计算机视觉实践实验报告

目录

计算机视觉实践实验报告.....1
一. 实验目的.....1
二. 实验原理.....1
三. 实验步骤.....3
四. 数据集.....3
五. 程序代码.....4
六. 实验结果.....7
七. 实验分析与总结.....8

一. 实验目的

- 实现SRCNN在Set5数据集上的测试，得到超分辨图像，并进行分析。
- 实现SRGAN在Set5数据集上的测试，得到超分辨图像，并进行分析。
- 对比两种类型的图像超分辨率方法在训练过程和生成图像质量上的不同，报告两者对比分析。

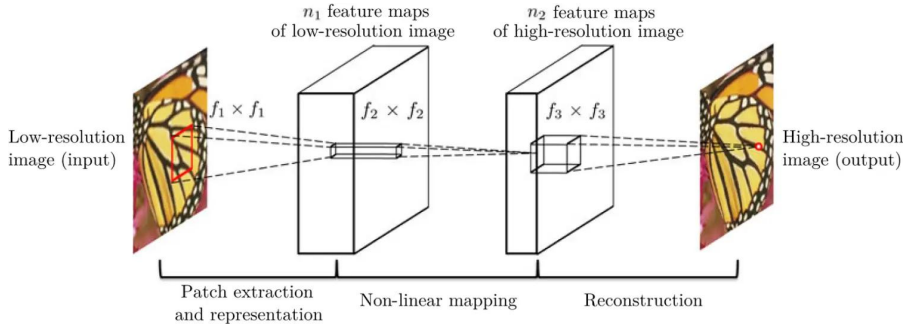
二. 实验原理

图像超分辨率重建：指通过低分辨率图像或图像序列恢复出高分辨率图像。高分辨率图像意味着图像具有更多的细节信息、更细腻的画质,，这些细节在高清电视、医学成像、遥感卫星成像等领域有着重要的应用价值。

2.1 SRCNN

2.1.1 框架

下图为SRCNN的框架，SRCNN将深度学习与传统稀疏编码之间的关系作为依据，将3层网络划分为图像块提取(Patch extraction and representation)、非线性映射(Non-linear mapping)以及最终的重建(Reconstruction)。



2.1.2 流程

（1）先将低分辨率图像使用双三次差值放大至目标尺寸（如放大至2倍、3倍、4倍），此时仍然称放大至目标尺寸后的图像为低分辨率图像(Low-resolution image)，即图中的输入(input)；

（2）将低分辨率图像输入三层卷积神经网络，（举例：在论文中的其中一实验相关设置，对YCrCb颜色空间中的Y通道进行重建，网络形式为(conv1+relu1)—(conv2+relu2)—(conv3)）第一层卷积：卷积核尺寸9×9(f1×f1)，卷积核数目64(n1)，输出64张特征图；第二层卷积：卷积核尺寸1×1(f2×f2)，卷积核数目32(n2)，输出32张特征图；第三层卷积：卷积核尺寸5×5(f3×f3)，卷积核数目1(n3)，输出1张特征图即为最终重建高分辨率图像。

