实验报告

实验要求：实现一种图像超分辨率方法在Set5数据集上的测试，得到超分辨率图像，测试其与原始真实图像之间的PSNR，SSIM指标值。对所选择方法的细节进行介绍，并试着讨论该方法可能存在的优缺点，以及可能改进的方向。

测试方式：现将图像用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。

1. **概念介绍**

VDSR提出的起源主要是由于SRCNN的局限性，主要表现在以下三个方面：

1、依赖于小图像区域的上下文信息

2、训练收敛太慢

3、网络只适用于单个比例因子

VDSR对其进行的改进：

1、增大了网络的感受野，将SRCNN的感受野由13\*13增加到VDSR的41×41，充分利用了非常大的图像区域的上下问信息。

2、使用了残差学习和极高的学习率，加速姥姥网络的训练

3、单一的网络有效的处理多尺度的SR问题

CDSR与SRCNN的区别：

1、vdsr的输入图像与输出图像一样大小，因为在训练期间对每一层进行了边框扩充（padding=0），而SRCNN的输出小于输入

2、VDSR对所有的层使用相同的学习率，SRCNN不同的层使用不同的学习率，以实现稳定的收敛。

在VDSR网络中用到了残差网络，并且仅且只有一个残差单元组成。

1. **VDSR的神经网络结构**

（1）网络架构

输入层：经过插值得到的LR图像（大小不定）

中间64个卷积层：3\*3\*64的滤波器

输出层：3\*3\*64的滤波器进行图像构建

激活函数：reLU

（2）前向传播

使用0作为padding保证每次经过卷积层输入和输出图像大小不变，既保证深层网络进行卷积的可行性，又能保证图像边缘的像素可以被正确的预测，而不是直接忽视掉。

定义残差图像：r = y - x（真实HR图像-输入的LR图像）

损失函数定义：1/2\*|| r-f(x)||2 这样损失函数的值就不会在0附近了

损耗层：输入是残差估计，网络输入（ILR图像）和真实HR图像。

输出：重建图像（网络输入和残差之和）与真实情况之间的欧几里得距离。

（3）后向传播

小批量梯度下降的方法：64个样本迭代一轮

正则化（抑制过拟合）：权值衰减（使用L2惩罚λw2/2，λ=0.0001）

动量参数：0.9（在下降时保留一部分之前的方向和速度，以此来抑制震荡。若当前的梯度方向与累积的历史梯度方向一致，当前梯度就会被加强，使得这一步的下降幅度变大，反之也成立。用于加速梯度下降）

学习率：每一层都为0.1，每20轮迭代减少一半（比SRCNN的0.0001高了不少，同时SRCNN不同层学习率还不同）

梯度裁剪：[-θ/γ，θ/γ]，其中γ表示当前学习速率。解决梯度爆炸问题（网络太深，不同层变化速度不同，导致梯度太大了，模型不稳定）

1. **VDSR的原理**

（1）越深越好

卷积神经网络通过强制相邻层神经元之间的局部连通性模式来利用空间局部相关性。换句话说，层m中的隐藏单元将层m-1中单元的子集作为输入，形成空间上连续的感受野。每个隐藏单元对于输入相对于感受野之外的变化均无响应。因此，该体系结构确保学习的滤波器对空间局部输入模式产生最强的响应。堆叠许多这样的层导致滤波器变得越来越全局化（即，响应于更大的像素空间区域）所以能够学到更多的信息。

（2）使用残差加速

由于已经有一个低分辨率的图像作为输入，因此对于SR而言，预测高频分量就足够了。同时采用高学习率保证收敛速度。

（3）多尺度训练

用同一个网络同时去学习不同尺度的图像，事实证明这样效果是比较好的。

1. **实验步骤**

（1）核心网络代码：

# 神经网络结构块  
class Conv\_ReLU\_Block(nn.Module):  
 def \_\_init\_\_(self):  
 super(Conv\_ReLU\_Block, self).\_\_init\_\_()  
 # stride步长为1 padding填充为1 bias不添加偏置参数作为可学习参数  
 self.conv = nn.Conv2d(in\_channels=64, out\_channels=64, kernel\_size=3, stride=1, padding=1, bias=False)  
 # 对从上层网络Conv2d中传递下来的tensor直接进行修改，inplace变量替换能够节省运算内存，不用多存储其他变量  
 self.relu = nn.ReLU(inplace=True)  
   
 def forward(self, x):  
 return self.relu(self.conv(x))  
  
