机器学习与数据挖掘 HOMEWORK3

21307035 邓栩瀛

> Multi-level Wavelet-CNN for Image Restoration
>
> 论文链接：https://arxiv.org/pdf/1805.07071.pdf
>
> Source Code：https://github.com/lpj-github-io/MWCNNv2

1、论文研究问题的背景和动机

该论文的研究问题是图像恢复中感受野大小和效率之间的权衡。在低级视觉任务中，感受野的大小对于算法的性能至关重要。普通的卷积神经网络（CNN）通过增加感受野的大小来提升性能，但这样会增加计算成本，采用扩张卷积（dilated filtering）可以解决这个问题。但扩张卷积会导致网格效应，并且所得到的感受野只是对输入图像进行稀疏采样，带有棋盘格样式。

该论文提出一种多级小波CNN（MWCNN）模型，能够在感受野大小和计算效率之间取得更好的平衡。通过修改U-Net架构，引入小波变换(haar)来减小收缩子网络中特征图的大小，同时，进一步使用另一个卷积层来减少特征图的通道数。在扩展的子网络中，然后部署逆小波变换来重建高分辨率特征图。

MWCNN 也可以解释为扩张过滤和二次采样的泛化，并且可以应用于许多图像恢复任务，实验结果清楚地显示了MWCNN在图像去噪、单图像超分辨率和JPEG图像伪影去除方面的有效性。

2、当前解决该问题的主要方法

目前，对于图像恢复问题，已经提出了许多方法，包括基于先验模型和判别学习的方法。最近，卷积神经网络（CNNs）在图像恢复任务中也得到了广泛研究，并在几个代表性的任务中取得了最先进的性能，如单图像超分辨率（SISR）、图像去噪、图像去模糊和压缩成像。

对于图像恢复任务，CNN实际上表示从退化观察到潜在清晰图像的映射。为了使输入和输出图像具有相同的大小，一种常见的策略是使用全卷积网络（FCN），即去除池化层。一般来说，通过增加网络深度或使用更大尺寸的滤波器，可以增加感受野的大小，从而提升恢复性能。然而，对于没有池化的FCN，感受野的大小可以通过增加网络深度或使用更大尺寸的滤波器来扩大，但这无疑会增加计算成本。为了解决这个问题，一种方法是使用扩张卷积（dilated filtering）来扩大感受野而不增加计算负担。然而，扩张卷积存在网格效应的问题，其中感受野只考虑了输入图像的稀疏采样，具有棋盘格样式。因此，在扩大感受野的同时，需要小心避免增加计算负担和潜在的性能损失。

