# 计算机视觉实践报告（三）

## 1.实验目的

* 实现SRCNN或其他一种基于逐像素损失的图像超分辨率算法在Set5数据集上的测试，得到超分辨率图像，并进行分析。
* 实现SRGAN或其他一种基于GAN的图像超分辨率算法在Set5计算机视觉实践作业信感5数据集上的测试，得到超分辨率图像，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。

### 2.实验原理

#### 2.1基于卷积神经网络的图像超分辨率重建的研究流程如下

#### 首先找到一组原始图像Image1；

#### 然后将这组图片降低分辨率为一组图像Image2；

#### 通过各种神经网络结构，将Image2超分辨率重建为Image3（Image3和Image1分辨率一样）

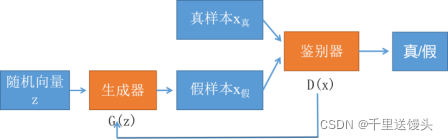
#### 通过PSNR等方法比较Image1与Image3，验证超分辨率重建的效果，根据效果调节神经网络中的节点模型和参数

#### 反复执行，直到第四步比较的结果满意

#### 2.2基于GAN的图像超分辨率

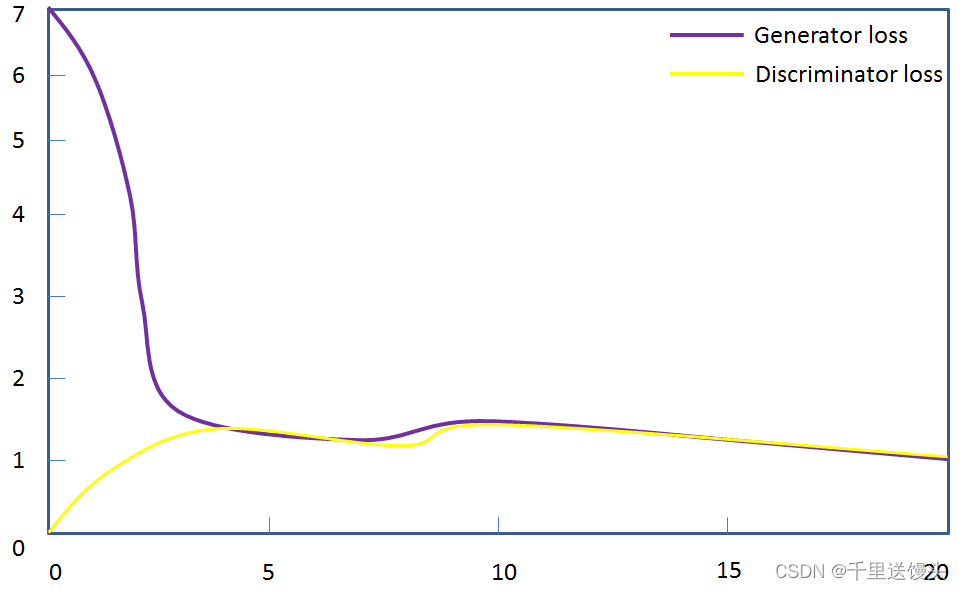
##### 2.2.1 GAN

GAN的全称是Generative adversarial network，中文翻译过来就是生成对抗网络。生成对抗网络其实是两个网络的组合：生成网络（Generator）负责生成模拟数据；判别网络Discriminator）负责判断输入的数据是真实的还是生成的。生成网络要不断优化自己生成的数据让判别网络判断不出来，判别网络也要优化自己让自己判断得更准确。二者关系形成对抗，因此叫对抗网络。