# 主要网络结构  
class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.residual\_layer = self.make\_layer(Conv\_ReLU\_Block, 18)  
 self.input = nn.Conv2d(in\_channels=1, out\_channels=64, kernel\_size=3, stride=1, padding=1, bias=False)  
 self.output = nn.Conv2d(in\_channels=64, out\_channels=1, kernel\_size=3, stride=1, padding=1, bias=False)  
 self.relu = nn.ReLU(inplace=True)  
  
 # Conv2d中参数的初始化 normal高斯  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 # 3 3 64 最后一次3 3 1  
 # print(m.kernel\_size[0], m.kernel\_size[1], m.out\_channels)  
 n = m.kernel\_size[0] \* m.kernel\_size[1] \* m.out\_channels  
 m.weight.data.normal\_(0, sqrt(2. / n))  
  
 def make\_layer(self, block, num\_of\_layer):  
 layers = []  
 for \_ in range(num\_of\_layer):  
 layers.append(block())  
 # Sequential一个有序的容器，神经网络模块将按照在传入构造器的顺序依次被添加到计算图中执行  
 return nn.Sequential(\*layers)  
  
 def forward(self, x):  
 residual = x  
 out = self.relu(self.input(x))  
 out = self.residual\_layer(out)  
 out = self.output(out)  
 out = torch.add(out, residual)  
 return out

（2）核心训练代码：

def train(training\_data\_loader, optimizer, model, criterion, epoch):  
 lr = adjust\_learning\_rate(optimizer, epoch-1)  
 # param\_groups的各项参数：[{'params','lr', 'momentum', 'dampening', 'weight\_decay', 'nesterov'},{……}]  
 for param\_group in optimizer.param\_groups:  
 param\_group["lr"] = lr  
 print("Epoch = {}, lr = {}".format(epoch, optimizer.param\_groups[0]["lr"]))  
 # 接下来是训练部分 model.train()保证Batch Normalization层用每一批数据的均值和方差，对于Droupout:model.train()是随机取一部分网络连接来训练更新参数  
 # model.eval()是保证Batch Normalization用全部训练数据的均值和方差，对于Droupout:model.eval()是利用到了所有网络连接  
 model.train()  
 # 遍历training\_data\_loader，从下标1开始  
 for iteration, batch in enumerate(training\_data\_loader, 1):  
 # running\_loss = 0.0  
 # Variable是一种可以不断变化的变量，符合反向传播，参数更新的属性  
 input, target = Variable(batch[0]), Variable(batch[1], requires\_grad=False)  
  
 if opt.cuda:  
 input = input.cuda()  
 target = target.cuda()  
  
 loss = criterion(model(input), target)  
  
 if iteration % 100 == 0:  
 output = "===> Epoch[{}]({}/{}): Loss: {:.10f}".format(epoch, iteration, len(training\_data\_loader), loss.item())  
 with open("loss.txt", "a+") as f:  
 f.write(output + '\n')  
 f.close  
  
 optimizer.zero\_grad()  
 loss.backward()  
 # 梯度裁剪  
 nn.utils.clip\_grad\_norm\_(model.parameters(), opt.clip)  
 optimizer.step()  
  
 if iteration%100 == 0:  
 print("===> Epoch[{}]({}/{}): Loss: {:.10f}".format(epoch, iteration, len(training\_data\_loader), loss.item()))  
  
 # running\_loss = running\_loss + loss.item()  
 # print(running\_loss)

1. 使用PSNR、SSIM进行测试

def SSIM(pred, gt, shave\_border=0, data\_range=None):  
 height, width = pred.shape[:2]  
 pred = pred[shave\_border:height - shave\_border, shave\_border:width - shave\_border]  
 gt = gt[shave\_border:height - shave\_border, shave\_border:width - shave\_border]  
 ssim = compare\_ssim(pred, gt, multichannel=True, data\_range=data\_range)  
 return ssim  
  
  
# 定义一个函数，用于计算峰值信噪比  
def PSNR(pred, gt, shave\_border=0):  
 # 计算图像的高度和宽度  
 height, width = pred.shape[:2]  
 # 对预测图像和原始图像进行裁剪，去除边界  
 pred = pred[shave\_border:height - shave\_border, shave\_border:width - shave\_border]  
 gt = gt[shave\_border:height - shave\_border, shave\_border:width - shave\_border]  
 # 计算预测图像与原始图像的差值  
 imdff = pred - gt  
 # 计算均方根误差  
 rmse = math.sqrt(np.mean(imdff \*\* 2))  
 # 如果均方根误差为0，则返回一个较大的值（100），避免除以0的错误  
 if rmse == 0:  
 return 100  
 # 计算峰值信噪比并返回  
 return 20 \* math.log10(255.0 / rmse)

