_fea: torch.Tensor, src_proj: torch.Tensor, ref_proj: torch.Tensor, depth_samples: torch.Tensor
):
    """Differentiable homography-based warping, implemented in Pytorch.

    Args:
        src_fea: [B, C, H, W] source features, for each source view in batch
        src_proj: [B, 4, 4] source camera projection matrix, for each source view in batch
        ref_proj: [B, 4, 4] reference camera projection matrix, for each ref view in batch
        depth_samples: [B, Ndepth, H, W] virtual depth layers
    Returns:
        warped_src_fea: [B, C, Ndepth, H, W] features on depths after perspective transformation
    """

    batch, channels, height, width = src_fea.shape
    num_depth = depth_samples.shape[1]

    with torch.no_grad():
        proj = torch.matmul(src_proj, torch.inverse(ref_proj))
        rot = proj[:, :3, :3]  # [B,3,3]
        trans = proj[:, :3, 3:4]  # [B,3,1]

        y, x = torch.meshgrid(
            [
                torch.arange(0, height, dtype=torch.float32, device=src_fea.device),
                torch.arange(0, width, dtype=torch.float32, device=src_fea.device),
            ]
        )
        y, x = y.contiguous(), x.contiguous()
        y, x = y.view(height * width), x.view(height * width)
        xyz = torch.stack((x, y, torch.ones_like(x)))  # [3, H*W]
        xyz = torch.unsqueeze(xyz, 0).repeat(batch, 1, 1)  # [B, 3, H*W]
        rot_xyz = torch.matmul(rot, xyz)  # [B, 3, H*W]

        rot_depth_xyz = rot_xyz.unsqueeze(2).repeat(1, 1, num_depth, 1) * depth_samples.view(
            batch, 1, num_depth, height * width
        )  # [B, 3, Ndepth, H*W]
        proj_xyz = rot_depth_xyz + trans.view(batch, 3, 1, 1)  # [B, 3, Ndepth, H*W]
        # avoid negative depth
        negative_depth_mask = proj_xyz[:, 2:] <= 1e-3
        proj_xyz[:, 0:1][negative_depth_mask] = width
        proj_xyz[:, 1:2][negative_depth_mask] = height
        proj_xyz[:, 2:3][negative_depth_mask] = 1
        proj_xy = proj_xyz[:, :2, :, :] / proj_xyz[:, 2:3, :, :]  # [B, 2, Ndepth, H*W]
        proj_x_normalized = proj_xy[:, 0, :, :] / ((width - 1) / 2) - 1  # [B, Ndepth, H*W]
```

```
        proj_y_normalized = proj_xy[:, 1, :, :] / ((height - 1) / 2) - 1
        proj_xy = torch.stack((proj_x_normalized, proj_y_normalized), dim=3)
 # [B, Ndepth, H*W, 2]
        grid = proj_xy

    warped_src_fea = F.grid_sample(
        src_fea,
        grid.view(batch, num_depth * height, width, 2),
        mode="bilinear",
        padding_mode="zeros",
        align_corners=True,
    )

    warped_src_fea = warped_src_fea.view(batch, channels, num_depth, height,
width)