2.1.3 特征提取层

特征提取层用了一层的CNN以及ReLU去将图像Y变成一堆堆向量，即feature map：

$$F_1(Y) = \max(0, W_1 \cdot Y + B_1)$$

其中W_1，B_1是滤波器(卷积核)的参数，通道数为Y的通道c，一共有n_1个滤波器。

2.1.4 非线性映射层

这一层就是将上一层的feature map再用卷积核过滤一次以及ReLU层进行激活，也可以理解为为了加深网络从而更好的学习函数F(·)：

$$F_2(Y) = \max(0, W_2 \cdot F_1(X) + B_2).$$

2.1.5 图像重建层

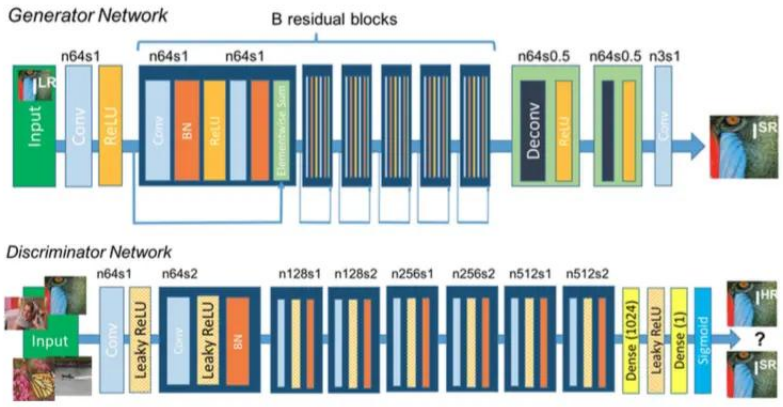
借鉴于传统超分的纯插值办法：对图像局部进行平均化的思想，其本质就是乘加结合的方式，因此作者决定采用卷积的方式(也是乘加结合的方式)去做重建：

$$F(Y) = W_3 \cdot F_2(Y) + B_3.$$

2.2 SRGAN

2.2.1 框架

生成网络由残差结构组成，并结合BN，生成的图像丢到判别网络中，交给判别器判断是生成的高分辨率图像还是真正的高分辨率图像。



2.2.2 生成网络部分

从上图来看，SRGAN网络的生成网络部分就是一个以B个Resnet块组成的深度网络。其中比如“k9n64s1”指的是n = 64个9 × 9，stride为1的卷积核。既然用到了Resnet，自然主要目的就是使用skip connection来加强信息跨层之间的流动以及防止网络深度的加深导致的梯度消失问题。单看SRResNet的结构和SRDenseNet类似，分为低层特征提取、高层特征提取、反卷积(转置卷积)层以及最后的CNN重建层。

2.2.3 判别网络部分

SRGAN网络的判别网络部分就是为了训练最小最大问题的maximization部分，它就是很普通的一个CNN网络，其中激活函数使用Leaky-ReLU(α=0.2)来防止一些负性输出坏死；此外，网络的末端使用了Dense块再接sigmoid函数做一个二分类。整体的判别网络就是一个没有池化层的VGG网络，其中每经过一次跨步卷积(主要为了减少冗余信息的计算)，图像的size就会减小，接着下一层feature map的数量就会翻倍。

2.2.4 感知损失函数

接下来介绍生成网络的Loss function——感知损失函数。在之前我们的Loss一般都是MSE，但是正如之前所说的MSE无法很好的恢复图像的细节，故我们改采用感知损失：

$$l^{SR} = \underbrace{l_X^{SR}}_{\text{content loss}} + \overbrace{10^{-3} \cdot l_{Gen}^{SR}}^{\text{adversarial loss}}.$$

内容损失有2种方案：pixel-wise级的MSE损失，feature-map-wise级的VGG损失。
首先是MSE损失，之所以还启用MSE损失，是因为PSNR也是我们比较看重的一个点，我们强调肉眼感知上的高分辨率，但也不能少了PSNR的评价，因此MSE可作为总体loss的一部分：

$$l_X^{SR} = l_{MSE}^{SR} = \frac{1}{r^2 \cdot W \cdot H} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2.$$