除了扩张卷积，该领域还涉及其他方法来解决感受野大小和计算效率之间的平衡。例如，采用金字塔结构、递归架构、生成对抗网络（GANs）等方法。

在该论文中提出了一种新的方法，即多级小波CNN（MWCNN），以在图像恢复任务中更好地平衡感受野大小和计算效率。

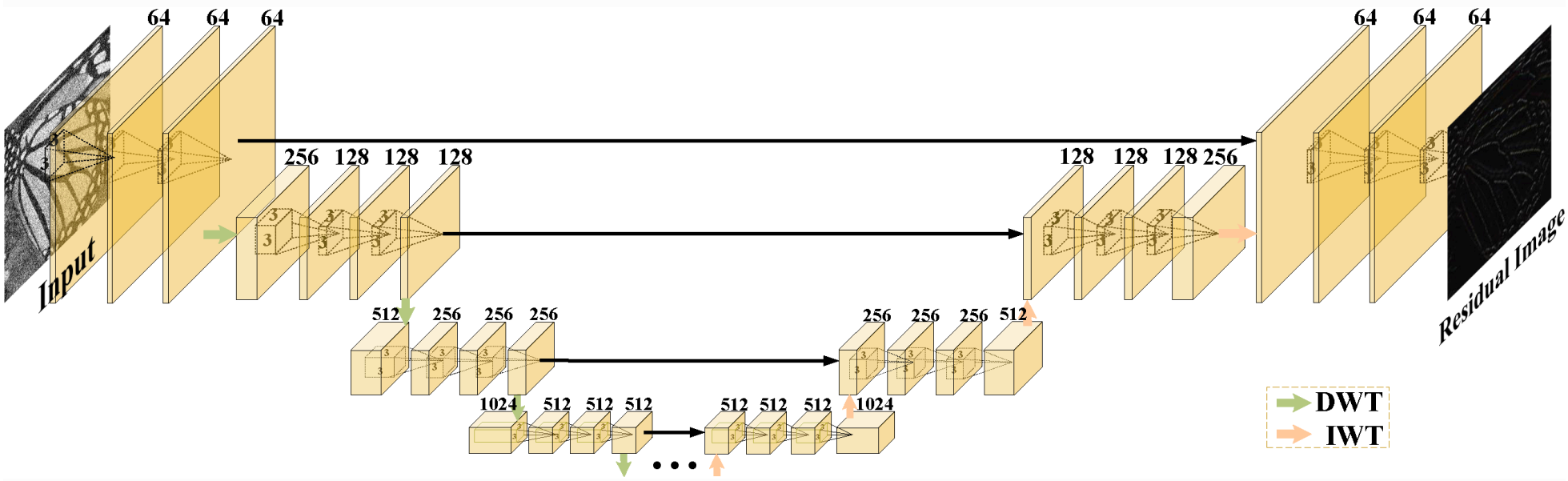
> - 多级小波包变换（WPT）

- 由多级 WPT 驱动的 MWCNN及其网络架构
- MWCNN 与扩张过滤和二次采样的联系

3、实现的模型、 算法或方法

网络结构

- 在编码器和解码器中加入residual block
- 在编码器中采用离散小波变换（DWT）作为下采样算子，在解码器中采用逆小波变换（IWT）作为上采样算子。
- 在 DWT 和 IWT 之后，分别使用 3 X 3 卷积来压缩和扩展特征图。
- 对于每个级别，进一步使用两个residual block来增强特征表示和重构



Part 1

Source Code：https://github.com/lpj-github-io/MWCNNv2/blob/master/MWCNN_code/model/common.py

代码实现了二维离散小波变换（DWT）和逆向二维离散小波变换（IWT），使用了haar小波变换来对图像进行变换和逆变换。

```
def dwt_init(x):
    x01 = x[:, :, 0::2, :] / 2
    x02 = x[:, :, 1::2, :] / 2
    x1 = x01[:, :, :, 0::2]
    x2 = x02[:, :, :, 0::2]
    x3 = x01[:, :, :, 1::2]
    x4 = x02[:, :, :, 1::2]
    x_LL = x1 + x2 + x3 + x4
    x_HL = -x1 - x2 + x3 + x4
    x_LH = -x1 + x2 - x3 + x4
    x_HH = x1 - x2 - x3 + x4
    return torch.cat((x_LL, x_HL, x_LH, x_HH), 1)
```

`dwt_init(x)` 实现二维离散小波变换：给定输入张量x，将其分解为四个子频带（LL、HL、LH、HH），其中，LL表示低频部分，HL、LH、HH表示高频部分。函数通过对输入张量进行切片和组合操作，将原始图像分解为不同频带的子图像。

其中

```
    x01 = x[:, :, 0::2, :] / 2
    x02 = x[:, :, 1::2, :] / 2
```

将矩阵分为偶数行和奇数行，并将所有值都除以2，后续进行求和和求差即可

```
    x1 = x01[:, :, :, 0::2]
    x2 = x02[:, :, :, 0::2]
    x3 = x01[:, :, :, 1::2]
    x4 = x02[:, :, :, 1::2]
```

分为偶数列和奇数列，假设矩阵为$2n \times 2n$大小，那么就将该矩阵分成了4个$n \times n$大小的x_1、x_2、x_3和x_4

假设矩阵为4×4，则有

x_1	x_2	x_1	x_2
x_3	x_4	x_3	x_4
x_1	x_2	x_1	x_2
x_3	x_4	x_3	x_4

```
    x_LL = x1 + x2 + x3 + x4
    x_HL = -x1 - x2 + x3 + x4
    x_LH = -x1 + x2 - x3 + x4
    x_HH = x1 - x2 - x3 + x4
```

进行行、列的和、差变换

1、x_LL：对行求和$x_1 + x_3$和$x_2 + x_4$，对列求和$(x_1 + x_3) + (x_2 + x_4) = x_1 + x_2 + x_3 + x_4$

2、x_HL：对行求差$x_3 - x_1$和$x_4 - x_2$，对列求和$(x_3 - x_1) + (x_4 - x_2) = -x_1 - x_2 + x_3 + x_4$

3、x_LH：对行求和$x_1 + x_3$和$x_2 + x_4$，对列求差$(x_2 + x_4) - (x_1 + x_3) = -x_1 + x_2 - x_3 + x_4$