    return warped_src_fea
```

通过 $Differentiable\ Homography$ 后，生成了 $N$ 个代价体 $V_{i=1}^{N}$，其中，$N$ 为视角的个数（对于参考帧只是把特征图 repeat 到每个空间层数即可）。将特征体 $V_{i=1}^{N}$ 进行代价聚合：

$$\boldsymbol{C}=\frac{\sum_{i=1}^{N}(\boldsymbol{V}_i-\overline{\boldsymbol{V}_i})^2}{N} \tag{10}$$

将代价聚合之后的代价体 $\boldsymbol{C}$ 经过 3D CNN 回归后对每个像素做 $softmax$ 得到概率体 $\boldsymbol{P}$，其意义为每个像素每一层的概率。根据对空间划分的深度 $d$ 估计深度，得到 Initial Depth Map：

$$\boldsymbol{D}=\sum_{d=d_{min}}^{d_{max}} d\times\boldsymbol{P}(d) \tag{11}$$

最后再利用原图信息得到 Refined Depth Map。将 Initial Depth Map 与原图进行 concat 操作经过残差网络获得 Refined Depth Map。最终的 Loss 定义为：

$$Loss=\sum_{p}(|d(p)-\widehat{d}_i(p)|+\lambda|d(p)-\widehat{d}_r(p)|) \tag{12}$$

其中，$d(p)$ 表示真实深度，$\widehat{d}_i(p)$ 表示 Initial Depth Map，$\widehat{d}_r(p)$ 表示 Refined Depth Map，$\lambda$ 为两个损失的比例。MVSNet 虽然能够实现重建但是还存在着一些缺陷：（1）在进行代价体回归的时候用了很大的 3D CNN 网络，占用了很大的内存。（2）对每个视角的像素进行等价聚合，不同视角的像素应该赋予不同的权重从而进行加权聚合。（3）Loss 计算的时候只是用了真实深度与预测深度的差的绝对值，但是更希望在得到的概率体上使得越靠近真实深度的层数概率越大，远离真实深度的概率概率越小，会大大增加模型的准确性。

#### PatchmatchNet--金字塔结构与自适应邻域[3]

为了解决上述问题，2021年提出了 PatchmatchNet，仅仅用简单的网络结构完成重建。PatchmatchNet 在与其他重建网络对比如下图所示：

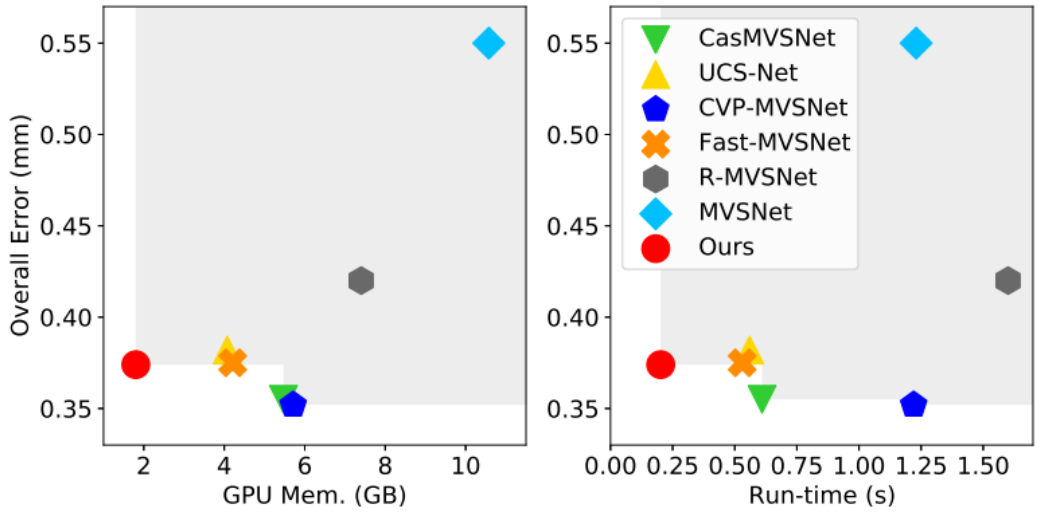


图2

可以看到，无论是内存的占用还是在运行时间上 PatchmatchNet 都远超其他的网络。而且其完整性超过目前现有的所有网络，综合指标在众多网络中名列前茅。Patchmatchnet 之所以使用少量网络结构就能够实现重建的原因是利用了金字塔结构和自适应邻域的结构，网络框架如下：

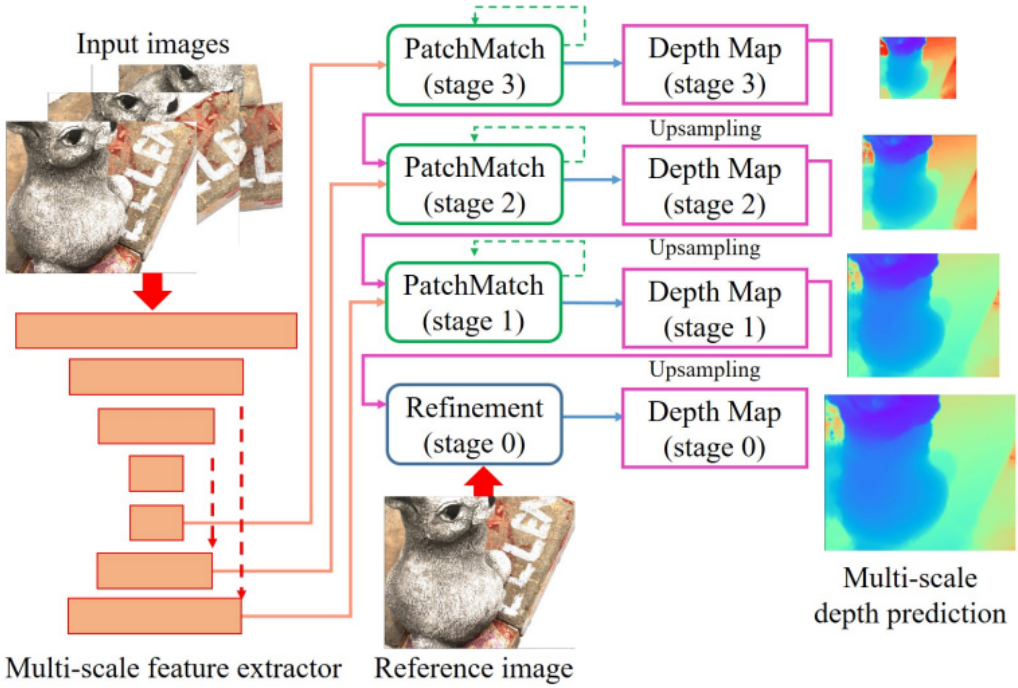


图3

图像经过提取特征的 CNN网络 后分为 4 个 stage。stage0 ，stage1，stage2，stage3 的图片尺寸分别为原图，原图的 $\frac{1}{2}$，$\frac{1}{4}$，$\frac{1}{8}$。该网络不同于 MVSNet 的特征提取网络，先将原图缩小再把特征图进行上采样并且和对应缩小时的特征图进行求和，其目的是用小感受野（缩小图像的部分）提取纹理丰富的区域，用大感受野（放大特征图的部分）提取弱纹理区域，能够提升重建的准确性。代码如下：

