计算机视觉实践——练习 3_SRCNN 和 SRGAN 实验报告

目录

1 实验目的

- 实现超分辨率算法 SRCNN 与 SRGAN。
- 对 SRCNN 与 SRGAN 在数据集 Set5 的测试结果进行分析。
- 对 SRCNN 与 SRGAN 在数据集 Set5 的测试结果进行对比分析。

2 SRCNN

2.1 模型结构

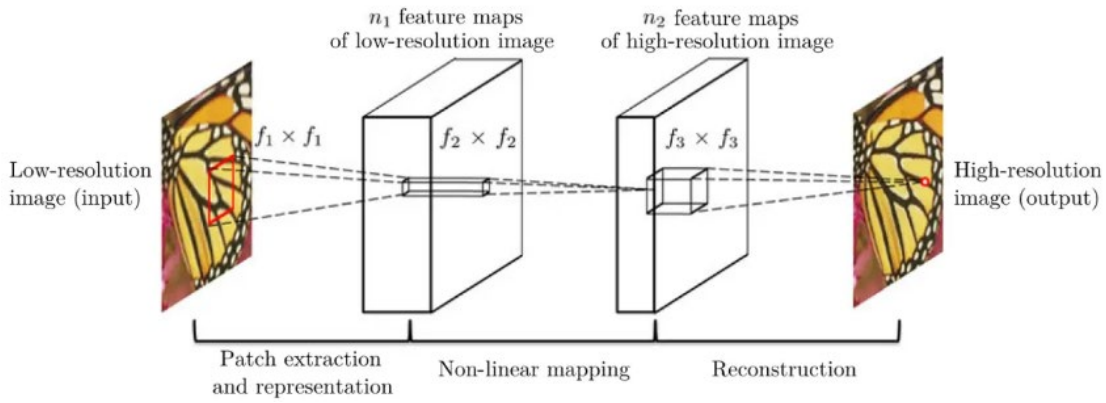

图 1 LeNet5 框架

SRCNN 中的主干网络为 CNN，激活函数为 ReLU。为符合本次实验的要求，对 SRCNN 的最后加入了上采样层，这样可使经 BICUBIC 下采样的测试图片输出的尺寸还原。

