计算机视觉应用与实践（三）

目录

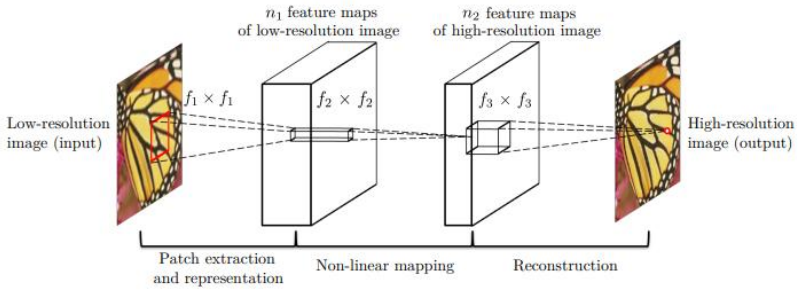
一、 实验目的

1、 实现 SRCNN 或其他一种基于逐像素损失的图像超分辨率算法，并在 Set5 数据集上进行测试。

2、 实现 SRGAN 或其他一种基于 GAN 的图像超分辨率算法，并在 Set5 数据集上进行测试。

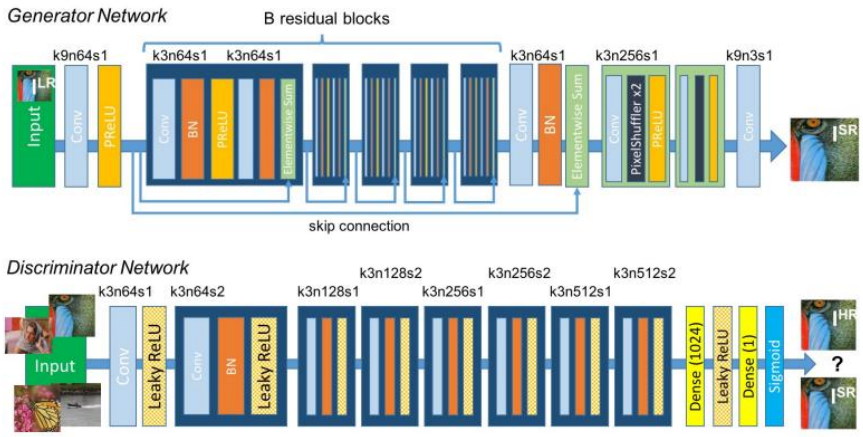
二、 实验原理

SRCNN（Super-Resolution Convolutional Neural Network）是一种基于深度学习的超分辨率图像重建方法。它通过学习低分辨率图像与其对应的高分辨率图像之间的映射关系，从而能够将低分辨率图像重建成高分辨率图像。它的网络结构图像如下所示。



SRCNN 共有三层卷积网络构成，第一层用于对输入的低分辨率图像进行特征提取，以便捕捉图像的局部特征。第二层对于提取的特征做非线性映射。最后一层重建得到高清图像。

SRGAN（Super-Resolution Generative Adversarial Network）是一种基于生成对抗网络的超分辨率图像重建方法。它通过引入判别器和生成器两个神经网络，以对抗的方式学习低分辨率图像和高分辨率图像之间的映射关系，从而能够生成更加逼真的高分辨率图像。SRGAN 的核心思想是在传统的超分辨率图像重建方法的基础上，通过对抗训练的方式进一步提高重建图像的真实感和细节，从而取得更加优秀的效果。它的网络结构图如下所示。



三、 数据集

本次实验使用的训练集是 DIV2K，测试集是 Set5。DIV2K 是一种广泛用于图像超分辨率研究的数据集，它由 800 个不同主题、不同场景、不同光照条件下的高质量图像组成，每个图像的分辨率为 2K 或 4K。这些图像来自于不同的源，包括互联网、数码相机和摄像机，具有多样性和真实性。Set5 是一种常用于图像超分辨率研究的小型基准数据集，其中包含 5 张经典图像，分别是“baby”、“bird”、“butterfly”、“head”和“woman”。这些图像分别涵盖了人像、动物和自然风景等多种场景，每个图像的分辨率为 512x512。Set5 数据集可以用于评估和比较不同的图像超分辨率算法的性能，并且其规模小、易于处理，是算法快速迭代和调试的理想数据集之一。

四、 程序代码

SRCNN 网络代码

```
class SRCNN(nn.Module):
    def __init__(self, num_channels=1):
        super(SRCNN, self).__init__()
        self.conv1 = nn.Conv2d(num_channels, 64, kernel_size=9, padding
=9 // 2)
        self.conv2 = nn.Conv2d(64, 32, kernel_size=5, padding=5 // 2)
        self.conv3 = nn.Conv2d(32, num_channels, kernel_size=5, padding
=5 // 2)
        self.relu = nn.ReLU(inplace=True)

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

SRGAN 网络代码