```
class FeatureNet(nn.Module):
    """Feature Extraction Network: to extract features of original images from each view"""

    def __init__(self):
        """Initialize different layers in the network"""

        super(FeatureNet, self).__init__()
```

```
        self.conv0 = ConvBnReLU(3, 8, 3, 1, 1)
        # [B,8,H,W]
        self.conv1 = ConvBnReLU(8, 8, 3, 1, 1)
        # [B,16,H/2,W/2]
        self.conv2 = ConvBnReLU(8, 16, 5, 2, 2)
        self.conv3 = ConvBnReLU(16, 16, 3, 1, 1)
        self.conv4 = ConvBnReLU(16, 16, 3, 1, 1)
        # [B,32,H/4,W/4]
        self.conv5 = ConvBnReLU(16, 32, 5, 2, 2)
        self.conv6 = ConvBnReLU(32, 32, 3, 1, 1)
        self.conv7 = ConvBnReLU(32, 32, 3, 1, 1)
        # [B,64,H/8,W/8]
        self.conv8 = ConvBnReLU(32, 64, 5, 2, 2)
        self.conv9 = ConvBnReLU(64, 64, 3, 1, 1)
        self.conv10 = ConvBnReLU(64, 64, 3, 1, 1)

        self.output1 = nn.Conv2d(64, 64, 1, bias=False)
        self.inner1 = nn.Conv2d(32, 64, 1, bias=True)
        self.inner2 = nn.Conv2d(16, 64, 1, bias=True)
        self.output2 = nn.Conv2d(64, 32, 1, bias=False)
        self.output3 = nn.Conv2d(64, 16, 1, bias=False)

    def forward(self, x: torch.Tensor) :
        """Forward method

        Args:
            x: images from a single view, in the shape of [B, C, H, W]. Generally, C=3

        Returns:
            output_feature: a python dictionary contains extracted features from stage_1 to stage_3
                keys are "stage_1", "stage_2", and "stage_3"
        """
        output_feature = {}

        conv1 = self.conv1(self.conv0(x))
        conv4 = self.conv4(self.conv3(self.conv2(conv1)))

        conv7 = self.conv7(self.conv6(self.conv5(conv4)))
        conv10 = self.conv10(self.conv9(self.conv8(conv7)))

        output_feature["stage_3"] = self.output1(conv10)

        intra_feat = F.interpolate(conv10, scale_factor=2, mode="bilinear") + self.inner1(conv7)
        del conv7, conv10
        output_feature["stage_2"] = self.output2(intra_feat)

        intra_feat = F.interpolate(intra_feat, scale_factor=2, mode="bilinear") + self.inner2(conv4)
        del conv4
        output_feature["stage_1"] = self.output3(intra_feat)

```

```
        del intra_feat
        return output_feature
```

在每一个 stage 都会产生深度图，把每一层深度图进行上采样后传入前一个 stage，前一个 stage 在基于这个深度图进行深度的采样。每一个 stage 的框架如下：

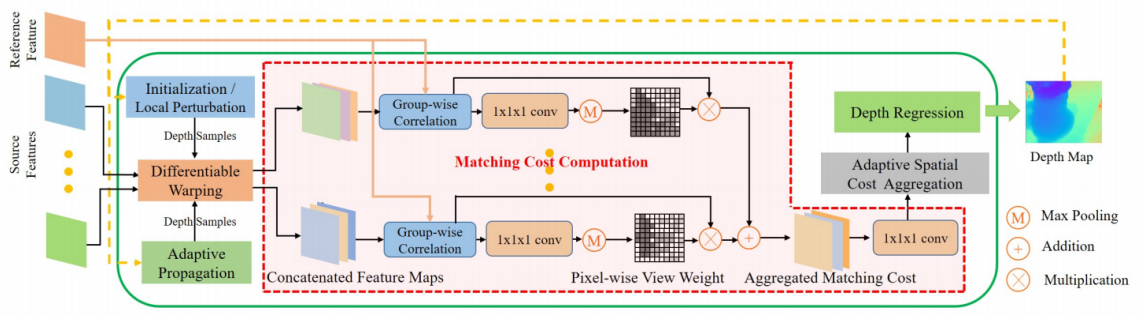


图4

网络的输入为参考帧的特征图和邻域帧的特征图。首先进行空间层的采样，分为局部扰动（$LocalPerturbation$）和自适应传播（$AdaptivePropagation$）。局部扰动指的是基于上一层的深度图对每个像素在当前深度下上下选取几层。自适应传播指的是在特征图上寻找自适应邻域，由于自适应邻域的深度一般差别很小，所以把自适应邻域的深度值当作当前像素深度值的初始化。

寻找自适应传播的方法为，在特征图当前点 $p$ 的固定的邻域 $\{o_i(p)\}_{i=1}^{K_p}$（$K_p$ 表示选取的邻域数）利用 2D CNN 寻找这些邻域的偏移量 $\{\Delta o_i(p)\}_{i=1}^{K_p}$，选取的深度层为：

$$D_p(p) = \{D(p + o_i + \Delta o_i(p))\}_{i=1}^{K_p} \tag{13}$$

其中，$D$ 为上一个 stage 经过上采样后的深度图。

选取自适应邻域的代码，输出的通道为两倍的邻域，指的是 x 和 y 方向上的偏移：

```
self.propa_conv = nn.Conv2d(
    in_channels=self.propa_num_feature,
    out_channels=2 * self.propagate_neighbors,
    kernel_size=3,
    stride=1,
    padding=self.dilation,
    dilation=self.dilation,
    bias=True,
)
```

计算邻域偏移量的代码：

```
 def get_propagation_grid(
        self, batch: int, height: int, width: int, offset: torch.Tensor,
device: torch.device, img: torch.Tensor = None
    ) :
        """Compute the offset for adaptive propagation
    Args:
        batch: batch size
        height: grid height
        width: grid width
        offset: grid offset #[B,2*N_neighbors,H*W]
        device: device on which to place tensor
        img: reference images, (B, C, image_H, image_W)

```

