

NLP 课设

大作业



2022 年 12 月 18 日

简 表

硬件环境（CPU/GPU）：
SwinIR:P40； IDN:1080Ti； RCAN: 1080Ti； VDSR:1080Ti;SRCNN:1080Ti；

操作系统：
在 Windows 下测试，采用 linux 训练。

采用的深度学习框架、工具、语言：
深度学习的框架都采用：PyTorch。语言采用 Python，IDE 采用 jupyter notebook

任务描述/问题定义：（限 200 字）
图像超分辨率重建任务：输入一个低分辨率图像，输出一个高分辨率图像。

数据集及来源（相关链接）：均采用 SR 图像的公开数据集。
DIV2K: <https://cv.snu.ac.kr/research/EDSR/DIV2K.tar>，Set5 + Set14 + BSD100 + Urban100 + Manga109: <https://drive.google.com/drive/folders/1B3DJGQKB6eNdwuQlhdsKA64qUuVKLZ9u>

采用的深度学习模型：
SwinIR;IDN;RCAN;VDSR;SRCNN

模型提出的年份及发表的会议/期刊：
SwinIR:ICCV-W-2021；
IDN： CVPR-2018; RCAN:ECCV-2018
VDSR:CVPR-2016； SRCNN:ECCV-2014；

最终设置的超参数（如 Learning rate, Batch size 等）：
SwinIR:数据集为 DIV2K 前 800 张，图片块大小 64,batch-size 40,优化函数 Adam,损失函数 L1，学习率 2×10^{-4} 。**IDN:** 数据集为 291 图，优化函数为 Adam，mini-batch 大小为 64。第一阶段：图片块大小为 29，损失函数为 L1，学习率为 10^{-4} 。第二阶段：图片块大小为 39，损失函数为 L2，学习率为 10^{-3} 。**RCAN:** 训练集为 DIV2K 前 800 图，损失函数 L1,优化函数 Adam,batch-size 16,Epoch100 次，学习率初试 10^{-4} ，每 200Epoch 下降一倍。**VDSR:**数据集 291 图，图片块 41,batch-size 64，损失函数 MSE,优化函数 SGD,初试学习率 0.1,每 10 个 Epoch 下降 10 倍。**SRCNN:**数据集 91 图，优化函数 Adam, batch-size 16，图片块大小 33，前两层学习率 10^{-4} ，第三层学习率 10^{-5} ,损失函数 MSE。

模型的效果（如准确率等与任务相关的评价指标）：

表 2-1 PSNR/SSIM 性能指标 [Ⓔ]							
↵	Scale [↵]	Bicubic [↵]	SRCNN [↵]	VDSR [↵]	RCAN [↵]	IDN [↵]	SwinIR [↵]
Set5 [↵]	×2 [↵]	33.67/0.9303 [↵]	36.30/0.9535 [↵]	37.18/0.9583 [↵]	38.27/0.9614 [↵]	37.77/0.9599 [↵]	38.41/0.9620 [↵]
Set14 [↵]	×2 [↵]	30.32/0.8699 [↵]	32.24/0.9054 [↵]	32.88/0.9127 [↵]	34.01/0.9205 [↵]	33.38/0.9163 [↵]	34.46/0.9250 [↵]
B100 [↵]	×2 [↵]	29.56/0.8434 [↵]	31.15/0.8860 [↵]	31.77/0.8954 [↵]	32.39/0.9023 [↵]	32.05/0.8986 [↵]	32.53/0.9041 [↵]
Urban100 [↵]	×2 [↵]	26.56/0.8376 [↵]	28.94/0.8957 [↵]	30.43/0.9178 [↵]	32.97/0.9444 [↵]	31.32/0.9279 [↵]	33.81/0.9427 [↵]
Manga109 [↵]	×2 [↵]	30.82/0.9349 [↵]	34.98/0.9661 [↵]	36.43/0.9731 [↵]	39.36/0.9786 [↵]	37.77/0.9599 [↵]	39.92/0.9797 [↵]

1. 深度学习框架、工具等的安装及配置过程

本机测试的时候在 Windows 下使用 Anaconda 建立虚拟环境，训练的时候使用云计算平台机器预装的 pytorch 并安装相关的支持库。

1. 下载并安装 Anaconda
2. 在 Anaconda 中设置虚拟环境并安装 pytorch
3. 使用 `conda install XXX` 安装其他的支持库, 比如 `hdf5, opencv-python` 等。

使用的 IDE 是 Anaconda 中就会预装的 jupyter notebook。

2. 采用的深度学习模型的详细描述

对于模型的评估，我采用峰值信噪比 (Peak signal to noise ratio, PSNR) 和结构相似度 (Structural similarity, SSIM) 这两个指标来评估，他们是广泛用来衡量图像质量的指标。这两个指标的公式为：

$$PSNR = 10 \times \lg \frac{MN}{\|f - \hat{f}\|^2}$$

$$SSIM = \frac{(2\mu_f \mu_{\hat{f}} + C_1)(\sigma_{f\hat{f}} + C_2)}{(\mu_f^2 + \mu_{\hat{f}}^2 + C_1)(\sigma_f^2 + \sigma_{\hat{f}}^2 + C_2)}$$