4、x_HH：对行求差$x_3 - x_1$和$x_4 - x_2$，对列求差$(x_4 - x_2) - (x_3 - x_1) = x_1 - x_2 - x_3 + x_4$

```
def iwt_init(x):
    r = 2
    in_batch, in_channel, in_height, in_width = x.size()
    # print([in_batch, in_channel, in_height, in_width])
    out_batch, out_channel, out_height, out_width = in_batch, int(in_channel /
(r ** 2)), r * in_height, r * in_width
    x1 = x[:, 0:out_channel, :, :] / 2
    x2 = x[:, out_channel:out_channel * 2, :, :] / 2
    x3 = x[:, out_channel * 2:out_channel * 3, :, :] / 2
    x4 = x[:, out_channel * 3:out_channel * 4, :, :] / 2
    h = torch.zeros([out_batch, out_channel, out_height, out_width]).float()

    h[:, :, 0::2, 0::2] = x1 - x2 - x3 + x4
    h[:, :, 1::2, 0::2] = x1 - x2 + x3 - x4
    h[:, :, 0::2, 1::2] = x1 + x2 - x3 - x4
    h[:, :, 1::2, 1::2] = x1 + x2 + x3 + x4
```

```
        return h
```

`iwt_init` 实现逆向二维离散小波变换：给定输入张量x，根据haar小波的逆变换公式将其重构为原始图像。函数中的变量r表示放大倍数，通过对输入张量进行切片和组合操作，将子频带图像重构为原始图像。

其中

```
        x1 = x[:, 0:out_channel, :, :] / 2
        x2 = x[:, out_channel:out_channel * 2, :, :] / 2
        x3 = x[:, out_channel * 2:out_channel * 3, :, :] / 2
        x4 = x[:, out_channel * 3:out_channel * 4, :, :] / 2
```

图像重构就是从这些值中恢复dwt中的x1,x2,x3,x4,分别放到h对应的位置变为原来的矩阵，如x1对应h(:, :, 0::2, 0::2)

重构的方法

$$
\begin{aligned}
x1 &= x\_LL/2 \\
x2 &= x\_HL/2 \\
x3 &= x\_LH/2 \\
x4 &= x\_HH/2
\end{aligned}
\tag{2}
$$

```
# 二维离散小波
class DWT(nn.Module):
    def __init__(self):
        super(DWT, self).__init__()
        self.requires_grad = False # 信号处理，非卷积运算，不需要进行梯度求导

    def forward(self, x):
        return dwt_init(x)

# 逆向二维离散小波
class IWT(nn.Module):
    def __init__(self):
        super(IWT, self).__init__()
        self.requires_grad = False # 信号处理，非卷积运算，不需要进行梯度求导

    def forward(self, x):
        return iwt_init(x)
```

`DWT` 和 `IWT` 是PyTorch的模块，分别用于封装二维离散小波变换和逆向二维离散小波变换，定义了前向传播方法，分别调用了 `dwt_init` 和 `iwt_init` 函数。

part 2

DWT/IWT 和卷积单元的核心代码

source code:https://github.com/lpj-github-io/MWCNNv2/blob/master/MWCNN_code/model/mwcnn.py

头部网络层

```
m_head = [common.BBlock(conv, nColor, n_feats, kernel_size, act=act)]
```

downsample第一层

```
d_l0 = []
d_l0.append(common.DBlock_com1(conv, n_feats, n_feats, kernel_size, act=act,
bn=False))
```

downsample第二层

```
d_l1 = [common.BBlock(conv, n_feats * 4, n_feats * 2, kernel_size, act=act,
bn=False)]
d_l1.append(common.DBlock_com1(conv, n_feats * 2, n_feats * 2, kernel_size,
act=act, bn=False))
```

downsample第三层

```
d_l2 = []
d_l2.append(common.BBlock(conv, n_feats * 8, n_feats * 4, kernel_size, act=act,
bn=False))
d_l2.append(common.DBlock_com1(conv, n_feats * 4, n_feats * 4, kernel_size,
act=act, bn=False))
```

downsample第四层，并与upsample进行连接，也是upsample的第四层

```
pro_l3 = []
pro_l3.append(common.BBlock(conv, n_feats * 16, n_feats * 8, kernel_size,
act=act, bn=False))
pro_l3.append(common.DBlock_com(conv, n_feats * 8, n_feats * 8, kernel_size,
act=act, bn=False))
pro_l3.append(common.DBlock_inv(conv, n_feats * 8, n_feats * 8, kernel_size,
act=act, bn=False))
pro_l3.append(common.BBlock(conv, n_feats * 8, n_feats * 16, kernel_size,
act=act, bn=False))
```