2.2 损失函数

本次 SRCNN 实验选取的损失函数为均方损失函数（MSELoss），由 Pytorch 中的 nn. MSELoss 实现。

MSELoss 公式如下：

$$\mathrm{loss}(\mathbf{x}_i, \mathbf{y}_i) = (\mathbf{x}_i - \mathbf{y}_i)^2$$

3 SRGAN

3.1 模型结构

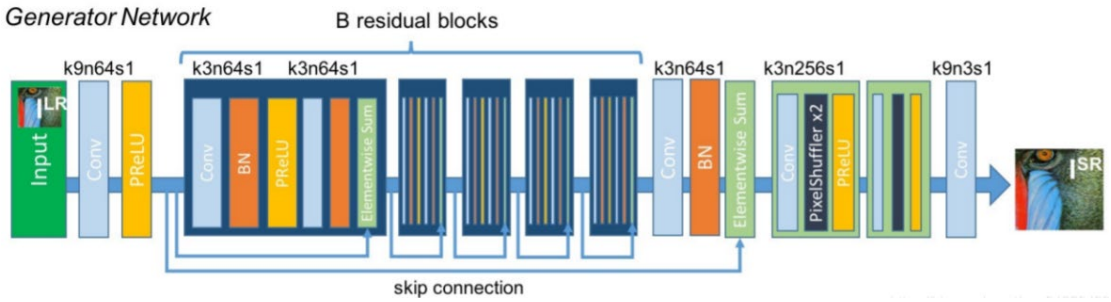

图 2 Generator 框架

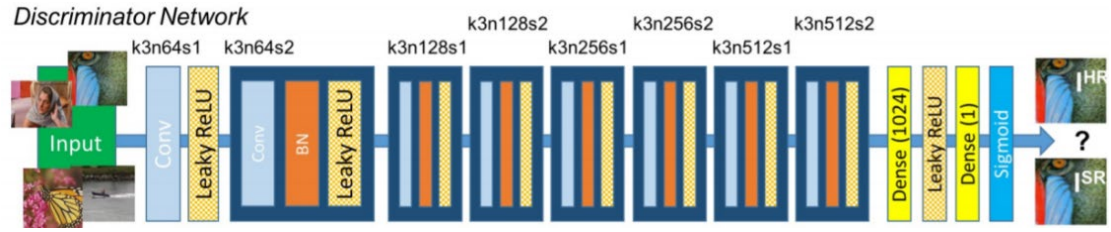

图 3 Discriminator 框架

生成器网络主干网络为 ResNet，激活函数为 PReLU；判别器的主干网络为 CNN，激活函数为 LeakyReLU，本次实验中将全连接层改为全局池化层。

3.2 损失函数

生成器的损失函数为：

$$\hat{\theta}_G = argmin_{\theta_G} \frac{1}{N} \sum_{n=1}^{N} l^{SR}(G_{\theta_G}(I_n^{LR}), I_n^{HR})$$

本次实验的代码中，l^{SR}为感知损失函数，使用 VGG 网络对生成图片与原图预测并对预测结果用 MSELoss 计算；另外还有使用 MSELoss 的图片损失函数，使用 L1Loss 的判别损失函数，以及正则化损失函数，生成器的损失函数为其之和。

判别器的损失函数：

$$\hat{\theta}_D = E_{p(I^{HR})}[logD_{\theta_D}(I^{LR})] + E_{q(I^{LR})}[log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))]$$

本次实验的代码中，使用 BCELoss 作为判别器的损失函数。

4 实验

4.1 数据集

本次实验的训练集为随机挑选的 350 张图，测试集为 Set5。

3.2 实验细节

对训练集进行了数据增强——中心裁剪，并对输入的图片使用 BICUBIC 下采样 4 倍。训练的轮次为 300，每次训练的 mini-batch 为 256，对训练的数据进行打乱。SRCNN 中，对模型的不同层使用不同的学习率，前两层卷积层的学习率为 0.0001，第三层卷积层、上采样层以及整体的的学习率为 0.00001；SRGAN 中生成器的学习率为 0.00001，判别器的学习率为 0.0001。

3.3 实验结果

本次实验中的测试图与还原图如下。

图 4 原图

图 5 测试图片复原（左图为 SRCNN，右图为 SRGAN）

3.4 实验分析与比较

本次实验的 SRCNN 与原论文的结构有所不同，由于加入了上采样层以应对测试集的下采样，故所生成的高分辨率图片有部分区域失真，整体上还是有一些分辨率的恢复。

本次实验的 SRGAN 所生成的图片还原的还可以。

SRCNN 与 SRGAN 生成图片效果的不同，我想有这几点原因。SRCNN 使用的网络是简单的 CNN 网络，且未使用归一化等操作，同时本次实验与原论文不同，本次实验是对测试图片进行下采样，之后再上采样，故 SRCNN 的效果就没有那么好了。而 SRGAN 使用了比较先进的网络 GAN，对生成的图片会进行相

应的判别以增强分辨率的还原效果，同时其网络深度相对 SRCNN 而言很深且有 ResNet 防止梯度消失，可以提取到不同的有用特征，其损失函数也更好，对不同的特征进行不同的损失计算能够更好地优化训练效果。

附录——代码展示

仅展示主要代码。

SRCNN 模型代码：

```
class SRCNN(nn.Module):
    iAffected *
    def __init__(self, in_channels=1, scale_factor=4):
        super().__init__()
        scale_resize = int(log2(scale_factor))
        self.conv1 = nn.Conv2d(in_channels, 64, 9, 1, 9 // 2)
        self.conv2 = nn.Conv2d(64, 32, 5, 1, 5 // 2)
        self.conv3 = nn.Conv2d(32, in_channels * scale_factor, 5, 1, 5 // 2)
        self.relu = nn.ReLU(inplace=True)
        self.upsample = nn.PixelShuffle(scale_resize)

    iAffected *
    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.upsample(self.conv3(x))
        return x
```