```
    Returns:
        generated grid: in the shape of [batch, propagate_neighbors*H, W, 2]
    """
    # 原始的规则的矩形mask相对中心点的偏移量比如四邻域：左[-1,0] 右[1,0] 上[0,-1] 下[0,1]
    # 仅支持4，8，16邻域 其它比如32可以自己实现
    if self.propagate_neighbors == 4:  # if 4 neighbors to be sampled in propagation
        original_offset = [[-self.dilation, 0], [0, -self.dilation], [0, self.dilation], [self.dilation, 0]]
    elif self.propagate_neighbors == 8:  # if 8 neighbors to be sampled in propagation
        original_offset = [
            [-self.dilation, -self.dilation],
            [-self.dilation, 0],
            [-self.dilation, self.dilation],
            [0, -self.dilation],
            [0, self.dilation],
            [self.dilation, -self.dilation],
            [self.dilation, 0],
            [self.dilation, self.dilation],
        ]
    elif self.propagate_neighbors == 16:  # if 16 neighbors to be sampled in propagation
        original_offset = [
            [-self.dilation, -self.dilation],
            [-self.dilation, 0],
            [-self.dilation, self.dilation],
            [0, -self.dilation],
            [0, self.dilation],
            [self.dilation, -self.dilation],
            [self.dilation, 0],
            [self.dilation, self.dilation],
        ]
        for i in range(len(original_offset)):
            offset_x, offset_y = original_offset[i]
            original_offset.append([2 * offset_x, 2 * offset_y])
    else:
        raise NotImplementedError

    with torch.no_grad():
        y_grid, x_grid = torch.meshgrid(
            [
                torch.arange(0, height, dtype=torch.float32, device=device),
                torch.arange(0, width, dtype=torch.float32, device=device),
            ]
        )
        y_grid, x_grid = y_grid.contiguous(), x_grid.contiguous()
        y_grid, x_grid = y_grid.view(height * width), x_grid.view(height * width)
        xy = torch.stack((x_grid, y_grid))  # [2, H*W]
        xy = torch.unsqueeze(xy, 0).repeat(batch, 1, 1)  # [B, 2, H*W] 存放的是xy的像素坐标

    xy_list = []
```

```
    # 将前面学习得到的自适应偏移量加到原始mask的偏移量上并加上中心点坐标得到用来传播的邻域像素的像素坐标
    for i in range(len(original_offset)):
        original_offset_y, original_offset_x = original_offset[i]

        offset_x_tensor = original_offset_x + offset[:, 2 * i, :].unsqueeze(1)
        offset_y_tensor = original_offset_y + offset[:, 2 * i + 1, :].unsqueeze(1)

        xy_list.append((xy + torch.cat((offset_x_tensor, offset_y_tensor), dim=1)).unsqueeze(2))

    xy = torch.cat(xy_list, dim=2)  # [B, 2, 9, H*W]