upsample第三层

```
i_l2 = [common.DBlock_inv1(conv, n_feats * 4, n_feats * 4, kernel_size, act=act,
bn=False)]
i_l2.append(common.BBlock(conv, n_feats * 4, n_feats * 8, kernel_size, act=act,
bn=False))
```

upsample第二层

```
i_l1 = [common.DBlock_inv1(conv, n_feats * 2, n_feats * 2, kernel_size, act=act,
bn=False)]
i_l1.append(common.BBlock(conv, n_feats * 2, n_feats * 4, kernel_size, act=act,
bn=False))
```

upsample第一层

```
i_l0 = [common.DBlock_inv1(conv, n_feats, n_feats, kernel_size, act=act,
bn=False)]
```

尾部网络层

```
m_tail = [conv(n_feats, nColor, kernel_size)]
```

将定义的网络层组合成Sequential模块

```
self.head = nn.Sequential(*m_head)
self.d_l2 = nn.Sequential(*d_l2)
self.d_l1 = nn.Sequential(*d_l1)
self.d_l0 = nn.Sequential(*d_l0)
self.pro_l3 = nn.Sequential(*pro_l3)
self.i_l2 = nn.Sequential(*i_l2)
self.i_l1 = nn.Sequential(*i_l1)
self.i_l0 = nn.Sequential(*i_l0)
self.tail = nn.Sequential(*m_tail)
```

前向传播过程

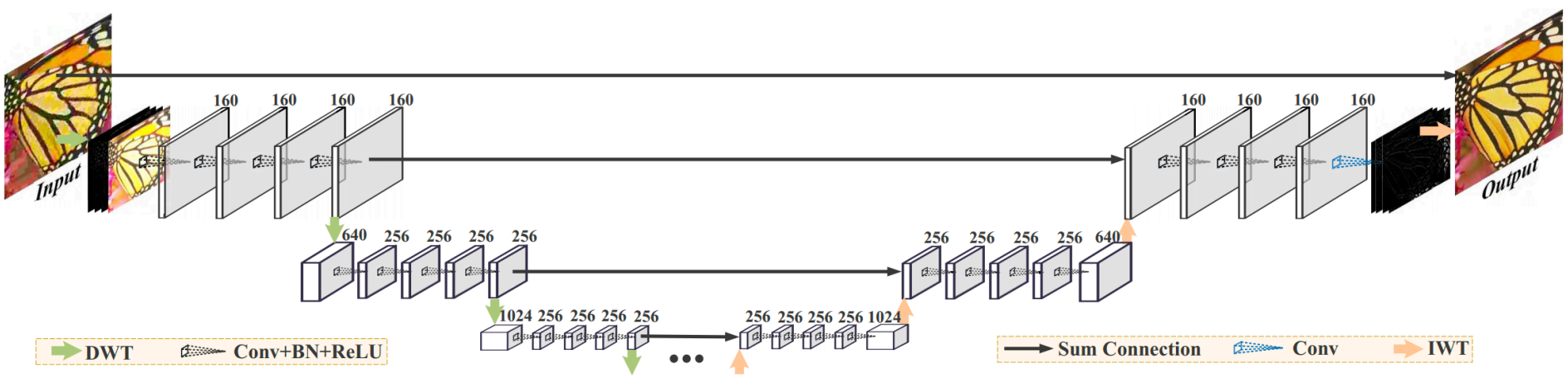
```
def forward(self, x):
    x0 = self.d_l0(self.head(x))
    x1 = self.d_l1(self.DWT(x0))
    x2 = self.d_l2(self.DWT(x1))
    x_ = self.IWT(self.pro_l3(self.DWT(x2))) + x2
    x_ = self.IWT(self.i_l2(x_)) + x1
    x_ = self.IWT(self.i_l1(x_)) + x0
    x = self.tail(self.i_l0(x_)) + x
    return x
```