SRGAN 模型代码：

```
class Bottleneck(nn.Module):
    新 *
    def __init__(self, in_channels=64, out_channels=64, shortcut=True, e=1):
        super().__init__()
        _out_channels = int(out_channels * e)

        self.res_block = nn.Sequential(
            nn.Conv2d(in_channels, _out_channels, 3, 1, 1),
            nn.BatchNorm2d(_out_channels),
            nn.PReLU(),

            nn.Conv2d(_out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels)
        )
        self.add = shortcut and in_channels == out_channels

    新 *
    def forward(self, x):
        return x + self.res_block(x) if self.add else self.res_block(x)
```

```python
class Upsample(nn.Module):
    新*
    def __init__(self, in_channels, scale_resize):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, in_channels * (scale_resize ** 2), 3, 1, 1)
        self.upsample = nn.PixelShuffle(scale_resize)
        self.act = nn.PReLU()

    新*
    def forward(self, x):
        return self.act(self.upsample(self.conv(x)))
```

```python
class Generator(nn.Module):
    新*
    def __init__(self, in_channels=3, scale_factor=4, nb=6):
        super().__init__()
        scale_resize = int(log2(scale_factor))  # number of upsample_block

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, 64, 9, 1, 4),
            nn.PReLU(),
        )
        self.residuals = nn.Sequential(*[Bottleneck() for _ in range(nb)])  # nb: number of bottleneck
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, 3, 1, 1),
            nn.BatchNorm2d(64),
        )
        self.upsamples = nn.Sequential(
            Upsample(64, scale_resize),
            nn.Conv2d(64, 3, 9, 1, 4),
        )

    新*
    def forward(self, x):
        y1 = self.conv1(x)
        y2 = self.residuals(y1)
        y = self.upsamples(y1 + y2)

        return (torch.tanh(y) + 1) / 2
```

```python
class Discriminator(nn.Module):
    新*
    def __init__(self, in_channels=3):
        super(Discriminator, self).__init__()
        self.conv_net = nn.Sequential(
            nn.Conv2d(in_channels, 64, 3, 1, 1),
            nn.LeakyReLU(0.2),

            nn.Conv2d(64, 64, 3, 2, 1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),

            nn.Conv2d(64, 128, 3, 1, 1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),

            nn.Conv2d(128, 128, 3, 2, 1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),

            nn.Conv2d(128, 256, 3, 1, 1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),

            nn.Conv2d(256, 256, 3, 2, 1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),

            nn.Conv2d(256, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),

            nn.Conv2d(512, 512, 3, 2, 1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),

            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(512, 1024, 1, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(1024, 1, 1, 1),
        )

    新*
    def forward(self, x):
        return torch.sigmoid(self.conv_net(x).view(x.size(0)))  # x.size(0): batch_size
```

SRGAN 损失函数代码：

```python
# Total Variation Loss
class TVLoss(nn.Module):
    def __init__(self, tv_loss_weight=1):
        super(TVLoss, self).__init__()
        self.tv_loss_weight = tv_loss_weight

    def forward(self, x):
        batch_size = x.size()[0]
        h_x = x.size()[2]
        w_x = x.size()[3]
        count_h = self.tensor_size(x[:, :, 1:, :])
        count_w = self.tensor_size(x[:, :, :, 1:])
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()
        return self.tv_loss_weight * 2 * (h_tv / count_h + w_tv / count_w) / batch_size

    @staticmethod
    def tensor_size(t):
        return t.size()[1] * t.size()[2] * t.size()[3]
```

```python
class GeneratorLoss(nn.Module):
    def __init__(self):
        super().__init__()
        vgg = torchvision.models.vgg.vgg16(weights='VGG16_Weights.IMAGENET1K_V1')
        loss_net = nn.Sequential(*list(vgg.features)[:31]).eval()
        for param in loss_net.parameters():
            param.requires_grad = False
        self.loss_net = loss_net.to(device)
        self.mse_loss = nn.MSELoss()
        self.tv_loss = TVLoss()

    def forward(self, fakes, p, targets):
        # p: (G)fakes: fake_images, (D)probabilities of real_images, targets: target_images(high-resolution)
        image_loss = self.mse_loss(fakes, targets)  # Image Loss
        adversarial_loss = torch.mean(1 - p)  # Adversarial Loss
        perception_loss = self.mse_loss(self.loss_net(fakes), self.loss_net(targets))  # Perception Loss
        tv_loss = self.tv_loss(fakes)  # TV Loss

        return image_loss + 0.001 * adversarial_loss + 0.006 * perception_loss + 2e-8 * tv_loss
```

在 15ms 中执行，16 Apr at 16:39:04

```python
class DiscriminatorLoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.bce_loss = nn.BCELoss()

    def forward(self, p, p_gt):
        # p: probability of real_image, p_gt: probability(ground truth) of real_image
        return self.bce_loss(p, torch.zeros_like(p, device=device)) + self.bce_loss(p_gt, torch.ones_like(p_gt, device=device))
```