训练结束之后，对数据进行测试：

(pytorch) lyf@MSN02:~/image-stitch/pytorch-vdsr-recurrence-main$ python eval.py --cuda --dataset Set5

=> use gpu id: '0'

Processing Set5\_mat/head\_GT\_x2.mat

Processing Set5\_mat/woman\_GT\_x2.mat

Processing Set5\_mat/bird\_GT\_x2.mat

Processing Set5\_mat/baby\_GT\_x2.mat

Processing Set5\_mat/butterfly\_GT\_x2.mat

Scale= 2

Dataset= Set5

PSNR\_predicted= 37.316248550487686

PSNR\_bicubic= 33.6903938129254

SSIM\_predicted= 0.8943958101480873

It takes average 0.05012679100036621s for processing

Processing Set5\_mat/bird\_GT\_x3.mat

Processing Set5\_mat/baby\_GT\_x3.mat

Processing Set5\_mat/head\_GT\_x3.mat

Processing Set5\_mat/butterfly\_GT\_x3.mat

Processing Set5\_mat/woman\_GT\_x3.mat

Scale= 3

Dataset= Set5

PSNR\_predicted= 33.52159807935353

PSNR\_bicubic= 30.407692343235453

SSIM\_predicted= 0.809177665859762

It takes average 0.0008878231048583985s for processing

Processing Set5\_mat/woman\_GT\_x4.mat

Processing Set5\_mat/baby\_GT\_x4.mat

Processing Set5\_mat/butterfly\_GT\_x4.mat

Processing Set5\_mat/head\_GT\_x4.mat

Processing Set5\_mat/bird\_GT\_x4.mat

Scale= 4

Dataset= Set5

PSNR\_predicted= 31.216452946969838

PSNR\_bicubic= 28.41454827257395

SSIM\_predicted= 0.7257850778408536

It takes average 0.0009095668792724609s for processing

结果展示：

|  |
| --- |
| woman1 |
| h1 |

1. **总结**

可以看出使用VDSR生成的图片较LR的图片效果要好许多，与origin相比，肉眼也很难分辨出区别。

优点：

深度网络：VDSR有20个权重层，这比SRCNN（只有3层）深得多。更深的网络可以捕获更复杂的特征，从而提高超分辨率的效果。

残差学习：VDSR采用了残差学习的方法，只学习图像的高频信息，学习量小，有利于网络更好的运作。这种方法可以加速网络的收敛，并且可以使用更高的学习率。

多尺度训练：VDSR可以使用一个模型训练不同的尺度放大图片，解决不同放大尺度的问题。

缺点：

训练时间：虽然VDSR的训练时间比SRCNN短，但是对于大规模的数据集，训练时间仍然可能会很长。

复杂度：VDSR是一个深度网络，其复杂度较高，这可能会导致过拟合的问题。

插值方法：VDSR在输入网络之前需要对低分辨率图像进行插值得到ILR图像，这可能会影响最终的超分辨率效果。

改进和优化：

自然图像质量评估器（NIQE）：一项研究提出了一种新的方法来提高VDSR在单图像超分辨率（SISR）中的性能。这项研究通过使用自然图像质量评估器（NIQE）来改进VDSR网络结构的输出。新的系数被提出来改进从VDSR方法和原始低分辨率图像得到的残差图像的性能。NIQE参数被定义为这些提出的系数的优化参数，并且在每个SISR应用中都希望将其最小化。

优化网络结构：另一项研究开发了一种增强的深度超分辨率网络（EDSR），其性能超过了当前最先进的SR方法。他们通过去除传统残差网络中不必要的模块来优化网络，从而显著提高了模型的性能。他们还提出了一种新的多尺度深度超分辨率系统（MDSR）和训练方法，可以在一个模型中重建不同放大因子的高分辨率图像。

使用更深的网络：VDSR是一个深度网络，但是我们可以尝试使用更深的网络来进一步提高性能。更深的网络可能会捕获更复杂的特征，从而提高超分辨率的效果。

引入注意力机制：注意力机制已经在许多深度学习任务中取得了成功，包括图像分类、语义分割和物体检测等。在VDSR中引入注意力机制可能会帮助模型更好地关注重要的特征，从而提高超分辨率的效果。

使用更复杂的损失函数：VDSR通常使用均方误差（MSE）作为损失函数。然而，MSE并不能完全反映人眼对图像质量的感知。因此，可以考虑使用更复杂的损失函数，如结构相似性（SSIM）或感知损失。