LIIF 复现报告

目录

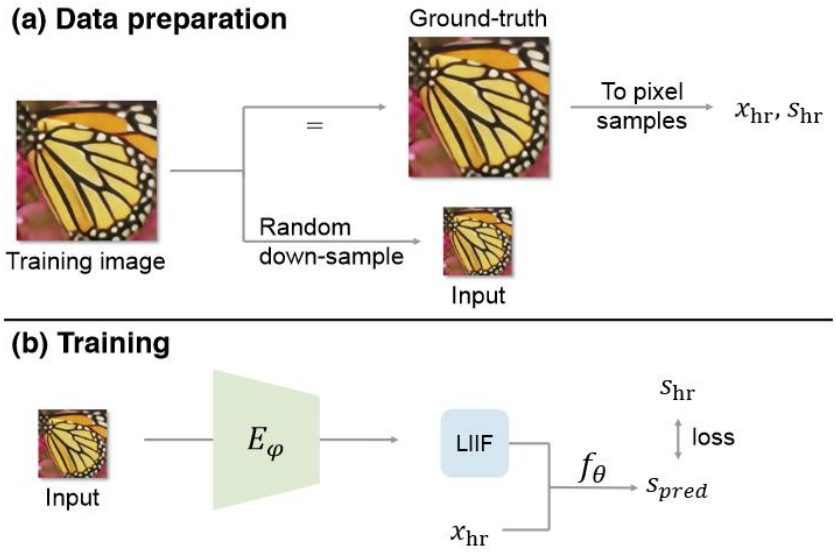
一、 动机

虽然基于像素的表示已成功应用于各种计算机视觉任务，但它们也受到分辨率的限制。例如，数据集通常由具有不同分辨率的图像呈现。目前通用的方法是将不同分辨率的图像先统一 resize 到同一大小再进行训练，这种方法方式无疑会损失一部分监督信息。作者建议研究图像的连续表示，而不是用固定分辨率表示图像。通过将图像建模为在连续域中定义的函数，我们可以根据需要以任意分辨率恢复和生成图像。

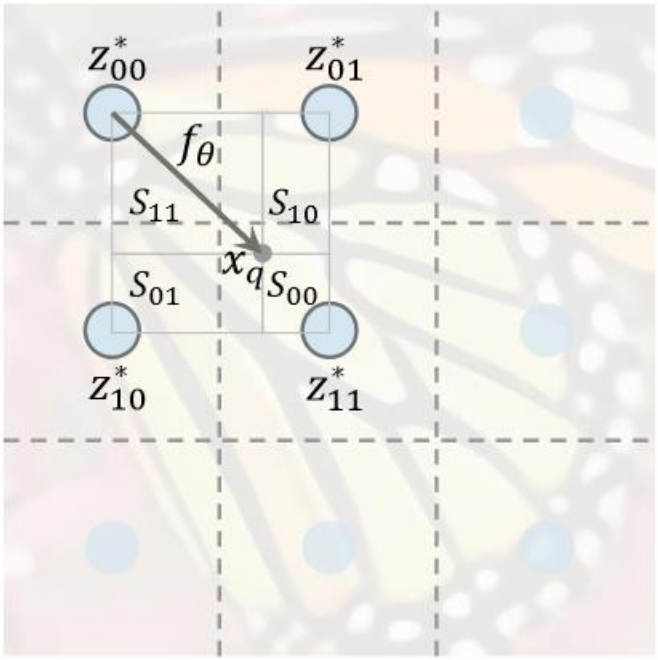
二、 模型框架。

模型的整体框架如下所示，在数据准备阶段，对于一张给定的高清图像，我们将其分解为坐标位置（二维）和对应位置的 RGB 值。训练流程上，输入的低分辨率图像首先经过一个特征提取模块（比如 EDSR）得到特征图，然后依照高清图像的坐标系采样出特征向量，送入 LIIF 中查询出对应位置的 RGB 值。这里的 LIIF 是一个由多层 MLP 拟合的函数，其表达形式如下所示。

$$s = f_\theta(z, x),$$



为了进一步更好预测 RGB 值，作者在文中对最初的函数做了进一步的改进。



首先，作者对函数做了进一步细化，将坐标表示为当前查询坐标与最近坐标点的相对位置，特征向量也设置为处于最近坐标位置的坐标向量。具体如下所示

$$I^{(i)}(x_q) = f_\theta(z^*, x_q - v^*),$$

之后，为了丰富特征向量的表示内容，我们使用 3*3 邻域的特征向量重新表示当前的特征向量。如下所示：

$$\hat{M}_{jk}^{(i)} = \text{Concat}(\{M_{j+l,k+m}^{(i)}\}_{l,m\in\{-1,0,1\}}),$$

