计算机视觉实践——练习 3 改进_LIIF

目录

1 实验目的

- 复现论文 LIIF（局部隐式图像函数），以实现 SRCNN 与 SRGAN 未能完成的较高超分辨率效果。

2 介绍

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缩放（resize）到同一大小再进行训练（如练习 2 中的 SRCNN 与 SRGAN），这种方法方式无疑会损失一部分监督信息。LIIF 论文的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。

2 实验原理

模型的训练框架如下所示，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。

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

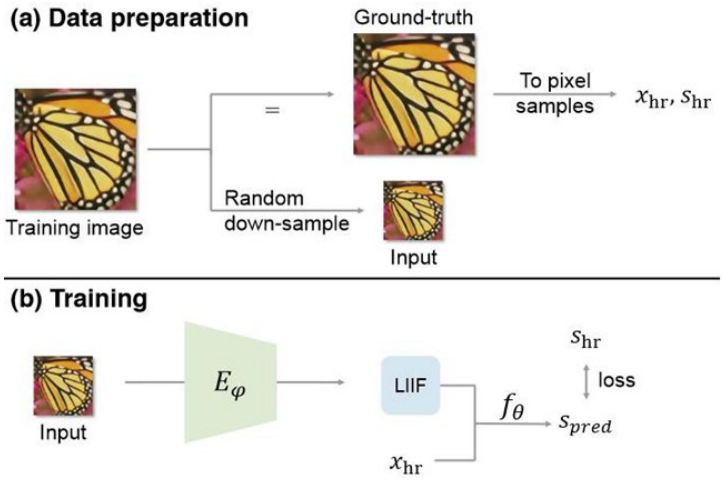

图 1 LIIF 训练流程

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。

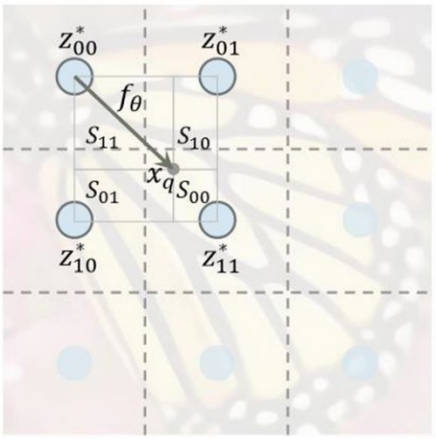

图 2 对角线区域计算像素值

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，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：

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

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，最终的函数为：

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

3 实验

3.1 数据集

本次实验的训练集为 DIV2K，测试集为 Set5。

3.2 实验细节

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。

表 1 指标得分对比

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

3.3 实验结果

实验结果如下所示，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。

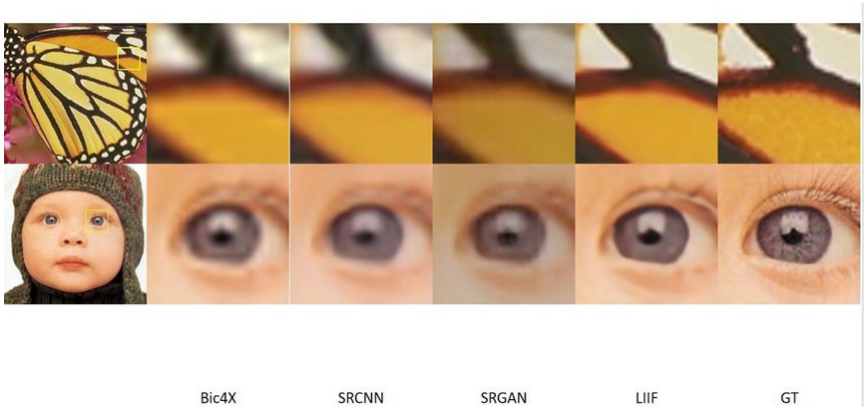

图 3 局部细节对比

3.4 实验分析

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。

附录——代码展示

仅展示主要代码。

```python
class LIIF(nn.Module):

    def __init__(
            self,
            in_channels: int = 3,
            encoder_channels: int = 64,
            out_channels: int = 3,
            encoder_arch: str = "edsr",
    ) -> None:
        super(LIIF, self).__init__()
        if encoder_arch == "edsr":
            self.encoder = _EDSR(in_channels, encoder_channels)
        else:
            self.encoder = _EDSR(in_channels, encoder_channels)

        mlp_channels = int(encoder_channels * 9)
        mlp_channels += 2  # Attach coord
        mlp_channels += 2  # Attach cell
        self.mlp = _MLP(mlp_channels, out_channels, [256, 256, 256, 256])


    def forward(self, x: Tensor, x_coord: Tensor, x_cell: Tensor = None) -> Tensor:
        return self._forward_impl(x, x_coord, x_cell)

    # Support torch.script function.

    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(features.shape[0],
                                                                         features.shape[1] * 9,
                                                                         features.shape[2],
                                                                         features.shape[3])
        # print("after feature unfold",features.shape)
        # Field radius (global: [-1, 1])
```

```
    # 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).expand(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
```

```
class _ResidualConvBlock(nn.Module):

    def __init__(self, channels: int) -> None:
        super(_ResidualConvBlock, self).__init__()
        self.rcb = nn.Sequential(
            nn.Conv2d(channels, channels, (3, 3), (1, 1), (1, 1)),
            nn.ReLU(True),
            nn.Conv2d(channels, channels, (3, 3), (1, 1), (1, 1)),
        )


    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.rcb(x)

        out = torch.mul(out, 0.1)
        out = torch.add(out, identity)

        return out
```

```python
class _EDSR(nn.Module):

    def __init__(
            self,
            in_channels: int = 3,
            out_channels: int = 64,
            channels: int = 64,
            num_blocks: int = 16,
    ) -> None:
        super(_EDSR, self).__init__()
        # First layer
        self.conv1 = nn.Conv2d(in_channels, channels, (3, 3), (1, 1), (1, 1))

        # Residual blocks
        trunk = []
        for _ in range(num_blocks):
            trunk.append(_ResidualConvBlock(channels))
        self.trunk = nn.Sequential(*trunk)

        # Second layer
        self.conv2 = nn.Conv2d(channels, out_channels, (3, 3), (1, 1), (1, 1))


    def forward(self, x: Tensor) -> Tensor:
        out1 = self.conv1(x)
        out = self.trunk(out1)
        out = self.conv2(out)
        out = torch.add(out, out1)

        return out
```

```python
class _MLP(nn.Module):

    def __init__(self, in_channels: int, out_channels: int, hidden_channels_list: list[int, int, int]):
        super(_MLP, self).__init__()
        layers = []

        last_channels = in_channels
        for hidden in hidden_channels_list:
            layers.append(nn.Linear(last_channels, hidden))
            layers.append(nn.ReLU(True))
            last_channels = hidden
        layers.append(nn.Linear(last_channels, out_channels))
        self.layers = nn.Sequential(*layers)
```