其中 M, N 为图片的真实尺寸, f 是高清图像, \hat{f} 为重建的超分辨率图像, μ_f 和 σ_f 分别为高清图像的平均灰度值与方差, 而 $\mu_{\hat{f}}$ 与 $\sigma_{\hat{f}}$ 分别为重建图像的灰度值与方差。 $\sigma_{f\hat{f}}$ 为原图与重建图像的协方差, C_1, C_2 为常数。

表 2-1 PSNR/SSIM 性能指标

	Scale	Bicubic	SRCNN	VDSR	RCAN	IDN	SwinIR
Set5	×2	33.67/0.9303	36.30/0.9535	37.18/0.9583	38.27/0.9614	37.77/0.9599	38.41/0.9620
Set14	×2	30.32/0.8699	32.24/0.9054	32.88/0.9127	34.01/0.9205	33.38/0.9163	34.46/0.9250
B100	×2	29.56/0.8434	31.15/0.8860	31.77/0.8954	32.39/0.9023	32.05/0.8986	32.53/0.9041
Urban100	×2	26.56/0.8376	28.94/0.8957	30.43/0.9178	32.97/0.9444	31.32/0.9279	33.81/0.9427
Manga109	×2	30.82/0.9349	34.98/0.9661	36.43/0.9731	39.36/0.9786	37.77/0.9599	39.92/0.9797

注：由于 SwinIR 训练时间单卡 V100 需要 120 天以上，所以采用作者论文中

的数据。我仅测试了 1000Epoch 保证代码正确可运行。其余四种模型中数据为我经过复现测试的真实数据。

2.1 SRCNN 相关细节

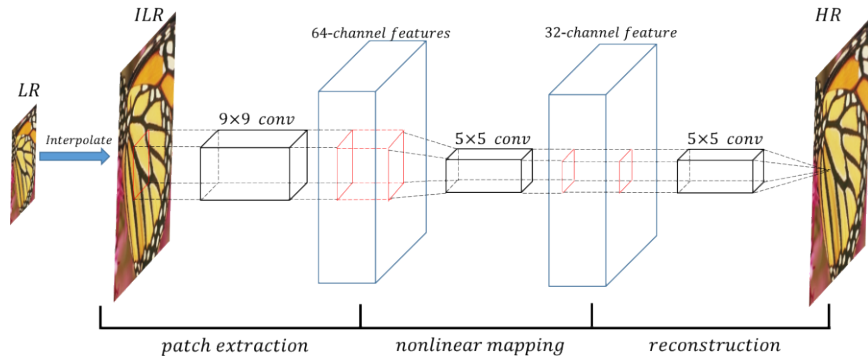


图 2.1-1 SRCNN 网络模型图

SRCNN 模型结构由三层卷积网络构成，第一层是特征提取层，作者现将低分辨率图像通过双三次插值放大到指定倍数，然后使用特征提取层提取低分辨率图像低频信息，我将第一层操作表示为 $F_1(Y) = \max(0, W_1 * Y + B_1)$ 。这里 W 为权重参数， B 为偏置参数。 W 的维度与卷积核大小 f_1 有关为 $c \times f_1 \times f_1 \times n_1$ ，其中卷积核为 $f_1 \times f_1$ ($f_1=9$)， c 为输入通道数， n_1 为滤波器个数 64，因为人眼对亮度变换改变感受明显，当仅在 y 通道上训练时，输入通道数为 1。在卷积结束后添加一个激活函数 Relu: $\max(0, x)$ ，激活函数保证了模型成为非线性，增加模型复杂度。第二层是非线性映射层，作者将上一层的滤波器个数 $n_1 = 64$ 作为特征提取层的输入，所以权重参数的维度应为 $n_1 \times f_2 \times f_2 \times n_2$ ，本层滤波器个数 $n_2=32$ 。通过对模型参数的调整，使用 1×1 大小的卷积核进行卷积运算，将提取到的低维信息映射到高维。这一层的操作表示为: $F_2(Y) = \max(0, W_2 * F_1(Y) + B_2)$ 。第三层是图像重建层，通过映射获得的高维信息进行重建，得到高分辨率的图像。这里的函数定义为: $F(Y) = W_3 * F_2(Y) + B_3$ ，这里 W 的权重参数为 $n_2 \times f_3 \times f_3 \times c$ ($f_3=5$)。

作者的整个三层卷积网络，不采用 padding 补 0。作者训练集使用 91 图和 ImageNet 数据集。将数据划分为 33×33 的图片块进行训练。使用 MSE 作为损失函数，使用 SGD 作为优化函数，前两层学习率设置为 0.0001，第三层学习率设置为 0.00001。在实现过程中，相比于作者论文中展示的流程，我的模型实现采用补 0 操作，保证输入输出图片尺寸大小不变。模型结构上，第二层卷积核采用

5×5 的卷积核，其余两层相同。训练数据采用 91 图分割成 33×33 的图片块，与作者相同，优化函数由 SGD 改为 Adam，加快训练速度。我使用的设备是 GTX 1080Ti，一个尺度训练时间大概约 8 小时。最终由训练得到的 $\times 2$ 模型对于 Set5 的 PSNR 为 36.61，对比作者原文中的 36.34，模型性能达到作者论文中水平。具体参数见表 2.1-1，其中 SRCNN 为 SRCNN 原作者在他论文中给出的数据，SRCNN-1 为我进行实现后，进行实验验证得到的数据。