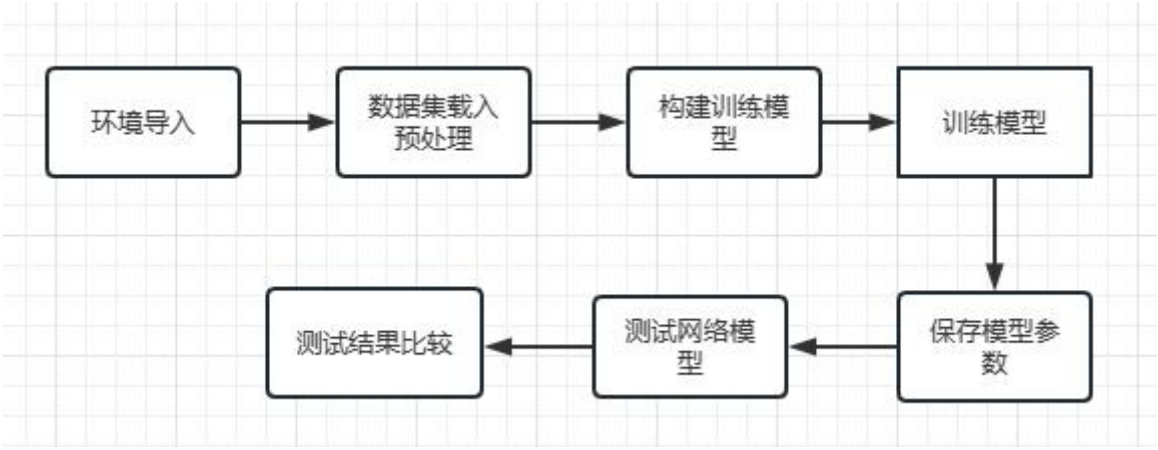
然后是VGG损失，所谓的VGG损失是作者采用预训练好的VGG-19网络的特征向量，使得生成网络的结果通过VGG某一层之后产生的feature map和标签通过VGG网络产生的feature map做loss：

$$l_X^{SR} = l_{VGG/(i,j)}^{SR} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y} - \phi_{i,j}(I^{HR})_{x,y})^2.$$

对抗损失函数就是GAN中常用的形式，我们要最小化：

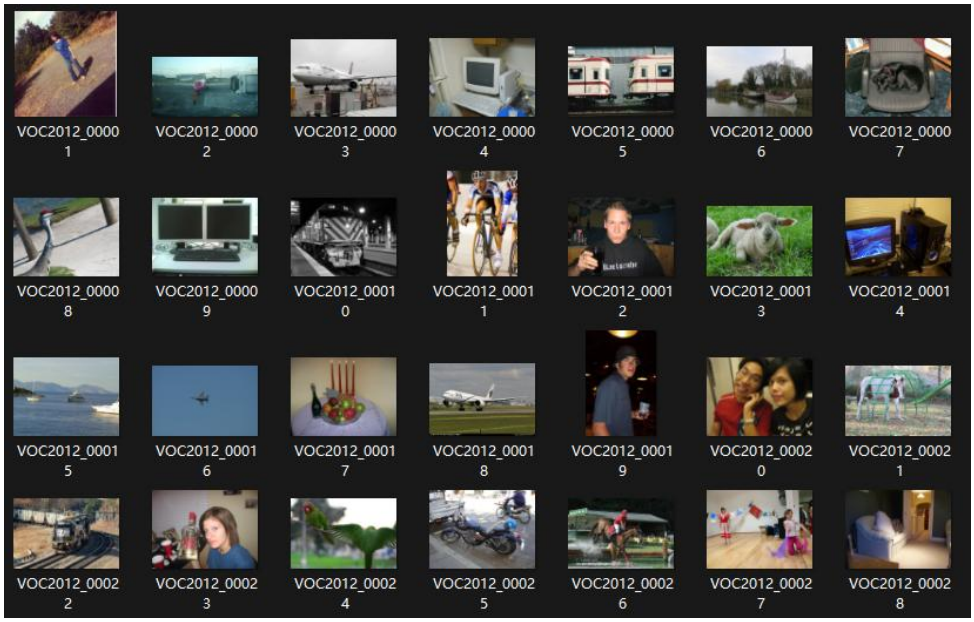
$$l_{Gen}^{SR} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR})).$$