- 将输入x经过`self.head`层进行卷积处理，得到$x0$
- 将$x0$经过离散小波变换（DWT）`self.DWT`得到$x1$，然后将$x1$经过`self.d_l1`层进行卷积处理，得到$x2$
- 将$x2$经过`self.DWT`进行离散小波变换，然后经过一系列卷积层`self.pro_l3`得到$x_$
- 将$x_$经过逆离散小波变换（IWT）`self.IWT`得到$x_$的逆变换结果，并与$x2$相加
- 将$x_$经过逆卷积层`self.i_l2`进行卷积处理，得到$x_$的逆变换结果，并与$x1$相加
- 将$x_$经过逆卷积层`self.i_l1`进行卷积处理，得到$x_$的逆变换结果，并与$x0$相加
- 将$x_$经过逆卷积层`self.i_l0`进行卷积处理，得到$x_$的逆变换结果，并与输入x相加
- 将结果x经过尾部卷积层`self.tail`进行卷积处理，得到最终的输出x

4、实验结果及分析

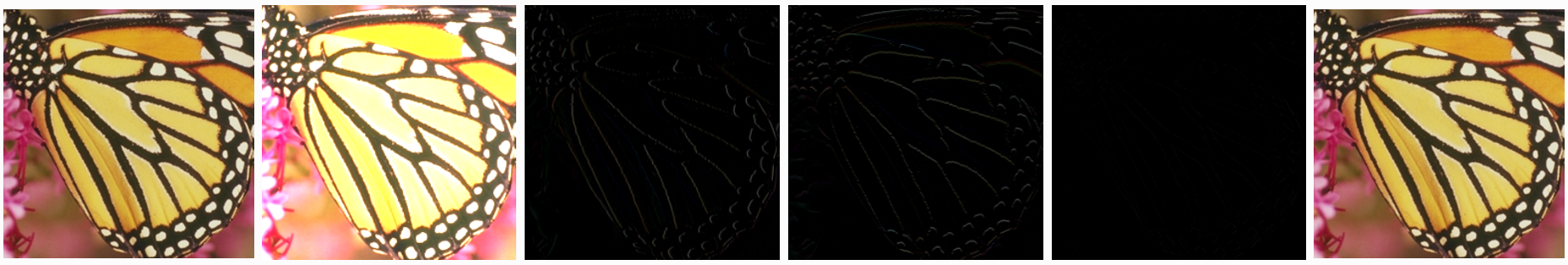
多级小波-CNN 架构

由两个部分组成：收缩子网络和扩展子网络



- CNN块为无池化的4层FCN
- 每次完整的卷积过程包括：卷积核为3*3的卷积层、BatchNormalization层以及ReLU层
- 扩张子网络中最后一个卷积块的最后一个卷积过程只有卷积操作，丢弃了BatchNormalization层以及ReLU层
- 整个网络共涉及24个网络层，图中可看到最终MWCNN借助三级小波变换/逆变换实现下采样、上采样操作
- MWCNN默认采用哈尔小波进行小波变换/逆变换

上述过程复现如下：



图像降噪(Image Denoising)

对源码进行复现，并对图像降噪(Image Denoising)部分进行了测试，使用Set12数据集

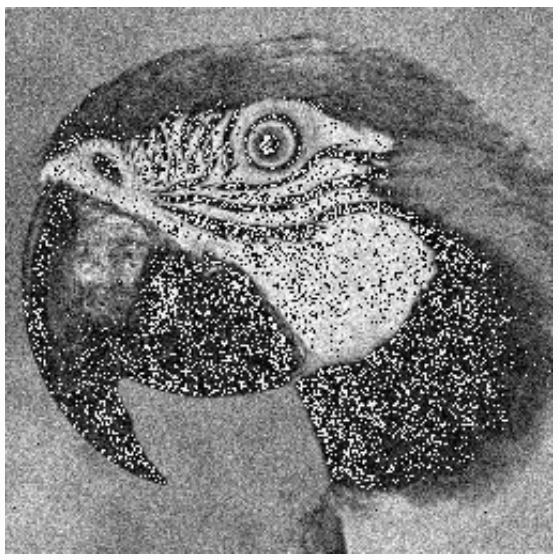
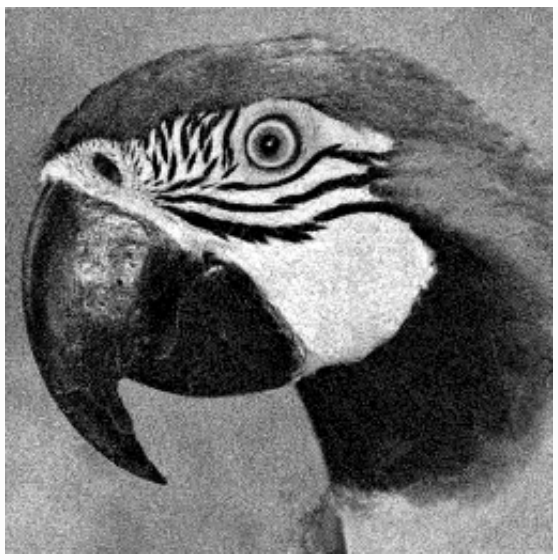
运行结果

```
Evaluation:
100%|                                                                                  | 12/12 [00:04<00:00,  2.76it/s][Set12 x25]     PSNR: 23.676 (Best: 2
3.676 @epoch 1)
Total time: 4.34s
```