表 2.1-1 作者论文 PSNR (dB) 与实现 PSNR (dB) 对照表

Set5	Scale	SRCNN	SRCNN-1	Scale	SRCNN	SRCNN-1	Scale	SRCNN	SRCNN-1
Baby	2	38.30	38.53	3	35.01	36.07	4	32.98	33.13
Bird	2	40.64	40.89	3	34.91	35.22	4	31.98	32.05
Butterfly	2	32.20	32.59	3	27.58	28.53	4	25.07	25.10
Head	2	35.64	35.77	3	33.55	35.16	4	32.19	32.45
Woman	2	34.94	35.26	3	30.92	31.37	4	28.21	28.40
Average	2	36.34	36.61	3	32.39	33.27	4	30.09	30.23

3.2 VDSR 相关细节

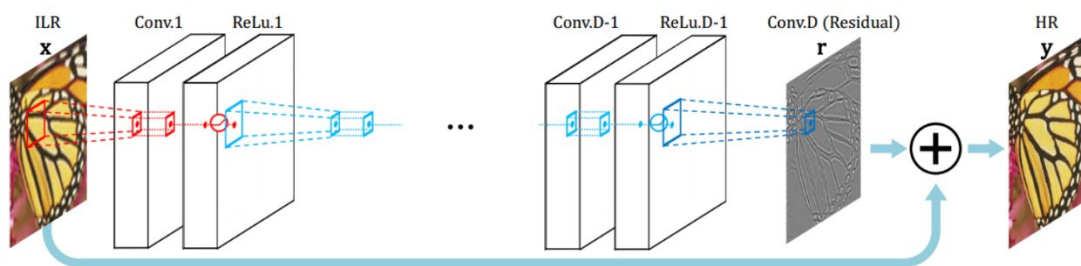


图 3.2-1 VDSR 网络模型图

作者的模型可以分为三个部分，开头的模型特征提取，中间的级联的滤波器，结尾的图像重建。除了开头的输入和结尾的输出需要符合通道数外，其他的滤波器个数都设置为 64，卷积核大小采用 3×3 的小卷积核。网络输入数据同 SRCNN 类似，需要将高分辨率 HR 图像和等大的双三次插值图像 LR 送入输入端，进行学习训练。

在训练细节上，作者采用的数据集是 291 的数据集，包含了之前 SRCNN 的 91 图。作者将图片划分为 41×41 的图片块大小，批次为 64，训练中使用 mini-batch 梯度下降反馈来优化回归。我设置动量参数为 0.9。权重正规化衰变的 (L2 惩罚因子为 0.0001)，损失函数为均方误差。作者使用数据增强，进行了翻转、旋转、缩放。训练周期采用 80Epoch（输入为 64，进行 9960 次迭代），学习率每 20Epoch 减少 10 倍，总共下降三次，到 80Epoch 停止。作者在 GPU Titan Z 上的训练时间大概是 4 个小时。

在我的实现过程中，数据处理使用 Matlab 进行数据处理，将 291 图进行分割成 41×41 的数据块，同时采用数组增强，最终数据可达到 15GB，为 $64 \times 41 \times 41 \times 9960$ 。优化函数与损失函数与作者相同，训练周期采用 50Epoch，学习率每 10Epoch 减少 10 倍。我在 GTX1080Ti 上进行训练，大概一周期耗时 20 分钟，50 个周期大概为 16 小时。进过训练，我的模型基本已经到达作者论文中性能参数，具体见表 2.2-1，VDSR 为作者在论文中给出的数据，VDSR-1 为我实现的 VDSR 模型经过实验计算后得到的数据。

表 2.2-1 论文 PSNR(dB)与生成 PSNR(dB)对照表

Set5	Bicubic	VDSR	VDSR-1
x2	33.70	37.51	37.53
x3	30.41	33.67	33.66
x4	28.41	31.26	31.35

2.3 RCAN 的细节

整个 RCAN 模型，由开始的提取特征部分，中间的 RIR 结构，结尾的图片重建由三部分组成。第一部分特征提取块只含有一个输入为图片通道数，输出为 64 的 3×3 卷积构成。第二部分为一个 RIR 结构，其中 $n=10$ ， $m=20$ 。结尾部分为图片重建的升尺度模块，由四部分组成，分别是一个输入输出为 64 的 3×3 卷积，一个输入为 64 输出为 $64 \times \text{Scale2}$ 的 3×3 卷积，一个 Pixelshuffle (scale) 以及一个输入为 64 输出为图片通道数的 3×3 卷积。

作者在模块构建上，提出了 RIR 结构 (Residual in Residual)，即残差里面内嵌残差网络，作者受到 ResNet 网络启发，进行改进所得到的一种结构。一个

RIR 结构由 n 个 RG 模块和一个长跳连接组成。在 RIR 模块中，长跳连接(LSC)将特征提取层的输出与经过 n 个 RG 模块运算的结果进行加法进行输出，而在作者的论文中 n 取 10。RIR 中的基本块 RG 由 m 个 RCAB 构成与一个短跳连接(SSC)构成。短跳连接将第一个 RCAB 的输入与经过 m 个 RCAB 运算后的结果相加进行输出，论文中 m 取 20。