```
class FeatureExtractor(nn.Module):
    def __init__(self):
        super(FeatureExtractor, self).__init__()
        vgg19_model = vgg19(pretrained=True)
        self.feature_extractor = nn.Sequential(*list(vgg19_model.featur
es.children())[:18])

    def forward(self, img):
        return self.feature_extractor(img)


class ResidualBlock(nn.Module):
    def __init__(self, in_features):
        super(ResidualBlock, self).__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1,
 padding=1),
            nn.BatchNorm2d(in_features, 0.8),
            nn.PReLU(),
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1,
 padding=1),
            nn.BatchNorm2d(in_features, 0.8),
        )
```

```
    def forward(self, x):
        return x + self.conv_block(x)


class GeneratorResNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, n_residual_blocks
=16):
        super(GeneratorResNet, self).__init__()

        # First layer
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels, 64, kernel_si
ze=9, stride=1, padding=4), nn.PReLU())

        # Residual blocks
        res_blocks = []
        for _ in range(n_residual_blocks):
            res_blocks.append(ResidualBlock(64))
        self.res_blocks = nn.Sequential(*res_blocks)

        # Second conv layer post residual blocks
        self.conv2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, str
ide=1, padding=1), nn.BatchNorm2d(64, 0.8))

        # Upsampling layers
        upsampling = []
        for out_features in range(2):
            upsampling += [
                # nn.Upsample(scale_factor=2),
                nn.Conv2d(64, 256, 3, 1, 1),
                nn.BatchNorm2d(256),
                nn.PixelShuffle(upscale_factor=2),
                nn.PReLU(),
            ]
        self.upsampling = nn.Sequential(*upsampling)

        # Final output layer
        self.conv3 = nn.Sequential(nn.Conv2d(64, out_channels, kernel_s
ize=9, stride=1, padding=4), nn.Tanh())

    def forward(self, x):
        out1 = self.conv1(x)
        out = self.res_blocks(out1)
        out2 = self.conv2(out)
        out = torch.add(out1, out2)
```

```
        out = self.upsampling(out)
        out = self.conv3(out)
        return out


class Discriminator(nn.Module):
    def __init__(self, input_shape):
        super(Discriminator, self).__init__()

        self.input_shape = input_shape
        in_channels, in_height, in_width = self.input_shape
        patch_h, patch_w = int(in_height / 2 ** 4), int(in_width / 2 **
 4)
        self.output_shape = (1, patch_h, patch_w)

        def discriminator_block(in_filters, out_filters, first_block=Fa
lse):
            layers = []
            layers.append(nn.Conv2d(in_filters, out_filters, kernel_siz
e=3, stride=1, padding=1))
            if not first_block:
                layers.append(nn.BatchNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            layers.append(nn.Conv2d(out_filters, out_filters, kernel_si
ze=3, stride=2, padding=1))
            layers.append(nn.BatchNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        layers = []
        in_filters = in_channels
        for i, out_filters in enumerate([64, 128, 256, 512]):
            layers.extend(discriminator_block(in_filters, out_filters,
first_block=(i == 0)))
            in_filters = out_filters

        layers.append(nn.Conv2d(out_filters, 1, kernel_size=3, stride=1,
 padding=1))

        self.model = nn.Sequential(*layers)

    def forward(self, img):
        return self.model(img)
```

dataset 代码，对于 SRCNN，其输入的 LR 图像需要先上采样到跟 HR 相同

大小再输入到网络中。为了提高对模型的训练效果，我们对于 DIV2k 做了 100 倍的扩充。同时，在训练时，我们将 Div2k 中的输入随机裁剪成 256*256 大小的 patch 进行训练。

```
class ImageDataset(Dataset):
    def __init__(self, gt_size, root, mode='train', upscale=False):
        self.gt_size = gt_size
        self.upscale = upscale
        # Transforms for low resolution images and high resolution imag
es
        self.files = []
        self.trans_norm=transforms.Normalize(config.mean,config.std)
        self.root = root
        if mode == 'train':
            self.files = sorted(os.listdir(self.root))
        elif mode == 'val':
            self.files = sorted(os.listdir(self.root))
        elif mode == 'test':
            self.lq = sorted(os.listdir(os.path.join(self.root, "LRbicx
4")))
            self.files = sorted(os.listdir(os.path.join(self.root, "GTm
od12")))
        self.mode = mode

    def __getitem__(self, index):
        # img_hr = cv2.imread(self.files[index])
        # print(self.files[index])
        index = index%len(self.files)
        if self.mode == 'train':
            img_hr = cv2.imread(os.path.join(self.root, self.files[inde
x])).astype(np.float32) / 255.
            gt_crop_image = utils.random_crop(img_hr, self.gt_size)
            gt_crop_image = utils.random_rotate(gt_crop_image, [90, 180,
 270])
            gt_crop_image = utils.random_horizontally_flip(gt_crop_imag
e, 0.5)
            img_hr = utils.random_vertically_flip(gt_crop_image, 0.5)
            img_lr = cv2.resize(img_hr, dsize=(self.gt_size // 4, self.
gt_size // 4), interpolation=cv2.INTER_CUBIC)
        elif self.mode == 'val':
            img_hr = cv2.imread(os.path.join(self.root, self.files[inde
x])).astype(np.float32) / 255.
            img_hr = utils.center_crop(img_hr, self.gt_size)
```