    del xy_list, x_grid, y_grid
    # 归一化到[-1,1]
    x_normalized = xy[:, 0, :, :] / ((width - 1) / 2) - 1
    y_normalized = xy[:, 1, :, :] / ((height - 1) / 2) - 1
    del xy
    grid = torch.stack((x_normalized, y_normalized), dim=3)  # [B, 9, H*W, 2]
    del x_normalized, y_normalized
    grid = grid.view(batch, self.propagate_neighbors * height, width, 2)
    return grid
```

选取层数之后把不同视角利用上述 $Differentiable\ Homography$ 方法转换到参考帧上，构造代价体。代价聚合利用了点积来表示归一化向量的相似性，即点积越接近 1 表示特征的相似性越强。设$i$表示第 $i$ 个参考帧，$j$ 表示第 $j$ 个视角，$F_0(p)$ 表示 $p$ 像素的特征向量，$F_i(P_{i,j})$ 表示第 $i$ 个视角第 $j$ 层参考帧像素 $P$ 对应邻域帧的像素位置的特征向量，将将这些特征向量的通道数分成 $G$ 组得到 $F_0(p)^g$，$F_i(P_{i,j})^g$。对每个视角的聚合代价为：

$$S_i(p,j)^g = \frac{G}{C} < F_0(p)^g, F_i(p_{i,j})^g > \tag{14}$$

把每个视角的代价进行聚合需要计算每个视角的权重。类似于 MVSNet 这里使用了 3D CNN 将 $S_i(p,j)$ 中的 $G$ 回归掉再经过 softmax 后得到了 $D$ 层深度层的概率体 $P_i$。计算第 $i$ 个视角每个像素的权重：

$$w_i(p) = max\{Pi(p,j)|j = 0,1,\ldots,D-1\} \tag{15}$$

有了视角权重之后将不同的 $N$ 个视角进行带权聚合：

$$\overline{S}(p,j) = \frac{\sum_{i=1}^{N-1} w_i(p) \cdot S_i(p,j)}{\sum_{i=1}^{N-1} w_i(p)} \tag{16}$$

类似于自适应传播，在聚合的时候也使用了自适应邻域，不同的是自适应传播的邻域只是进行深度层的选取不需要计算权重，而这里的邻域要进行对当前像素的修正，故要计算权重。设 $w_k$ 表示特征的相似性，$d_k$ 表示深度的相似性，$p$ 表示当前点，$p_k$ 表示固定的邻域，$\Delta p_k$ 表示固定邻域的偏移量，$C$ 表示 $\overline{S}$ 经过 3D CNN 将 $G$ 回归掉的特征体，$K_e$ 表示选取的邻域数量，邻域 refine 的公式为：

$$\widetilde{C}=\frac{1}{\sum_{k=1}^{K_e} w_k d_k} C(p+p_k+\Delta p_k, j) \tag{17}$$

和 MVSNet 一样得到每个像素深度，Loss 计算：

$$L_{total}=\sum_{k=1}^{3}\sum_{i=1}^{n_k} L_i^k + L_{ref}^0 \tag{18}$$

其中，$n_k$ 表示每个 stage 迭代的次数，$L$ 表示 L1 损失，计算方法和 MVSNet 一样。

**MaGNet--依分布采样和层级损失[4]**

patchmatchnet 改进了 MVSNet 中（1）和（2）的不足，MaGNet改进了其（3）的不足。MaGNet 主要做了以下贡献：（1）层数采样采用了依分布采样，靠近估计的准确准确层采样越密集，否则越稀疏。（2）对概率体做 Loss。使得靠近估计深度的层数概率大远离该深度的层数概率小。代码框架如下：

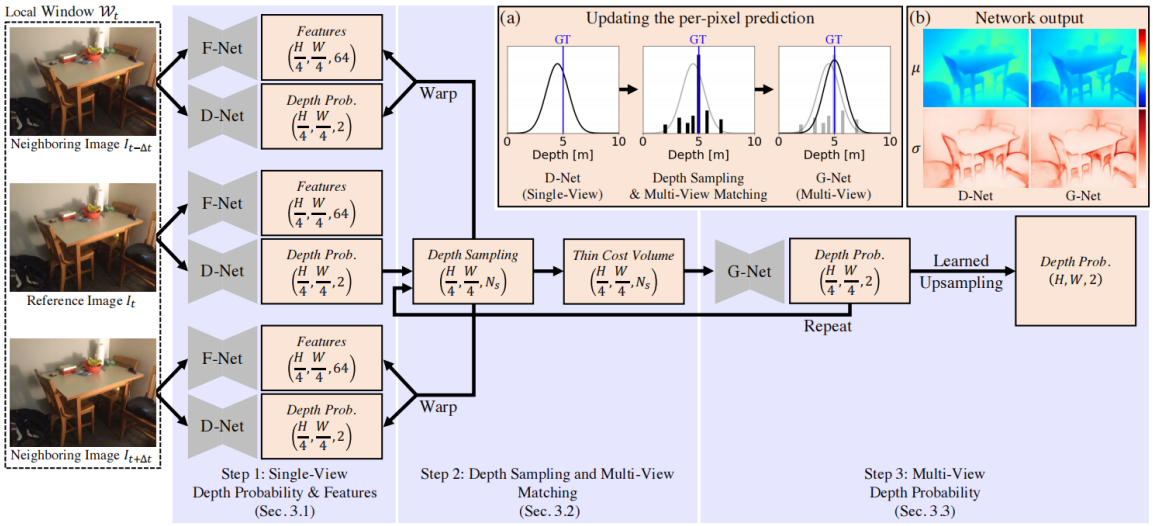


图5

输入一组多视角图片，经过 F-Net 提取图像特征，经过 D-Net 学习每个像素深度的高斯分布参数 $\mu$ 和 $\sigma$。利用高斯分布进行依分布采样，推导如下：

设 $(u,v)$ 为参考帧上像素的位置，D-Net 估计出出每个像素的均值和方差记为 $\mu_{u,v}$，$\sigma_{u,v}$。利用高斯分布采样原理确定采样范围 $[\mu_{u,v}-\beta\sigma_{u,v}, \mu_{u,v}+\beta\sigma_{u,v}]$，$\beta$ 为超参数。给定概率分布函数 $p_{u,v}(x)$，在这个区间上的概率为：

$$\begin{aligned} P_{u,v}^* &= \int_{\mu_{u,v}-\beta\sigma_{u,v}}^{\mu_{u,v}+\beta\sigma_{u,v}} p_{u,v}(x)dx \\ &= Fu,v(\mu_{u,v}+\beta\sigma_{u,v}) - Fu,v(\mu_{u,v}-\beta\sigma_{u,v}) \\ &= erf(\frac{\beta}{\sqrt{2}}) \end{aligned} \tag{19}$$

其中，$F_{u,v}(\cdot)$ 为累积概率函数，$erf(\cdot)$ 为误差函数。设取样的层数为 $N_s$，将区间 $[\mu_{u,v}-\beta\sigma_{u,v}, \mu_{u,v}+\beta\sigma_{u,v}]$ 依等概率划分为 $N_s$ 份，取每一份的中间值作为采样的层数，第 $k$ 层采样的深度 d_{u,v,k}为：

$$d_{u,v,k}=\frac{1}{2}[F_{u,v}^{-1}(\frac{k-1}{N_s}P^*+\frac{1-P^*}{2})+F_{u,v}^{-1}(\frac{k}{N_s}P^*+\frac{1-P^*}{2})] \tag{20}$$

又由于

$$F_{u,v}^{-1} = \mu_{u,v} + \sigma_{u,v}\Phi^{-1}(p) \tag{21}$$

其中，$\Phi^{-1}(p)$ 为标准正态分布的密度函数，有：

$$\begin{aligned} du, v, k &= \mu_{u,v} + b_k \sigma u, v \\ b_k &= \frac{1}{2}[\Phi^{-1}(\frac{k-1}{N_s}P^* + \frac{1-P^*}{2}) + \Phi^{-1}(\frac{k}{N_s}P^* + \frac{1-P^*}{2})] \end{aligned} \tag{22}$$

$b_k$ 计算代码实现如下：