三. 实验步骤



四. 数据集

本实验训练集采用是VOC2012中提取的图像，总共16700张图片作为训练集，验证集425张图像。



最后在Set-5数据集上进行测试。

五. 程序代码

- SRCNN网络模型构建，构建卷积层，ReLU激活函数。

```python
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init


class SRCNN(nn.Module):
    def __init__(self):
        super(SRCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=9, padding=4)
        self.conv2 = nn.Conv2d(64, 32, kernel_size=1, padding=0)
        self.conv3 = nn.Conv2d(32, 1, kernel_size=5, padding=2)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.relu(self.conv2(out))
        out = self.conv3(out)

        return out
```

- SRGAN网络模型构建，生成网络构建。

```python
class Generator(nn.Module):
    def __init__(self, scale_factor):
        upsample_block_num = int(math.log(scale_factor, 2))

        super(Generator, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, padding=4),
            nn.PReLU()
        )
        self.block2 = ResidualBlock(64)
        self.block3 = ResidualBlock(64)
        self.block4 = ResidualBlock(64)
        self.block5 = ResidualBlock(64)
        self.block6 = ResidualBlock(64)
        self.block7 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.PReLU()
        )
        block8 = [UpsampleBLock(64, 2) for _ in range(upsample_block_num)]
        block8.append(nn.Conv2d(64, 3, kernel_size=9, padding=4))
        self.block8 = nn.Sequential(*block8)

    def forward(self, x):
        block1 = self.block1(x)
        block2 = self.block2(block1)
        block3 = self.block3(block2)
        block4 = self.block4(block3)
        block5 = self.block5(block4)
        block6 = self.block6(block5)
        block7 = self.block7(block6)
        block8 = self.block8(block1 + block7)

        return (F.tanh(block8) + 1) / 2
```

- SRGAN网络模型构建，判别网络构建。

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2),

            nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),

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

            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),

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

            nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),

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

            nn.Conv2d(512, 512, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),

            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(512, 1024, kernel_size=1),
```

- SRGAN构建残差块和上采样块。

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.prelu = nn.PReLU()
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

    def forward(self, x):
        residual = self.conv1(x)
        residual = self.bn1(residual)
        residual = self.prelu(residual)
        residual = self.conv2(residual)
        residual = self.bn2(residual)

        return x + residual


class UpsampleBLock(nn.Module):
    def __init__(self, in_channels, up_scale):
        super(UpsampleBLock, self).__init__()
        self.conv = nn.Conv2d(in_channels, in_channels * up_scale ** 2, kernel_size=3, padding=1)
        self.pixel_shuffle = nn.PixelShuffle(up_scale)
        self.prelu = nn.PReLU()

    def forward(self, x):
        x = self.conv(x)
        x = self.pixel_shuffle(x)
        x = self.prelu(x)
        return x
```

- SRCNN训练前数据集载入，损失以及优化器的设置。

```
trainset = DatasetFromFolder("D:/BaiduNetdiskDownload/VOC2012/VOC2012/train", zoom_factor=args.zoom_factor)
testset = DatasetFromFolder("D:/BaiduNetdiskDownload/VOC2012/VOC2012/val", zoom_factor=args.zoom_factor)

trainloader = DataLoader(dataset=trainset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
testloader = DataLoader(dataset=testset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)

model = SRCNN().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(  # we use Adam instead of SGD like in the paper, because it's faster
    [
        {"params": model.conv1.parameters(), "lr": 0.0001},
        {"params": model.conv2.parameters(), "lr": 0.0001},
        {"params": model.conv3.parameters(), "lr": 0.00001},
    ], lr=0.00001,
)
```

- SRCNN模型的训练以及保存。

```python
for epoch in range(args.nb_epochs):

    # Train
    epoch_loss = 0
    for iteration, batch in enumerate(trainloader):
        input, target = batch[0].to(device), batch[1].to(device)
        optimizer.zero_grad()

        out = model(input)
        loss = criterion(out, target)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()

    print(f"Epoch {epoch}. Training loss: {epoch_loss / len(trainloader)}")

    # Test
    avg_psnr = 0
    with torch.no_grad():
        for batch in testloader:
            input, target = batch[0].to(device), batch[1].to(device)

            out = model(input)
            loss = criterion(out, target)
            psnr = 10 * log10(1 / loss.item())
            avg_psnr += psnr
    print(f"Average PSNR: {avg_psnr / len(testloader)} dB.")

    # Save model
    torch.save(model, f"model_{epoch}.pth")
```