```
            img_lr = cv2.resize(img_hr, dsize=(self.gt_size // 4, self.
gt_size // 4), interpolation=cv2.INTER_CUBIC)
        else:
            img_lr, img_hr = cv2.imread(os.path.join(self.root, "LRbicx
4", self.lq[index])).astype(
                np.float32) / 255., cv2.imread(os.path.join(self.root,
"GTmod12", self.files[index])).astype(
                np.float32) / 255.
            # print(img_hr.shape)
            # print(img_hr.shape)
        if self.upscale:
            img_lr = cv2.resize(img_lr, dsize=(img_hr.shape[1],img_hr.s
hape[0]), interpolation=cv2.INTER_CUBIC)
        img_hr = cv2.cvtColor(img_hr, cv2.COLOR_BGR2RGB)
        img_lr = cv2.cvtColor(img_lr, cv2.COLOR_BGR2RGB)

        # Convert image data into Tensor stream format (PyTorch).
        # Note: The range of input and output is between [0, 1]
        img_hr = utils.image_to_tensor(img_hr, False, False)
        img_lr = utils.image_to_tensor(img_lr, False, False)
        if self.mode != 'test':
            img_hr = self.trans_norm(img_hr)
            img_lr = self.trans_norm(img_lr)
        return {"lr": img_lr, "hr": img_hr, 'hr_path': self.files[index]
}

    def __len__(self):
        if self.mode !='test':
            return len(self.files)*100
        else:
            return len(self.files)
```

五、 实验结果

训练细节

本次的训练任务是超分领域常见的 4 倍放大任务。

对于 SRCNN，在实验时使用双三次插值将输入的 LR 图像上采样跟 HR 图像同大小。优化器为 Adam，训练 epoch 为 80

对于 SRGAN，在实验中我们先将它的生成器部分（SRResNet）在 Div2k 上

预训练了 800 次，损失为 L1Loss，然后将生成器和判别器联合训练了 80 次，损失为感知损失，像素损失和对抗损失。

其他细节可以参照代码中的 train_SRCNN.py 和 train_SRGAN.py 部分。

定量分析

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, +∞)，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，其中 SRResNet 为 SRGAN 中预训练好的生成器部分。最优结果加粗表示。

方法	PSNR	SSIM
SRCNN	**27.7541**	0.7979
SRResNet	22.1220	0.7811
SRGAN	26.5206	**0.7982**

可以看出，在 PSNR 指标上 SRCNN 要比其他两种方法高，这是因为 MSELoss 有利于获得较高的 PSNR 值。在 SSIM 指标上，SRGAN 要比其他的两种方法要高。

定性分析

为了更好的评价超分结果，我们对于实验结果做了可视化，如下图所示。具体的超分结果可以在 result 文件夹下。

根据图中的结果，可以看出除了直接双三次插值上采样的方法之外，其余三种方法恢复的都不错。其中，SRCNN 的恢复结果相对更加平滑，这可能是因为在其训练时使用了 MSELoss，但是因为其网络结构较浅，因此效果比其他两种方法要差一点。而相较于 SRResNet，SRGAN 引入了 GAN 来增强生成图像的真实性和细节，因此效果最接近于 GT。

六、 实验总结

通过这次实验，我复现了 SRCNN 和 SRGAN 两种不同的超分辨率算法。SRCNN 是一种单纯的卷积神经网络，通过端到端的学习方式，将低分辨率图像映射到高分辨率图像，生成图像质量较高。SRGAN 则采用了对抗生成网络（GAN）的框架，通过生成器和判别器之间的博弈，不断提升生成图像的质量。在训练过程上，SRCNN 可以直接端到端的训练，而 SRGAN 则需要先对生成器进行预训练，然后在跟判别器一起联合训练以生成更加真实和细节丰富的图像。在生成图像质量方面，SRCNN 和 SRGAN 也有所不同。SRCNN 虽然可以生成高分辨率的图像，但其图像质量可能存在一定的平滑化和模糊化问题，尤其在超分辨率倍数较大的情况下。而 SRGAN 则可以生成更加真实和细节丰富的图像，其生成图像的质量与原始高分辨率图像非常接近，同时还可以保持图像的纹理和细节。通过这次实验，我收获很大。