```
def depth_sampling(self):
    from scipy.special import erf
    from scipy.stats import norm
    P_total = erf(self.sampling_range / np.sqrt(2))
    idx_list = np.arange(0, self.n_samples + 1)
    p_list = (1 - P_total)/2 + ((idx_list/self.n_samples) * P_total)
    k_list = norm.ppf(p_list)
    k_list = (k_list[1:] + k_list[:-1])/2
    return list(k_list)
```

选择层数后进行代价聚合，对每个视角进行权重计算，这里采用二值权重，即能看到参考帧像素的视角权重为 1，否则为 0。设 $d_{ik}$ 为第 $i$ 个视角下参考帧 $k$ 深度对应的深度，$I_i$ 为第 $i$ 个视角，$p_{u_{ik},v_{ik}}(d_{ik}|I_i)$ 表示第 $i$ 个视角的高斯分布的深度值为 $d_{ik}$ 的概率密度，如果 $p_{u_{ik},v_{ik}}(d_{ik}|I_i)$ 小，可能该视角看不到对应的参考帧像素，例如下图所示（视角被阻塞的情况）：

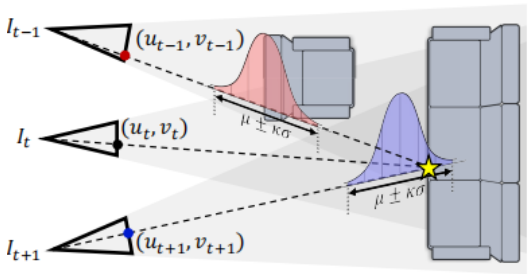


图6

权重定义如下：

$$\begin{aligned} w_{u_{ik},v_{ik},dik} &= [p_{u_{ik},v_{ik}}(d_{ik}|I_i) > p_{thres}] \\ p_{thres} &= \frac{e^{(\frac{-\kappa}{2})}}{\sigma_{u_{ik},v_{ik}}\sqrt{2\pi}} \end{aligned} \tag{23}$$

其中，$\kappa$ 为超参数。当大于阈值 $p_{thres}$ 时权重为 1，否则为 0。$\sigma_{u_{ik},v_{ik}}$ 使得阈值具有自适应的性质，当重建一些弱纹理区域时，由于重建的深度模糊性大，$\sigma_{u_{ik},v_{ik}}$ 的值比较大，使得阈值会减小，使得更多的视角参与重建。

代价聚合采用点积的方式，类似 PatchmatchNet，聚合公式为：

$$s_{u,v,k}(I_t) = \sum_{i \neq t} w_{u_{ik},v_{ik},dik} < f_{u,v}(I_t), f_{u_{ik},v_{ik}}(I_i) > \tag{24}$$

通过 G-Net 计算高斯分布 $\mu$ 和 $\sigma$ 的增量，最后再通过可学习的上采样恢复到原图，得到原图的深度图。

Loss 定义为：

$$Loss = \frac{1}{2} log\sigma_{u,v}^2(I_t) + \frac{(d_{u,v}^{gt} - \mu_{u,v}(I_t))^2}{2\sigma_{u,v}^2(I(t))} \tag{25}$$

该 Loss 没有直接用估计得到的深度层和真值做 Loss 计算，而是用均值层与真值计算损失，其好处为可以使均值层（最可能的深度层）接近于真值并且使得 $\sigma_{u,v}$ 变小，保证在均值附近的层概率比较大而远离均值的层概率小，增加了结果的准确性。并且在重建纹理丰富的区域时 $\sigma_{u,v}$ 偏小，在重建弱纹理区域时由于真值和均值差别很大，所以减小 Loss 的途径是增大 $\sigma_{u,v}$，这可以使更多的深度层参与计算，提高结果的准确性。

**NeRF[5] 与NerfingMVS[6]--利用空间信息优化重建结果**

NeRF是近年来大火的视角合成技术。它可以学习到空间点的密度，可以用来对重建好的点云进行优化。其框架如下：

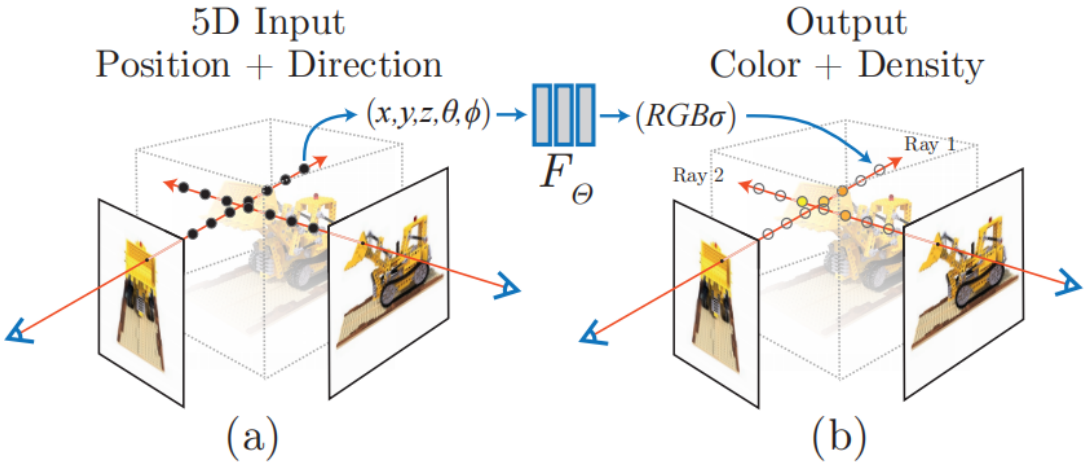


图7

NeRF 函数是将一个连续的场景表示为一个输入为 5D 向量的函数，包括一个空间点的 3D 坐标位置 $\boldsymbol{x} = (x, y, z)$，以及视角方向 $\boldsymbol{d} = (\theta, \phi)$。这个神经网络（是一个MLP）可以写成：

$$F : (\boldsymbol{x}, \boldsymbol{d}) \rightarrow (\boldsymbol{c}, \sigma) \tag{26}$$

输出结果中，$\sigma$ 是对应 3D 位置的密度，而 $\boldsymbol{c} = (r, g, b)$ 是视角相关的该 3D 点颜色。通过视线上的积分便能够获得任意视角的图像，即为视角合成。将一个相机射线标记为 $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$，这里 $\boldsymbol{o}$ 是射线原点，$\boldsymbol{d}$ 是前述的相机射线角度，$t$ 近端和远端边界为 $t_n$ 和 $t_f$。那么这条射线的颜色，则可以用积分的方式表示为：

$$C(r) = \int_{t_n}^{t_f} T(t) \cdot \sigma(\boldsymbol{r}(t)) \cdot \boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d}) dt \tag{27}$$

$T(t)$ 是射线从 $t_n$ 到 $t$ 这一段路径上的累计透明度，即从这条射线从 $t_n$ 到 $t$ 一路上没有击中任何粒子的概率。具体形式为：

$$T(t) = e^{-\int_{tn}^{tf} \sigma(\boldsymbol{r}(s) ds)} \tag{28}$$

对积分进行离散化进行模拟，将射线需要积分的区域分成 $N$ 份，然后在每个小区域上进行均匀随机采样 $U$。第 $i$ 个采样点为：

$$t_i = U[t_n + \frac{i-1}{N}, t_n + \frac{i}{N}(t_f - t_n)] \tag{29}$$

基于这些采样点，将积分化简成求和的形式：

$$\begin{aligned}\hat{C} &= \sum_{i=1}^{N} T_i(1-e^{-\sigma_i\delta_i})\boldsymbol{c}_i \\ \delta_i &= t_{i+1}-t_i \\ T_i &= e^{-\sum_{j=1}^{i-1}\sigma_j\delta_j}\end{aligned} \tag{30}$$

NeRF的渲染过程计算量很大，每条射线都要采样很多个点，但是一条射线的大部分区域是空区域，对最终的颜色贡献小。为了解决这个问题，首先进行 coarse 采样，采样较为稀疏的 $N_c$ 个点，按照上述方式进行采样，将前述离散形式重新表示为：

$$\begin{aligned}\hat{C}_c(\boldsymbol{r}) &= \sum_{i=1}^{N_c} w_i c_i \\ w_i &= T_i(1-e^{-\sigma_i\delta_i})\end{aligned} \tag{31}$$

对 $w_i$ 做归一化

$$\hat{w}_i = \frac{w_i}{\sum_{j=1}^{N_c} w_j} \tag{32}$$

渲染部分的代码实现：