RCAB 本身则由四部分组成(见图 2.3-1)，按照顺序分别为，一个输入输出 64， 3×3 的卷积，一个 Relu 激活函数，一个输入输出为 64 的 3×3 卷积，一个 CA 块，RCAB 同样应用残差进行学习。而 CA 块由一个自适应均值池化层，一个输入 64，输出 $64//16$ ， 3×3 的卷积，一个 Relu 激活函数，一个输入 $64//16$ ，输出 64 的 3×3 卷积，以及一个 Sigmoid 激活函数构成，最后将输入 x 与经过 CA 块运算的权重参数相乘作为输出结果。在整个模型结构中，针对于第 g 个 RG 模块中的第 b 个 RB 模块，给出公式： $F_{g,b} = F_{g,b-1} + R(X_{g,b}) \cdot X_{g,b}$ 。这里的函数 $R()$ 表示的是 CA 块的运算函数， $F_{g,b-1}$ 和 $F_{g,b}$ 分别为上一层的输出(这一层的输入)与这一层的输出， $X_{g,b}$ 是 F 经过卷积运算、Relu 激活函数、一个卷积运算后得到的结果，公式为： $X_{g,b} = W_{g,b}^2 \cdot U(W_{g,b}^1 F_{g,b-1})$ 。其中 $W_{g,b}^1$ 和 $W_{g,b}^2$ 为 RCAB 的两个卷积层参数。将学习的特征与经过 CA 块运算得到的权重参数相乘，再与上一层的输出相加，得到输出结果 $F_{g,b}$ 。

在训练细节上，作者采用的数据集为 DIV2K，进行翻转旋转等数据增强手段处理，批次大小设置为 16，一周周期迭代 10 次，一共训练 1000 个周期，完成 10^4 次迭代。初始学习率为 0.0001，每过 200 个周期学习率下降一倍。损失函数设置为 L1，优化函数为 Adam，无其他特殊设置，作者在 Titan X 进行训练，在 $\times 4$ 尺寸下，一个周期大概 818s，一个模型训练越 10 天。我为了验证训练模型代码的正确性，实现模型的实现，但由于时间关系，仅验证对于 RCAN 的 $\times 2$ 尺寸的实验，这足以证明模型训练代码是否正确。在我训练过程中，我通过验证得到了对于 DIV2K 第 801 到 805 张图片作为验证集的最好的模型，同时我将第 1000 个周期的模型列入性能评估中，得到了表 2.3-1 的内容。对比而言不难发现我的模型已经基本实现了论文中关于 RCAN 的性能指标，模型训练代码具有足够的正确性，但由于模型训练具有不确定的映射关系，我没有充分验证这 1000 个模型中最好的模型性能是否可以超过已经验证的第 1000 个模型，重新训练是否可以得

到更好的效果，但仍可以验证模型性能与论文中基本符合。由于我采用的机器 GTX1080Ti 训练 $\times 2$ 尺寸 1000 周期约为 11 天，我便不进行更多的重复性训练。

表 2.3-1 论文 PSNR (dB)/SSIM 与实现 PSNR (dB)/SSIM 对照表

	Scale	Mysbest RCAN	Mylast RCAN	RCAN	RCAN+
Set5	$\times 2$	38.25/0.9614	38.27/0.9614	38.27/0.9614	38.33/0.9617
Set14	$\times 2$	34.00/0.9206	34.01/0.9205	34.12/0.9216	34.23/0.9225
B100	$\times 2$	32.38/0.9023	32.39/0.9023	32.41/0.9027	32.46/0.9031
Urban100	$\times 2$	32.98/0.9446	32.97/0.9444	33.34/0.9384	33.54/0.9399
Manga109	$\times 2$	39.40/0.9786	39.36/0.9786	39.44/0.9786	39.61/0.9788

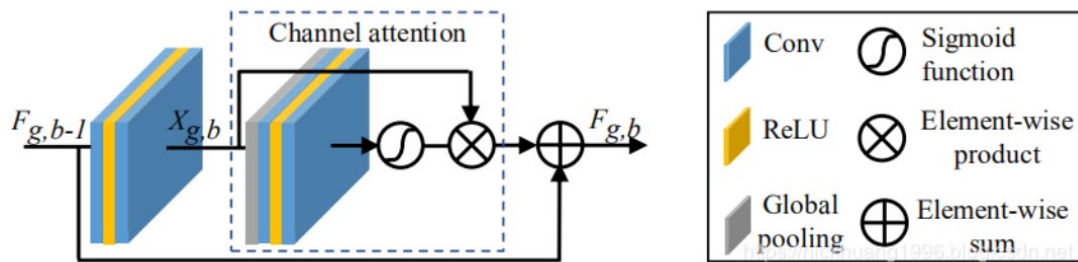


图 2.3-1 RCAB 网络模型图

2.4 IDN 的相关细节

在模型的输入输出上，IDN 模型输入为低分辨率图像，输出为放大相应尺寸后的高分辨率图像，不需要预先进行双三次插值进行处理。在模型结构上，IDN 与传统模类似，主要分为三个基本部分，即输入特征提取模块，中间模型处理模块，最后的图片重建模块。IDN 对于中间模型处理，使用堆叠的信息蒸馏模块，具体的模型结构如图 2.4-1。