部分结果展示

High-Resolution　　Low-Resolution　　Super-Resolution

High-Resolution　　Low-Resolution　　Super-Resolution

High-Resolution　　Low-Resolution　　Super-Resolution

结构如下

1、head

```
(head): Sequential(
  (0): BBlock(
    (body): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
```

2、d_l2

```
(d_l2): Sequential(
  (0): BBlock(
    (body): Sequential(
      (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
  (1): DBlock_com1(
    (body): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
```

3、d_l1

```
(d_l1): Sequential(
  (0): BBlock(
    (body): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
  (1): DBlock_com1(
    (body): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
  )
)
```

4、d_l0

```
(d_10): Sequential(
  (0): DBlock_com1(
    (body): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
  )
)
```

5、pro_l3

```
(pro_l3): Sequential(
  (0): BBlock(
    (body): Sequential(
      (0): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
  (1): DBlock_com(
    (body): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(3, 3), dilation=(3, 3))
      (3): ReLU(inplace)
    )
  )
  (2): DBlock_inv(
    (body): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(3, 3), dilation=(3, 3))
      (1): ReLU(inplace)
      (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (3): ReLU(inplace)
    )
  )
  (3): BBlock(
    (body): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
)
```

6、i_l2

```
(i_l2): Sequential(
  (0): DBlock_inv1(
    (body): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
  )
  (1): BBlock(
    (body): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
)
```

7、i_l1

```
(i_l1): Sequential(
  (0): DBlock_inv1(
    (body): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
  )
  (1): BBlock(
    (body): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace)
    )
  )
)
```

8、i_l0

```
(i_10): Sequential(
  (0): DBlock_inv1(
    (body): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2))
      (1): ReLU(inplace)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace)
    )
  )
)
```

9、tail

```
(tail): Sequential(
  (0): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

总结分析：

1、CNN块使用无池化(Pooling)的4层FCN

2、每次卷积过程包括：卷积核为3*3的卷积层、BatchNormalization层以及ReLU层

3、扩张子网络中最后一个卷积块的最后一个卷积过程只有卷积操作，而没有BatchNormalization层以及ReLU层

4、采用哈尔小波进行小波变换/逆变换

5、结论

文章提出了一种用于图像恢复的多级小波-CNN（MWCNN）架构，由收缩子网络和扩展子网络组成。收缩子网由多级 DWT 和 CNN 块组成，而扩展子网由多级 IWT 和 CNN 块组成。由于 DWT 的可逆性、频率和位置特性，MWCNN 可以安全地执行子采样而不会丢失信息，并且可以有效地从退化的观察中恢复详细的纹理和锐利的结构。因此，MWCNN 可以扩大感受野，在效率和性能之间取得更好的平衡。大量的实验证明了 MWCNN 在三个恢复任务上的有效性和效率，即图像去噪、SISR 和 JPEG 压缩伪影去除。

MWCNN与U-Net的区别：

- 对于上采样和下采样，U-Net中采用最大池化和上卷积（up-convolution），而在MWCNN中采用DWT和IWT。
- 对于MWCNN，下采样会导致feature map的数量增加，而在U-Net中下采样不会印象特征图的channel数目，而是采用随后的卷积层来增加feature map 的channel
- 在MWCNN中，elementwise summation被用于结合来自于收缩网络和扩展网络的特征图，而在传统的U-Net中采用级联

MWCNN对U-Net做的改进：

- U-Net中的下采样、上采样操作分别被DWT和IWT代替
- MWCNN中除了第一个卷积块对特征图通道进行了增加，其他的卷积块都是减少特征图通道，以此学习压缩表示
- 采用元素级加法连接同层的压缩子网与扩张子网的特征图