```
def render_rays(ray_batch,
                network_fn,
                network_query_fn,
                N_samples,
                retraw=False,
                lindisp=False,
                perturb=0.,
                N_importance=0,
                network_fine=None,
                white_bkgd=False,
                raw_noise_std=0.,
                verbose=False,
                pytest=False):
    """Volumetric rendering.
    Args:
      ray_batch: array of shape [batch_size, ...]. All information
necessary
        for sampling along a ray, including: ray origin, ray direction, min
        dist, max dist, and unit-magnitude viewing direction.
      network_fn: function. Model for predicting RGB and density at each
point
        in space.
      network_query_fn: function used for passing queries to network_fn.
      N_samples: int. Number of different times to sample along each ray.
      retraw: bool. If True, include model's raw, unprocessed predictions.
      lindisp: bool. If True, sample linearly in inverse depth rather than
in depth.
      perturb: float, 0 or 1. If non-zero, each ray is sampled at
stratified
        random points in time.
      N_importance: int. Number of additional times to sample along each
ray.
```

```
        These samples are only passed to network_fine.
      network_fine: "fine" network with same spec as network_fn.
      white_bkgd: bool. If True, assume a white background.
      raw_noise_std: ...
      verbose: bool. If True, print more debugging info.
    Returns:
      rgb_map: [num_rays, 3]. Estimated RGB color of a ray. Comes from fine model.
      disp_map: [num_rays]. Disparity map. 1 / depth.
      acc_map: [num_rays]. Accumulated opacity along each ray. Comes from fine model.
      raw: [num_rays, num_samples, 4]. Raw predictions from model.
      rgb0: See rgb_map. Output for coarse model.
      disp0: See disp_map. Output for coarse model.
      acc0: See acc_map. Output for coarse model.
      z_std: [num_rays]. Standard deviation of distances along ray for each
        sample.
    """
    N_rays = ray_batch.shape[0]
    rays_o, rays_d = ray_batch[:,0:3], ray_batch[:,3:6] # [N_rays, 3] each
    viewdirs = ray_batch[:,-3:] if ray_batch.shape[-1] > 8 else None
    bounds = torch.reshape(ray_batch[...,6:8], [-1,1,2])
    near, far = bounds[...,0], bounds[...,1] # [-1,1]

    t_vals = torch.linspace(0., 1., steps=N_samples)
    if not lindisp:
        z_vals = near * (1.-t_vals) + far * (t_vals)
    else:
        z_vals = 1./(1./near * (1.-t_vals) + 1./far * (t_vals))

    z_vals = z_vals.expand([N_rays, N_samples])

    if perturb > 0.:
        # get intervals between samples
        mids = .5 * (z_vals[...,1:] + z_vals[...,:-1])
        upper = torch.cat([mids, z_vals[...,-1:]], -1)
        lower = torch.cat([z_vals[...,:1], mids], -1)
        # stratified samples in those intervals
        t_rand = torch.rand(z_vals.shape)

        # Pytest, overwrite u with numpy's fixed random numbers
        if pytest:
            np.random.seed(0)
            t_rand = np.random.rand(*list(z_vals.shape))
            t_rand = torch.Tensor(t_rand)

        z_vals = lower + (upper - lower) * t_rand

    pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None] # [N_rays, N_samples, 3]