在具体的模块细节上，模型的第一个模块特征提取模块主要由两个卷积层构成，每个卷积层后接一个 LReLU 激活函数。其中第一个卷积层输入为图片通道数（默认为 3），输出通道数默认为 $n_{fea}=64$ ，卷积核大小为 3，padding 补零参数为 1，LReLU 参数设置为 0.05 且所有 LReLU 参数设置为 0.05。第二个卷积层的输入通道数输出通道数默认为 64，卷积核大小为 3，采用 padding 补零参数为 1，后接 LReLU。中间采用堆叠的若干个信息蒸馏模块 (DBlock)，而每一个信息蒸馏块

由两个基础部分组成，分别是提升单元块(enhancement unit)和压缩单元块(compression unit)。提升单元块结构图见图 2.4-2，图中仅显示卷积而省略 LReLU 激活函数。由结构图可见提升单元块主要由两部分，分别是首部堆叠的三个卷积和尾部堆叠的三个卷积，这三个卷积层卷积核都为 3 且均采用 padding 补零操作参数为 1。我假设每层特征图维度为 $D_i (i = 1, \dots, 6)$ ，则首部卷积层关系有： $D_3 - D_1 = D_1 - D_2 = d$ ，相应尾部卷积层关系有： $D_6 - D_4 = D_4 - D_5 = d$ ，其中 $D_4 = D_3$ 。我由论文知，默认特征图参数为 $nfea=64$ ， $d=16$ ， $s=4$ 。从第三个卷积层到第四个卷积层，采用分组卷积加快训练减少特征，其中将进行分片操作，将 $D_3(1 - 1/s)$ 作为尾部第一层卷积的输入，而剩余的 $D_3(1/s)$ 与首部卷积层的输入特征进行拼接后，与尾部卷积层的输出做加法。整个提升单元采用分组卷积，减少了特征数量提高了训练速度，同时经过首部卷积层提取的短路径特征部分继续进行长路径特征提取，而保留了部分短路径特征进行连接操作，获得了更全面的特征信息。经过提升单元的操作后，输出的特征维度为 $nfea+d$ ，我经过压缩单元将特征维度重新整理到 $nfea$ ，所以仅采用一个卷积层，输入为 $nfea+d$ ，输出为 $nfea$ ，卷积核大小为 1。最后是模块重建模块，采用一个反卷积层，输入为 $nfea$ ，输出为图片通道数，卷积核大小为 17，步长为缩放尺寸，padding 补零为 8，outputpadding 参数为 1。

论文中使用的数据集为 291 图，在二倍尺寸下，先使用尺寸为 29 的图片块进行训练，损失函数为 L1，优化函数为 Adam，mini-batch 大小设置为 64，学习率为 1×10^{-4} ，经验性的训练 10^5 次迭代作为 IDN 的初始值。然后损失函数使用 L2，学习率由 0.0001 下降 10 倍，图片块大小更改为 39，其余参数不变进行微调模型。在实现细节上，数据集我采用 DIV2K 作为数据集，使用双三次插值对图片进行下采样将图片剪切成图片块送入训练。我进行了三组实验，分别是 mini-batch 大小设置为 16，没有预训练模型进行训练，一周期进行 1000 次迭代，数据块数为 16000。mini-batch 大小设置为 64，没有预训练模型进行训练，一周期进行 50 次迭代，数据块数为 3200。mini-batch 大小设置为 16，没有预训练模型进行训练，一周期进行 100 次迭代，数据块数为 1600。我使用的设备是 GTX1080Ti，每 1000Epoch 大概需要 14 小时。具体模型性能结果见表 2.4-1，其中 IDN 为作者论文中的数据，其余参数则是由我实现的后进行实验得到的数据，Batch 和 Iteration 分别为对应的训练参数。显而易见可得，当我增加训练集图片块数后，训练结果将更加稳定，性能将更加优越，只要提升样本数，最终将可以达成与论文中相近似的性能。我设置训练参数 mini-batch 为 16，进行一周期 1000 次迭代的模型与论文模型相比虽然在 Set5 数据集 PSNR 指标稍有落后，但在结构相似性 SSIM 上我实现的模型性能比论文中基本较高，且在 Set14 数据集上我的 PSNR 指标较论文中有所提高。综上所述我实现的模型训练框架拥有复现论文中模型性