- SRGAN模型损失。

```python
class GeneratorLoss(nn.Module):
    def __init__(self):
        super(GeneratorLoss, self).__init__()
        vgg = vgg16(pretrained=True)
        loss_network = nn.Sequential(*list(vgg.features)[:31]).eval()
        for param in loss_network.parameters():
            param.requires_grad = False
        self.loss_network = loss_network
        self.mse_loss = nn.MSELoss()
        self.tv_loss = TVLoss()

    def forward(self, out_labels, out_images, target_images):
        # Adversarial Loss
        adversarial_loss = torch.mean(1 - out_labels)
        # Perception Loss
        perception_loss = self.mse_loss(self.loss_network(out_images), self.loss_network(target_images))
        # Image Loss
        image_loss = self.mse_loss(out_images, target_images)
        # TV Loss
        tv_loss = self.tv_loss(out_images)
        return image_loss + 0.001 * adversarial_loss + 0.006 * perception_loss + 2e-8 * tv_loss
```

- SRCNN训练前参数，数据集载入以及优化器的设置。

```
opt = parser.parse_args()

CROP_SIZE = opt.crop_size
UPSCALE_FACTOR = opt.upscale_factor
NUM_EPOCHS = opt.num_epochs

train_set = TrainDatasetFromFolder('D:/BaiduNetdiskDownload/VOC2012/VOC2012/train', crop_size=CROP_SIZE, upscale_factor=UPSCALE_FACTOR)
val_set = ValDatasetFromFolder('D:/BaiduNetdiskDownload/VOC2012/VOC2012/val', upscale_factor=UPSCALE_FACTOR)
train_loader = DataLoader(dataset=train_set, num_workers=0, batch_size=64, shuffle=True)
val_loader = DataLoader(dataset=val_set, num_workers=0, batch_size=1, shuffle=False)

netG = Generator(UPSCALE_FACTOR)
print('# generator parameters:', sum(param.numel() for param in netG.parameters()))
netD = Discriminator()
print('# discriminator parameters:', sum(param.numel() for param in netD.parameters()))

generator_criterion = GeneratorLoss()

if torch.cuda.is_available():
    netG.cuda()
    netD.cuda()
    generator_criterion.cuda()

optimizerG = optim.Adam(netG.parameters())
optimizerD = optim.Adam(netD.parameters())
```

- SRCNN模型的训练以及测试。

```
for epoch in range(1, NUM_EPOCHS + 1):
    train_bar = tqdm(train_loader)
    running_results = {'batch_sizes': 0, 'd_loss': 0, 'g_loss': 0, 'd_score': 0, 'g_score': 0}

    netG.train()
    netD.train()
    for data, target in train_bar:
        g_update_first = True
        batch_size = data.size(0)
        running_results['batch_sizes'] += batch_size

        ############################
        # (1) Update D network: maximize D(x)-1-D(G(z))
        ###########################
        real_img = Variable(target)
        if torch.cuda.is_available():
            real_img = real_img.cuda()
        z = Variable(data)
        if torch.cuda.is_available():
            z = z.cuda()
        fake_img = netG(z)

        netD.zero_grad()
        real_out = netD(real_img).mean()
        fake_out = netD(fake_img).mean()
        d_loss = 1 - real_out + fake_out
        d_loss.backward(retain_graph=True)
        optimizerD.step()

        ############################
        # (2) Update G network: minimize 1-D(G(z)) + Perception Loss + Image Loss + TV Loss
        ###########################
        netG.zero_grad()
        g_loss = generator_criterion(fake_out, fake_img, real_img)
        g_loss.backward()
        optimizerG.step()
        fake_img = netG(z)
```