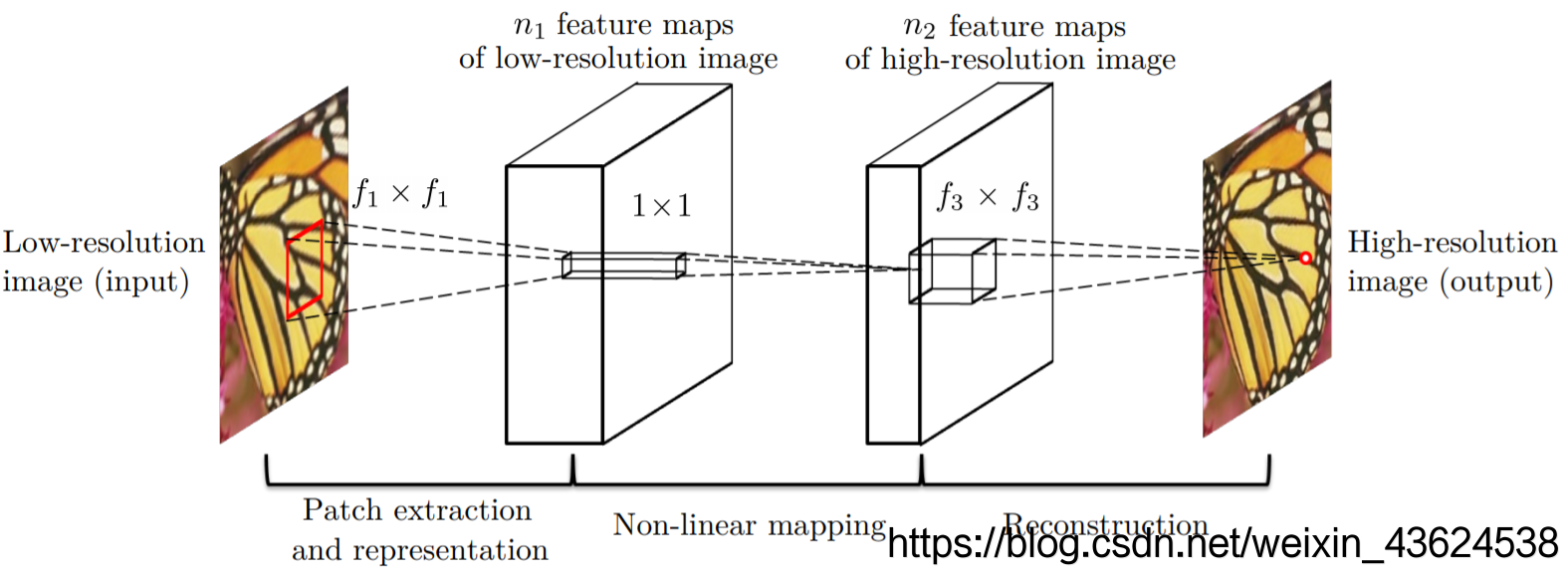
GAN的基本结构图

GAN模型不是一上来就能实现具体功能的，需要经历一个训练的过程。这是一个生成器和判别器博弈的过程。生成器生成假数据，然后将生成的假数据和真数据都输入判别器，判别器要判断出哪些是真的哪些是假的。判别器第一次判别出来的肯定有很大的误差，然后我们根据误差来优化判别器。现在判别器水平提高了，生成器生成的数据很难再骗过判别器了，所以我们得反过来优化生成器，之后生成器水平提高了，然后反过来继续训练判别器，判别器水平又提高了，再反过来训练生成器，就这样循环往复，直到达到纳什均衡。



## 3网络架构

#### 3.1 SRCNN

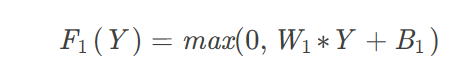


网络架构图

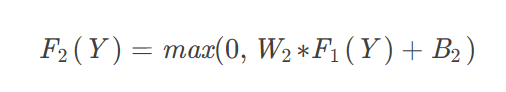
#### 3.1.1 网络介绍

SRCNN的结构较简单,整个卷积网络包括三个卷积层，甚至没有池化和全连接层，但都通过Relu函数三个卷积层使用的卷积核的大小分为为9x9,，1x1和5x5，前两个的输出特征个数分别为64和32：