能的能力。

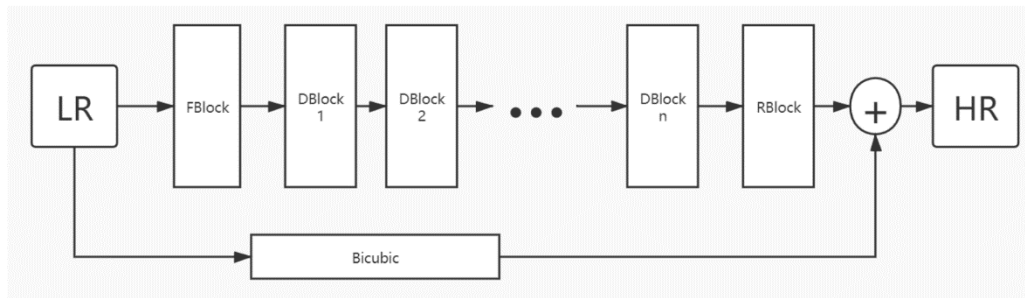


图 2.4-1 IDN 网络模型图

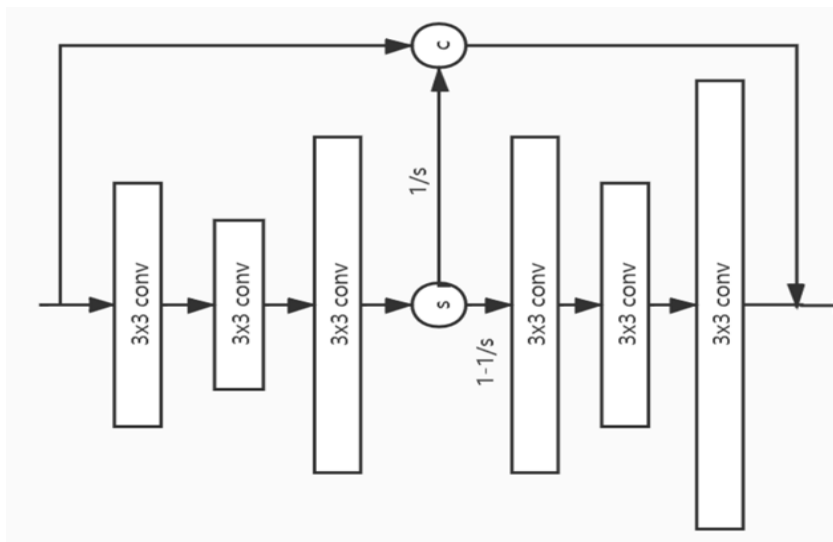


图 2.4-2 提升单元块结构图

表 2.4-1 论文 PSNR (dB)/SSIM 与实现 PSNR (dB)/SSIM 对照表

	Scale	IDN	Bitch=16, Iteration=100	Bitch=64, Iteration=500	Bitch=16, Iteration=1000	Bitch=64, Iteration=1000
Set5	× 2	37. 83/0. 9600	37. 33/0. 9583	37. 58/0. 9593	37. 74/0. 9597	37. 77/0. 9599
Set14	× 2	33. 30/0. 9148	32. 94/0. 9123	33. 19/0. 9143	33. 36/0. 9160	33. 38/0. 9163
B100	× 2	32. 08/0. 8985	31. 75/0. 8943	31. 94/0. 8971	32. 04/0. 8985	32. 05/0. 8986
Urban100	× 2	31. 27/0. 9196	30. 22/0. 9140	30. 82/0. 9220	31. 24/0. 9270	31. 32/0. 9279

2.5 SwinIR 相关细节

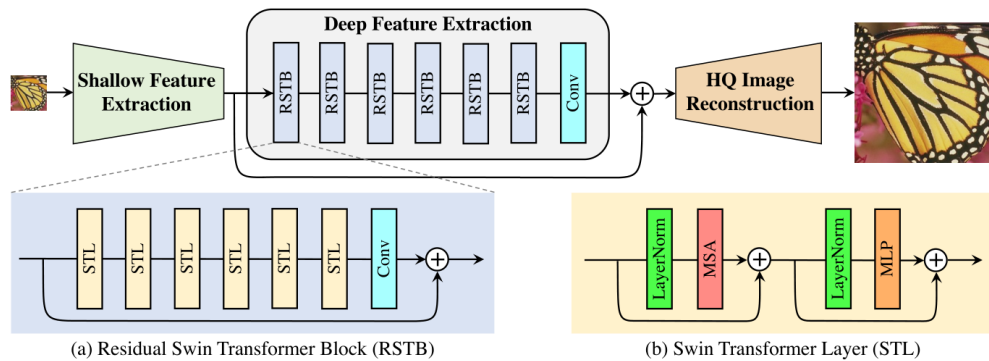


Figure 2: The architecture of the proposed SwinIR for image restoration.

图 2.5-1 SwinIR 网络模型图

SwinIR 由三个部分组成：浅层特征提取、深度特征提取和高质量图像重建，模型结构如图 2.5-1 所示。浅层特征提取模块使用卷积层来提取浅层特征，该浅层特征被直接传输到重建模块，以保存低频信息。深度特征提取模块主要由剩余 Swin Transformer 块（RSTB）组成，每个块利用几个 Swin Transformers 层进行局部关注和跨窗口交互。此外，作者在块的末尾添加卷积层以进行特征增强，并使用残差连接来提供特征聚合的快捷方式。最后，在重建模块中融合浅特征和深特征，以实现高质量的图像重建。