#     raw = run_network(pts)
    raw = network_query_fn(pts, viewdirs, network_fn)
```

```
    rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd, pytest=pytest)

    if N_importance > 0:

        rgb_map_0, disp_map_0, acc_map_0 = rgb_map, disp_map, acc_map

        z_vals_mid = .5 * (z_vals[...,1:] + z_vals[...,:-1])
        z_samples = sample_pdf(z_vals_mid, weights[...,1:-1], N_importance, det=(perturb==0.), pytest=pytest)
        z_samples = z_samples.detach()

        z_vals, _ = torch.sort(torch.cat([z_vals, z_samples], -1), -1)
        pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None] # [N_rays, N_samples + N_importance, 3]

        run_fn = network_fn if network_fine is None else network_fine
#         raw = run_network(pts, fn=run_fn)
        raw = network_query_fn(pts, viewdirs, run_fn)

        rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd, pytest=pytest)

    ret = {'rgb_map' : rgb_map, 'disp_map' : disp_map, 'acc_map' : acc_map}
    if retraw:
        ret['raw'] = raw
    if N_importance > 0:
        ret['rgb0'] = rgb_map_0
        ret['disp0'] = disp_map_0
        ret['acc0'] = acc_map_0
        ret['z_std'] = torch.std(z_samples, dim=-1, unbiased=False)  # [N_rays]

    for k in ret:
        if (torch.isnan(ret[k]).any() or torch.isinf(ret[k]).any()) and DEBUG:
            print(f"! [Numerical Error] {k} contains nan or inf.")

    return ret
```

此处的 $\hat{w}_i$ 可以看作沿着射线的概率函数，通过这个概率函数，可以粗略得到射线上物体的分布情况。类似于 MaGNet 基于得到的概率函数采样 $N_f$ 个点（即在概率大的地方采样密集）。利用这 $N_f$ 个点和前面的 $N_c$ 个点按照上述公式一同计算 fine 渲染结果 $\hat{C}_f(\boldsymbol{r})$，用其与照片颜色 $C(\boldsymbol{r})$ 作 Loss：

$$Loss = \sum_{\boldsymbol{r}\in\mathcal{R}}[\| \hat{C}_c(\boldsymbol{r}) - C(\boldsymbol{r}) \|_2^2 + \| \hat{C}_f(\boldsymbol{r}) - C(\boldsymbol{r}) \|_2^2] \tag{33}$$

NeRF 虽然完成的是视角合成，但是其蕴含了整个场景的信息，更有潜力重建出低纹理区域，于是出现了 NerfingMVS 的方法，主要用来对重建的结果进行 refine，增强重建的效果。类似于公式（30），计算视角深度时只需要将 $\boldsymbol{c}_i$ 改为 $t_i$ 即可：

$$D(\boldsymbol{r}) = \sum_{i=1}^{N} T_i(1 - e^{-\sigma_i\delta_i})t_i \tag{34}$$

与 NeRF 另一点不同的是 NerfingMVS 在视线上进行采样的时候是按照多视角一致性检测得到的误差图进行采样的，在误差小的地方进行密集采样，在误差大的地方采样稀疏些，让更多的深度参与计算，其思想类似于 MaGNet 的 Loss 的意义。NerfingMVS 又提出了一个滤波器，即如果渲染的 RGB 图像置信度低，那么计算出的深度应该也不准确，这些像素的深度应该被过滤掉，定义为：

$$S_j^i = 1 - \frac{1}{3} \parallel C_{gt}^i(j) - C_{render}^i(j) \parallel \tag{35}$$

其中 $C_{gt}^i$ 为第 $i$ 个视角的真实图像，$C_{render}^i$ 为第 $i$ 个图像的渲染图像，其值都要先除 255 进行归一化。$s_j^i$ 越高表示重建效果越好。

**四、深度学习仍旧面临的挑战**

虽然很多网络都一直在增强三维重建的效果，但是对于弱纹理区域重建效果还是不够理想，其根本原因在于空间中进行层数采样后进行代价聚合网络搜索不到正确的层数，比如说一面白墙，在很多层数代价聚合的值都几乎一样，不能判断像素所在的深度。现有的算法是通过邻域的深度具有相似性，在重建不好的区域扩大采样层数等方法来尽可能增强弱纹理区域的重建。不仅仅是深度学习方法，传统方法对弱纹理区域的重建效果也不理想，在这方面仍然有大量工作去做。

**参考文献**

[1] Knapitsch A，Park J，Zhou Q Y，et al. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction[J]. ACM Transactions on Graphics, 2017, 36(4CD):78.1-78.13.

[2] Yao Y，Luo Z，Li S，et al. MVSNet: Depth Inference for Unstructured Multi-view Stereo[J]. 2018.

[3] Wang F，Galliani S，Vogel C，et al. PatchmatchNet: Learned Multi-View Patchmatch Stereo[J]. 2020.

[4] Bae G，Budvytis I，Cipolla R . Multi-View Depth Estimation by Fusing Single-View Depth Probability with Multi-View Geometry[J]. 2021.

[5] Mildenhall B，Srinivasan P P，Tancik M，et al. NeRF: representing scenes as neural radiance fields for view synthesis[J]. Communications of the ACM, 2022, 65(1):99-106.

[6] Wei Y，Liu S，Rao Y，et al. NerfingMVS: Guided Optimization of Neural Radiance Fields for Indoor Multi-view Stereo[J]. 2021.

文章中对应的代码：https://github.com/Xiaohan-Z/-.git