然后，为了使整个函数的变化连续，作者使用左上，右上，左下和右下四个位置加权计算出最后查询的的 RGB 值。如下所示：

$$I^{(i)}(x_q) = \sum_{t\in\{00,01,10,11\}} \frac{S_t}{S} \cdot f_\theta(z_t^*, x_q - v_t^*),$$

最后，为了达到任意倍率放大的需要，作者将网格大小作为输入加入到函数中，最终的函数表示如下：

$$s = f_{cell}(z, [x, c]),$$

三、 程序代码

模型的特征提取部分是一个去除了上采样模块的 EDSR，在此就不赘述了，下面将主要讲述下函数构建部分。

首先是如何构造坐标系，使用的 make_coord 这个函数，对于输入的每一个维度 d，作者将 x 的范围设置为[0,2d]，然后使用 torch.meshgrid 将每个维度的坐标整合成最终的坐标系。具体实现如下。

```
def make_coord(shape: list, ranges: list = None, flatten: bool = True)
-> Tensor:
    coord_seqs = []
    for i, n in enumerate(shape):
        if ranges is None:
            v0, v1 = -1, 1
        else:
            v0, v1 = ranges[i]
        r = (v1 - v0) / (2 * n)
        seq = v0 + r + (2 * r) * torch.arange(n).float()
        coord_seqs.append(seq)
    coord = torch.stack(torch.meshgrid(*coord_seqs, indexing="ij"), dim
=-1)

    if flatten:
        coord = coord.view(-1, coord.shape[-1])
```

```
    return coord
```

模型的 forward 的函数如下所示，输入的 x 是低分辨率图像，x_coord 是 GT 的坐标系，x_cell 是 GT 的网格大小。模型首先使用一个去掉上采样模块的 EDSR 模型提取特征图，然后使用 unfold 函数对特征图进行扩充，之后根据面积大小加权计算结果，得到最后预测的 RGB 值。

```
def _forward_impl(self, x: Tensor, x_coord: Tensor, x_cell: Tensor) ->
Tensor:
    vx_lst = [-1, 1]
    vy_lst = [-1, 1]
    eps_shift = 1e-6
    # print("x_coord",x_coord.shape)
    # print("x",x.shape)
    features = self.encoder(x)
    # print("features",features.shape)
    features = F_torch.unfold(features, (3, 3), padding=(1, 1)).view(fe
atures.shape[0],
                                                                     fe
atures.shape[1] * 9,
                                                                     fe
atures.shape[2],
                                                                     fe
atures.shape[3])
    # print("after feature unfold",features.shape)
    # Field radius (global: [-1, 1])
    rx = 2 / features.shape[-2] / 2
    ry = 2 / features.shape[-1] / 2

    features_coord = make_coord(features.shape[-2:], flatten=False).to(
x.device)
    # print("features_coord",features_coord.shape)
    features_coord = features_coord.permute(2, 0, 1).unsqueeze(0).expan
d(features.shape[0], 2, *features.shape[-2:])
    # print("features_coord",features_coord.shape)
    preds = []
    areas = []
    for vx in vx_lst:
        for vy in vy_lst:
            # prepare coefficient & frequency
            coord_ = x_coord.clone()
            # print(coord_[0,0])
            coord_[:, :, 0] += vx * rx + eps_shift
            coord_[:, :, 1] += vy * ry + eps_shift
```

```
            # print(coord_[0,0])
            coord_.clamp_(-1 + 1e-6, 1 - 1e-6)
            # print(coord_.flip(-1).unsqueeze(1).shape)
            q_features = F_torch.grid_sample(
                input=features,
                grid=coord_.flip(-1).unsqueeze(1),
                mode="nearest",
                align_corners=False)
            # print(q_features.shape)
            q_features = q_features[:, :, 0, :].permute(0, 2, 1)
            q_coord = F_torch.grid_sample(
                input=features_coord,
                grid=coord_.flip(-1).unsqueeze(1),
                mode="nearest",
                align_corners=False)[:, :, 0, :].permute(0, 2, 1)
            # print("q_features",q_features.shape)
            # print("q_coord",q_coord.shape)
            rel_coord = x_coord - q_coord
            rel_coord[:, :, 0] *= features.shape[-2]
            rel_coord[:, :, 1] *= features.shape[-1]
            inputs = torch.cat([q_features, rel_coord], -1)

            # prepare cell
            rel_cell = x_cell.clone()
            rel_cell[:, :, 0] *= features.shape[-2]
            rel_cell[:, :, 1] *= features.shape[-1]
            inputs = torch.cat([inputs, rel_cell], -1)
            # print("input",inputs.shape)
            # basis generation
            batch_size, q = x_coord.shape[:2]
            pred = self.mlp(inputs.view(batch_size * q, -1)).view(batch
_size, q, -1)
            # print("pred",pred.shape)
            preds.append(pred)

            area = torch.abs(rel_coord[:, :, 0] * rel_coord[:, :, 1])
            # print("area",area.shape)
            areas.append(area + 1e-9)

    tot_area = torch.stack(areas).sum(dim=0)
    # print("tot_area",tot_area.shape)
    t = areas[0]
    areas[0] = areas[3]
    areas[3] = t
```

```
    t = areas[1]
    areas[1] = areas[2]
    areas[2] = t

    out = 0
    for pred, area in zip(preds, areas):
        # print(pred.shape,area.shape)
        out = out + pred * (area / tot_area).unsqueeze(-1)
    # print("out",out.shape)
    return out
```