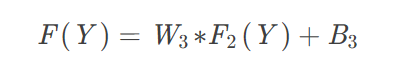
1. 对低分辨率的图进行卷积操作，生成*n*1​维的feature maps



1. 对*n*1维的feature map进行卷积操作生成*n*2维 feature maps



1. 对*n*2维的feature maps进行卷积生成超分辨的图像（反卷积）



分别对应三个处理流程：

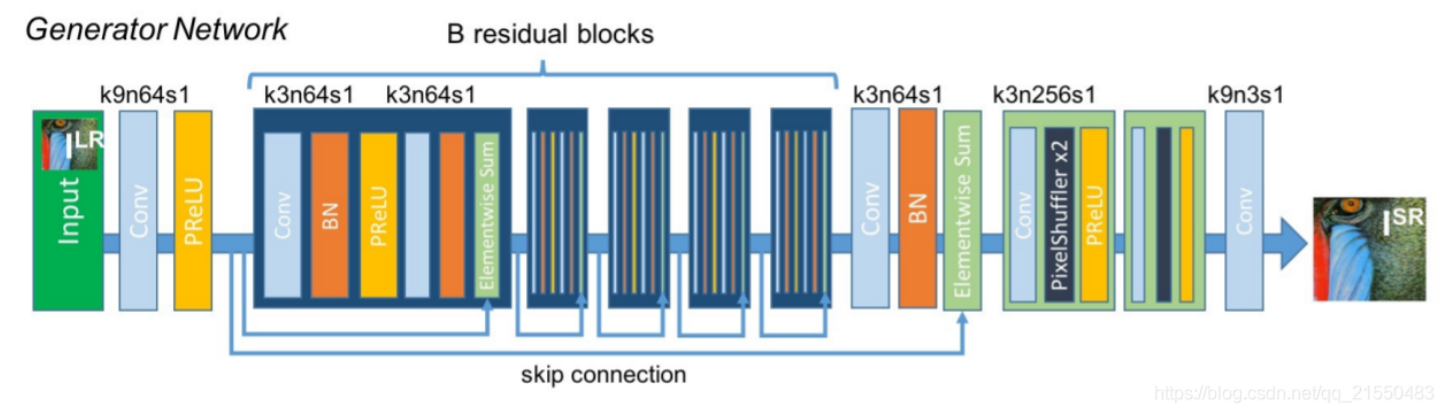
1. 提取图像特征： 从低分辨率图像中提取多个patch图像块，每个块被卷积操作表示为多维的向量（维数等于filter的数量），所有的特征向量组成特征矩阵（feature maps）
2. 非线性映射： 将*n*1维特征矩阵，通过卷积操作实现非线性映射，变成另一*n*2维特征矩阵。
3. 重构图像： 等于是个反卷积的过程，将*n*2的特征矩阵还原为超分辨图像，特别提出的一点：所谓非线性映射指的是地位的LR图像的特征向高维图像HR的特征的映射，而所应用的函数是Relu非线性函数，所以为非线性映射。

**激活函数**：ReLU。  
**损失函数**：MSE（均方误差）。原因可获得高PSNR。  
**PSNR**：一种广泛使用的用于定量评估图像恢复质量的指标。

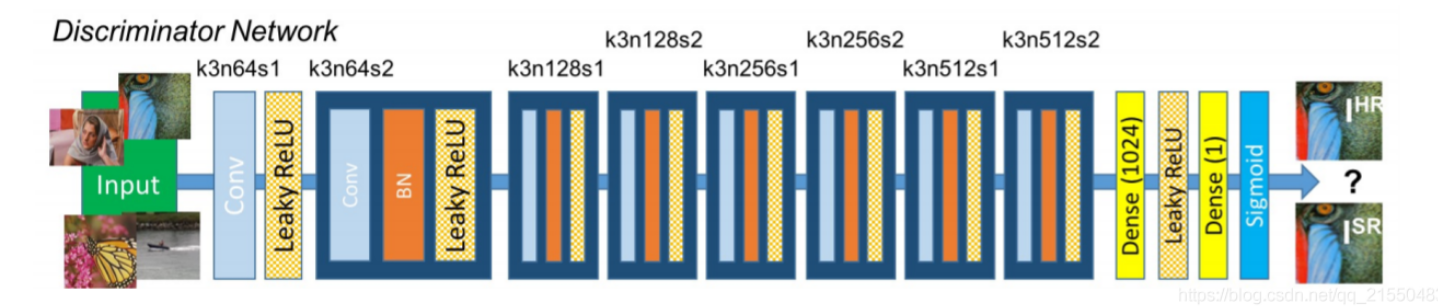
##### 3.1.2训练

从GroundTruth里选择图片并对其进行剪裁从而生成子图片。并对这些字图片应用升尺度因子进行子采样，最后再用双三次插值将其放大到相同大小。损失函数为MSE,通过从具有零均值和标准偏差0.001（和偏差为0）的高斯分布中随机绘制来初始化每个层的滤波器权重。前两层的学习率e-4为最后一层的学习率为e-5。