首先，对于浅层特征提取部分，给定低质量的输入，作者使用 3×3 卷积层 H_{SF} 提取浅层特征。接着深层特征提取模块将来自浅层特征提取模块的特征图分割成多个不重叠的 patch embeddings，再通过多个串联的残差 Swin Transformer 块（RSTB）将多个不重叠的 patch embeddings 重新组合成与输入特征图分辨率一样，最后通过一个卷积层（1 层或 3 层卷积）输出，在每个 RSTB 中都引入残差连接。其中残差 Swin Transformer 块（RSTB）中的 STL 指的就是 Swin Transformer Layer，它首先通过一个归一化层 LayerNorm，再通过多头自注意力（Multi-head Self Attention）模块，在多头自注意力结尾引入残差，再通过一个归一化层 LayerNorm，最后通过一个多层感知机 MLP，结尾同样引入残差。在图像重建模块部分，论文给出了四种结构，分别是经典超分（卷积 + pixelshuffle 上采样 + 卷积）、轻量超分（卷积 + pixelshuffle 上采样）、真实图像超分（卷积 + 卷积插值上采样 + 卷积插值上采样 + 卷积）、像去噪和 JPEG 压缩去伪影（卷积 + 引入残差）。

与流行的基于 CNN 的图像恢复模型相比，基于 Transformer 的 SwinIR 通过

移位窗口机制实现了长期依赖性建模，与现有的图像 SR 方法相比，SwinIR 以更少的参数实现了更好的 PSNR。

在我的复现过程中，我发现如果采用单卡 P40 进行迭代训练，达到论文中所描述的 100w 次 Epoch 需要 300 天以上，如果采用 V100 单卡进行训练仍需要 120 天以上。由于时间有限和设备有限，我采用了 P40 进行迭代训练了 1000 个 Epoch，同时由于显存原因缩减了 batch 大小。我仅确认了代码是可以运行进行训练的，但是由于设备原因未能完成完整的论文复现。因此在本论文中，源于 SwinIR 的所有数据，我均采用来自论文作者在论文中所贴出的数据。

3. 源代码及注释

3.1 通用模块

3.1.1 打印 loss 图

```
#加载头文件
import torch
import matplotlib.pyplot as plt
import matplotlib

torch.set_printoptions(precision=8)#设置精度
myloss=torch.load('./last_loss.pth')#设置读取路径

#打印 loss-epoch 图
myloss=myloss[1::]
#曲线刻度名
lmylabel='SRCNNx2'
lPictureName='myLossx2.jpg'

fig = plt.figure()
plt.rcParams['lines.linewidth']=2 #线条粗细
plt.rcParams['figure.figsize']=[20,12] #图大小

label='Loss'
plt.title(label)
```

```
plt.plot(myloss,label=lmylabel)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.savefig(IPictureName) #保存到本地
plt.close(fig)
```

3.1.2 简易评估 psnr 并生成 SR 图

由于生成 pytorch 下 PSNR 的计算和生成 SR 图，只需要替换对应的 model 即可，所以不放在每个 model 中讲解代码。仅在本节以 SRCNN 为例讲解代码。同理，生成单组一个 model 的 SR 图，比如第 X Epoch 的 Set5 测试集的 SR 图为一次，以此为例在下方进行讲解。想要成组生成 SR 图，仅需要在本节基础上进行循环复用即可。

```
#这里的代码是 jupyter notebook 的交互代码
import os #计算文件路径
def file_name(file_dir):
    for root, dirs, files in os.walk(file_dir):
        #print(root) #当前目录路径
        #print(dirs) #当前路径下所有子目录
        print(files) #当前路径下所有非目录子文件
    return files
#控制 c 只有需要检测的图片
c=file_name("testdata")
for i in c:
    print(i)
import os
filename = os.path.join("testdata/"+c[0])
count=len(c)
print(count) # 判断路径和路径下文件

#加载模型
import argparse
import torch
import torch.backends.cudnn as cudnn
import numpy as np
import PIL.Image as pil_image
```

```

from temodel import SRCNN
from myutils import rgb2ycbcr, ycbcr2rgb, calc_psnr
cudnn.benchmark = True
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model = SRCNN().to(device)
#加载模型参数
state_dict = model.state_dict()

#定义 psnr 函数
def get_psnr(filename,myscale):
    scale=myscale
    image = pil_image.open(filename).convert('RGB')
    image_width = (image.width // scale) * scale
    image_height = (image.height // scale) * scale
    image = image.resize((image_width, image_height), resample=pil_image.BICUBIC)
    image = image.resize((image.width // scale, image.height // scale),
resample=pil_image.BICUBIC)
    image = image.resize((image.width * scale, image.height * scale),
resample=pil_image.BICUBIC)
    image.save(filename.replace('.', '_bicubic_x{}'.format(scale)))
    image = np.array(image).astype(np.float32)
    ybcr = rgb2ycbcr(image)

    y = ybcr[..., 0]
    y /= 255.
    y = torch.from_numpy(y).to(device)
    y = y.unsqueeze(0).unsqueeze(0)

    with torch.no_grad():
        preds = model(y).clamp(0.0, 1.0)

    psnr = calc_psnr(y, preds)

    print('PSNR: {:.2f}'.format(psnr))

    preds = preds.mul(255.0).cpu().numpy().squeeze(0).squeeze(0)

    output = np.array([preds, ybcr[..., 1], ybcr[..., 2]]).transpose([1, 2, 0])
    output = np.clip(ycbcr2rgb(output), 0.0, 255.0).astype(np.uint8)
    output = pil_image.fromarray(output)
    output.save(filename.replace('.', '_srcnn_x{}'.format(scale)))#保存 SR 图