四、 实验结果

本次实验的训练集为 DIV2K，测试集为 Set5 对比方法为 SRCNN 和 SRGAN。

定量结果如下所示：

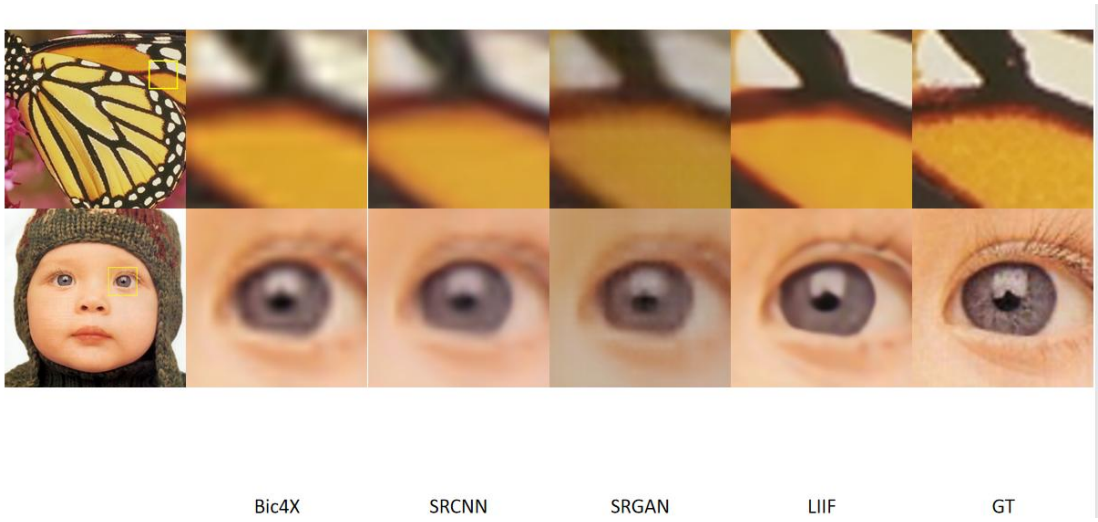
在本次实验中，我们采用的评价指标是 PSNR 和 SSIM，其中 SSIM 指标是一种结构相似度测量方法，它基于人眼的视觉特性，考虑了图像的亮度、对比度和结构等方面的特征。SSIM 的取值范围为[-1,1]，越接近 1 表示处理后的图像与原始图像越相似。而 PSNR 指标是一种峰值信噪比测量方法，它通过计算原始图像和处理后的图像之间的均方误差（MSE）来衡量图像的相似度，PSNR 的单位是分贝（dB）。PSNR 的取值范围为$[0, +\infty)$，越接近无穷大表示处理后的图像与原始图像越相似。

实验结果如下表所示，可以看到 LIIF 在两个指标上都显著优于 SRCNN 和 SRGAN。

方法	PSNR	SSIM
SRCNN	27.7541	0.7979
SRGAN	26.5206	0.7982
LIIF	**32.1538**	**0.8955**

定性分析结果如下所示，可以明显的看出，LIIF 的效果在局部细节上要远远好于 SRCNN 和 SRGAN，这是因为 LIIF 的模型更加复杂且机制更合理，能够更好的去捕获低分辨率图像中的局部细节并恢复到超分图像中。除此之外，LIIF

还可以实现任意分辨率的方法，而不需要重新训练新的模型，这也是模型的一大亮点。



五、 实验总结

总的来说，LIIF 提出了一种新的图像超分模型，在二维离散和连续表示之间搭建了一座桥梁，通过 LIIF，我们能够在保持高保真度的前提下，实现任意倍率的图像方法且不需要重新训练模型，为超分领域提供了一条新的解决思路。