#### SRGAN



生成网络架构图



辨别网络架构图

##### 网络介绍

**普通的超分辨率模型训练网络时只用到了均方差作为损失函数，虽然能够获得很高的峰值信噪比，但是恢复出来的图像通常会丢失高频细节**。

SRGAN利用**感知损失(perceptual loss)和对抗损失(adversarial loss)来提升恢复出的图片的真实感**。

生成网络的构成如上图所示，生成网络的作用是输入一张低分辨率图片，生成高分辨率图片：

SRGAN的生成网络由三个部分组成。

1. 低分辨率图像进入后会经过一个卷积+RELU函数。
2. 然后经过B个残差网络结构，每个残差结构都包含两个卷积+标准化+RELU，还有一个残差边。
3. 然后进入上采样部分，在经过两次上采样后，原图的高宽变为原来的4倍，实现分辨率的提升。

判别网络的构成如上图所示：

SRGAN的判别网络由不断重复的 卷积+LeakyRELU和标准化 组成。

对于判断网络来讲，它的目的是判断输入图片的真假，它的输入是图片，输出是判断结果。

判断结果处于0-1之间，利用接近1代表判断为真图片，接近0代表判断为假图片。

判断网络的构建和普通卷积网络差距不大，都是不断的卷积对图片进行下采用，在多次卷积后，最终接一次全连接判断结果。

##### 3.2.2训练

SRGAN的训练可以分为生成器训练和判别器训练：

每一个step中一般先训练判别器，然后训练生成器。

一、判别器的训练

在训练判别器的时候我们希望判别器可以判断输入图片的真伪，因此我们的输入就是真图片、假图片和它们对应的标签。

因此判别器的训练步骤如下：

1、随机选取batch\_size个真实高分辨率图片。

2、利用resize后的低分辨率图片，传入到Generator中生成batch\_size个虚假高分辨率图片。

3、真实图片的label为1，虚假图片的label为0，将真实图片和虚假图片当作训练集传入到Discriminator中进行训练。

二、生成器的训练

在训练生成器的时候我们希望生成器可以生成极为真实的假图片。因此我们在训练生成器需要知道判别器认为什么图片是真图片。

因此生成器的训练步骤如下：

1、将低分辨率图像传入生成模型，得到虚假高分辨率图像，将虚假高分辨率图像获得判别结果与1进行对比得到loss。（与1对比的意思是，让生成器根据判别器判别的结果进行训练）。

2、将真实高分辨率图像和虚假高分辨率图像传入VGG网络，获得两个图像的特征，通过这两个图像的特征进行比较获得loss

## 4.数据集

本次实验测试采用SET5数据集。



#### 4.1数据集测试处理

先将图像用Bicubic插值进行下采样，再使用超分辨率算法处理，将得到的超分辨率图像与真实的原始图像进行对比。

[最邻近插值算法](https://juejin.cn/post/6844904054338879496)的目标像素值由源图上单个像素决定，[双线性插值算法](https://juejin.cn/post/6844904054338879501)由源像素某点周围4个像素点按一定权重获得，而双立方插值算法更进一步参考了源像素某点周围4\*4个像素来获得。

## 5.程序代码

#### 5.1 SRCNN代码

class SRCNN(nn.Module): #搭建SRCNN 3层卷积模型，Conve2d（输入层数，输出层数，卷积核大小，步长，填充层）  
 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

#### 5.2训练代码

# 初始参数设定  
parser = argparse.ArgumentParser() # argparse是python用于解析命令行参数和选项的标准模块  
parser.add\_argument('--train-file', type=str, required=True,) # 训练 h5文件目录  
parser.add\_argument('--eval-file', type=str, required=True) # 测试 h5文件目录  
parser.add\_argument('--outputs-dir', type=str, required=True) #模型 .pth保存目录  
parser.add\_argument('--scale', type=int, default=3) # 放大倍数  
parser.add\_argument('--lr', type=float, default=1e-4) #学习率  
parser.add\_argument('--batch-size', type=int, default=16) # 一次处理的图片大小  
parser.add\_argument('--num-workers', type=int, default=0) # 线程数  
parser.add\_argument('--num-epochs', type=int, default=400) #训练次数  
parser.add\_argument('--seed', type=int, default=123) # 随机种子  
args = parser.parse\_args()  
  