六. 实验结果

- 在SRCNN网络以及SRGAN网络上训练数据集，训练100个epoch，保存模型，用于之后对Set-5数据集的测试。

```
Average PSNR: 23.39983332859599 dB.
Epoch 93. Training loss: 0.006448093015197972
Average PSNR: 23.401148661827193 dB.
Epoch 94. Training loss: 0.006446197892791552
Average PSNR: 23.400614259412794 dB.
Epoch 95. Training loss: 0.006445100215419814
Average PSNR: 23.399876584898973 dB.
Epoch 96. Training loss: 0.006444825667864922
Average PSNR: 23.38157700410989 dB.
Epoch 97. Training loss: 0.006444214492786012
Average PSNR: 23.3937226369702 dB.
Epoch 98. Training loss: 0.006442398207591886
Average PSNR: 23.39708343196547 dB.
```

- 保存SRCNN和SRGAN的模型参数后在Set-5数据集上进行测试。

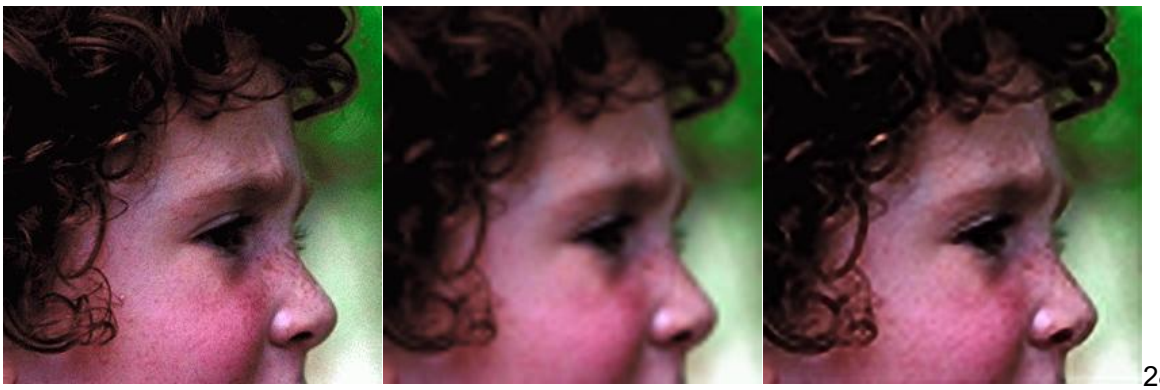
SRCNN实验结果（先用BICUBIC进行下采样，再通过算法处理，与原图算psnr）：

原图 BICUBIC_4x SRCNN_4x psnr

28.16dB

28.95dB

28.11dB


28.93dB


28.14dB

SRGAN实验结果：

原图	BICUBIC_4x	SRGAN_4x	psnr
			33.38dB
			33.11dB
			31.14dB
			31.78dB


32.75dB

七. 实验分析与总结

- 首先由实验结果可见，SRCNN和SRGAN都可以很好地可以将低分辨率的图像放大为高分辨率的图像，以达到更好的视觉效果。
- SRGAN相比于SRCNN不管是从峰值信噪比（PSNR）还是视觉感官上都要比SRCNN的效果更好，究其原因一是SRCNN的模型更为简单，使用的卷积神经网络不论是深度和宽度可能在后续都可以做进一步的改善。而SRGAN它相比SRCNN拥有更复杂的模型，可以通过不断训练生成器和判别器可以达到更好的效果。
- SRCNN正因为它的体量比较小，所以它有着更快的训练速度，而SRGAN由于引入GAN，相比于SRCNN的训练会需要更长的时间。
- 后续的改进一是可以从数据集入手，可以选择更适合超分辨任务的数据集，二是SRCNN可以尝试更深的网络，更先进的优化器以及损失函数来进一步提高模型的性能。