# 输出放入固定文件夹里  
args.outputs\_dir = os.path.join(args.outputs\_dir, 'x{}'.format(args.scale))  
# 没有该文件夹就新建一个文件夹  
if not os.path.exists(args.outputs\_dir):  
 os.makedirs(args.outputs\_dir)  
  
# benckmark模式，加速计算，但寻找最优配置，计算的前馈结果会有差异  
cudnn.benchmark = True  
  
# gpu或者cpu模式，取决于当前cpu是否可用  
device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  
  
# 每次程序运行生成的随机数固定  
torch.manual\_seed(args.seed)  
  
# 构建SRCNN模型，并且放到device上训练  
model = SRCNN().to(device)  
  
# 恢复训练，从之前结束的那个地方开始  
# model.load\_state\_dict(torch.load('outputs/x3/epoch\_173.pth'))  
  
# 设置损失函数为MSE  
criterion = nn.MSELoss()  
  
# 优化函数Adam，lr代表学习率，  
optimizer = optim.Adam([  
 {'params': model.conv1.parameters()},  
 {'params': model.conv2.parameters()},  
 {'params': model.conv3.parameters(), 'lr': args.lr \* 0.1}  
], lr=args.lr)  
  
# 预处理训练集  
train\_dataset = TrainDataset(args.train\_file)  
train\_dataloader = DataLoader(  
 # 数据  
 dataset=train\_dataset,  
 # 分块  
 batch\_size=args.batch\_size,  
 # 数据集数据洗牌,打乱后取batch  
 shuffle=True,  
 # 工作进程，像是虚拟存储器中的页表机制  
 num\_workers=args.num\_workers,  
 # 锁页内存，不换出内存，生成的Tensor数据是属于内存中的锁页内存区  
 pin\_memory=True,  
 # 不取余，丢弃不足batchSize大小的图像  
 drop\_last=True)  
# 预处理验证集  
eval\_dataset = EvalDataset(args.eval\_file)  
eval\_dataloader = DataLoader(dataset=eval\_dataset, batch\_size=1)  
  
# 拷贝权重  
best\_weights = copy.deepcopy(model.state\_dict())  
best\_epoch = 0  
best\_psnr = 0.0  
  
# 画图用  
lossLog = []  
psnrLog = []  
  
# 恢复训练  
# for epoch in range(args.num\_epochs):  
for epoch in range(1, args.num\_epochs + 1):  
 # for epoch in range(174, 400):  
 # 模型训练入口  
 model.train()  
  
 # 变量更新，计算epoch平均损失  
 epoch\_losses = AverageMeter()  
  
 # 进度条，就是不要不足batchsize的部分  
 with tqdm(total=(len(train\_dataset) - len(train\_dataset) % args.batch\_size)) as t:  
 # t.set\_description('epoch:{}/{}'.format(epoch, args.num\_epochs - 1))  
 t.set\_description('epoch:{}/{}'.format(epoch, args.num\_epochs))  
  
 # 每个batch计算一次  
 for data in train\_dataloader:  
 # 对应datastes.py中的\_\_getItem\_\_，分别为lr,hr图像  
 inputs, labels = data  
  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
 # 送入模型训练  
 preds = model(inputs)  
  
 # 获得损失  
 loss = criterion(preds, labels)  
  
 # 显示损失值与长度  
 epoch\_losses.update(loss.item(), len(inputs))  
  
 # 梯度清零  
 optimizer.zero\_grad()  
  
 # 反向传播  
 loss.backward()  
  
 # 更新参数  
 optimizer.step()  
  
 # 进度条更新  
 t.set\_postfix(loss='{:.6f}'.format(epoch\_losses.avg))  
 t.update(len(inputs))  
 # 记录lossLog 方面画图  
 lossLog.append(np.array(epoch\_losses.avg))  
 # 可以在前面加上路径  
 np.savetxt("lossLog.txt", lossLog)  
  
 # 保存模型  
 torch.save(model.state\_dict(), os.path.join(args.outputs\_dir, 'epoch\_{}.pth'.format(epoch)))  
  
 # 是否更新当前最好参数  
 model.eval()  
 epoch\_psnr = AverageMeter()  
  
 for data in eval\_dataloader:  
 inputs, labels = data  
  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
  
 # 验证不用求导  
 with torch.no\_grad():  
 preds = model(inputs).clamp(0.0, 1.0)  
  
 epoch\_psnr.update(calc\_psnr(preds, labels), len(inputs))  
  
 print('eval psnr: {:.2f}'.format(epoch\_psnr.avg))  
  
 # 记录psnr  
 psnrLog.append(Tensor.cpu(epoch\_psnr.avg))  
 np.savetxt('psnrLog.txt', psnrLog)  
 # 找到更好的权重参数，更新  
 if epoch\_psnr.avg > best\_psnr:  
 best\_epoch = epoch  
 best\_psnr = epoch\_psnr.avg  
 best\_weights = copy.deepcopy(model.state\_dict())  
  
 print('best epoch: {}, psnr: {:.2f}'.format(best\_epoch, best\_psnr))  
  
 torch.save(best\_weights, os.path.join(args.outputs\_dir, 'best.pth'))  
  
print('best epoch: {}, psnr: {:.2f}'.format(best\_epoch, best\_psnr))  
  
torch.save(best\_weights, os.path.join(args.outputs\_dir, 'best.pth'))

## SRGAN代码

import math  
import torch  
from torch import nn  
  
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(channels)  
 self.conv2 = nn.Conv2d(channels, channels, kernel\_size=3, padding=1)  
 self.bn2 = nn.BatchNorm2d(channels)  
  
 def forward(self, x):  
 short\_cut = x  
 x = self.conv1(x)  
 x = self.bn1(x)  
 x = self.prelu(x)  
  
 x = self.conv2(x)  
 x = self.bn2(x)  
  
 return x + short\_cut  
  
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(in\_channels)  
  
 def forward(self, x):  
 x = self.conv(x)  
 x = self.pixel\_shuffle(x)  
 x = self.prelu(x)  
 return x  
  
class Generator(nn.Module):  
 def \_\_init\_\_(self, scale\_factor, num\_residual=16):  
 upsample\_block\_num = int(math.log(scale\_factor, 2))  
  
 super(Generator, self).\_\_init\_\_()  
  
 self.block\_in = nn.Sequential(  
 nn.Conv2d(3, 64, kernel\_size=9, padding=4),  
 nn.PReLU(64)  
 )  
  
 self.blocks = []  
 for \_ in range(num\_residual):  
 self.blocks.append(ResidualBlock(64))  
 self.blocks = nn.Sequential(\*self.blocks)  
   
 self.block\_out = nn.Sequential(  
 nn.Conv2d(64, 64, kernel\_size=3, padding=1),  
 nn.BatchNorm2d(64)  
 )  
  
 self.upsample = [UpsampleBLock(64, 2) for \_ in range(upsample\_block\_num)]  
 self.upsample.append(nn.Conv2d(64, 3, kernel\_size=9, padding=4))  
 self.upsample = nn.Sequential(\*self.upsample)  
  
 def forward(self, x):  
 x = self.block\_in(x)  
 short\_cut = x  
 x = self.blocks(x)  
 x = self.block\_out(x)  
  
 upsample = self.upsample(x + short\_cut)  
 return torch.tanh(upsample)  
  
  
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),  
 nn.LeakyReLU(0.2),  
 nn.Conv2d(1024, 1, kernel\_size=1)  
 )  
  
 def forward(self, x):  
 batch\_size = x.size(0)  
 return self.net(x).view(batch\_size)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 from torchsummary import summary  
  
 # 需要使用device来指定网络在GPU还是CPU运行  
 device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
 model = Generator(4).to(device)  
 summary(model, input\_size=(3,56,56))

## 6.实验结果

#### 6.1SRCNN结果



#### 6.2SRGAN结果