```

```
return psnr

#单独计算 psnr
modelname='./model/best.pth'
print(modelname)
myscale = 2 #设置尺度
#按路径加载模型
for n, p in torch.load(modelname, map_location=lambda storage, loc: storage).items():
    if n in state_dict.keys():
        state_dict[n].copy_(p)
    else:
        raise KeyError(n)

model.eval()
sum_psnr=0
for i in range(count):
    filename = os.path.join("testdata/"+c[i])
    sum_psnr += get_psnr(filename,myscale)#在调用 psnr 函数中已生成 SR 图

print("psnr is {:.2f}".format(sum_psnr/count))
```

3.1.3 特征图输出

以 SRCNN 为例进行示范:

```
#对 model 的返回进行了修改:
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):

        outputs = []
        x = self.relu(self.conv1(x))
        outputs.append(x)
```

```

x = self.relu(self.conv2(x))
outputs.append(x)
x = self.conv3(x)
outputs.append(x)

return outputs #如此就可以在 outputs 中接收到返回的特征图

```

在我 3.6.2 的基础上进行修改，分组输出上文返回的 `outputs` 即可，如 `out_put[0].shape = torch.Size([1, 64, 512, 512])`。

```

for i in range(len(out_put)):
    print(i)
    myimgs=out_put[i].detach().cpu().numpy()
    im = np.squeeze(myimgs)

    if not os.path.exists(os.path.join(savepath,str(i))):#创建路径
        os.mkdir(os.path.join(savepath,str(i)))

    if(im.ndim == 2):
        im = np.expand_dims(im,axis = 0)
    im=np.transpose(im,[1,2,0])
    for j in range(im.shape[2]):

        saveimg = im[:, :,j]
        print(saveimg.ndim)
        if saveimg.ndim==2:
            saveimg = np.expand_dims(saveimg,axis = 2)
        print(saveimg.shape)
        saveimg*=255.0
        saveimg=saveimg.astype('uint8')
        skimage.io.imsave(os.path.join(savepath,str(i),"{}.jpg".format(j+1)),saveimg)#保存图片

```

3.1.4 训练流程

因训练框架基本一致，只需要保证能够保存 `loss` 到本地，保存模型保证可以断点训练的 `checkpoint`。差异在于读取数据生成 `DataSet` 的形式和模型的 `model` 文件。所以将训练框架

提取到通用模块进行说明。下文以 SRCNN 为例，贴出代码并给出注释。

```

for epoch in range(args.num_epochs):
    model.train()#训练
    epoch_losses = AverageMeter()#定义求平均损失的类

    for data in train_dataloader:
        inputs, labels = data#读数据

        inputs = inputs.to(device) #放入GPU
        labels = labels.to(device)
        preds = model(inputs) #用模型得到生成的高清图
        loss = criterion(preds, labels) #计算loss
        epoch_losses.update(loss.item(), len(inputs)) #更新到定义的类中
        optimizer.zero_grad()#计算梯度
        loss.backward()#反向传播
        optimizer.step()#更新

    print( '{}_epoch avg loss : {:.8f}'.format(epoch, epoch_losses.avg)) #输出loss
    myloss=torch.full((1,1), epoch_losses.avg)
    logmyloss = torch.cat([logmyloss, myloss])
    torch.save(logmyloss, os.path.join("loss", 'last_loss.pth')) #存储loss到本地
    torch.save(model.state_dict(), os.path.join(args.outputs_dir,
'epoch_{}.pth'.format(epoch))) #存储模型到本地

    model.eval()#评估
    epoch_psnr = AverageMeter()#定义psnr存储

    for data in eval_dataloader: #评估
        inputs, labels = data #输入
        inputs = inputs.to(device) #存gpu
        labels = labels.to(device)
        with torch.no_grad():#设置评估, 加快速度
            preds = model(inputs).clamp(0.0, 1.0)
            epoch_psnr.update(calc_psnr(preds, labels), len(inputs)) #更新psnr
    print( '{}_epoch eval psnr: {:.2f}'.format(epoch, epoch_psnr.avg)) #输出到控制台
    mypsnr=torch.full((1,1), epoch_psnr.avg)
    logmypsnnr = torch.cat([logmypsnnr, mypsnnr])
    torch.save(logmypsnnr, os.path.join("mypsnnr", 'psnr_epoch.pth')) #保存psnr到本地

    if epoch_psnr.avg > best_psnr: #更新best模型
        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')) #存储
```

3.2 SRCNN

SRCNN 的整体实现代码参考了 yjn870 的 SRCNN-pytorch。在他的基础上我进行了一些功能代码的完善和补充。数据的读入方面我未做出改动。在训练的过程中,我加入了关于训练的 loss 的保存和用 pytorch 计算的 psnr 的保存,方便后期我对 loss-epoch 和 psnr-epoch 的绘制。放弃了原始代码提供给我的 eval 代码的评估,而是整体使用 MATLAB 进行计算 PSNR 和 SSIM。为此,提供了成组生成 SR 图的代码。除此之外,我在 SRCNN 中提供了打印特征图的代码,效果如图 3.1-1。生成特征图的代码见我 3.1.3。

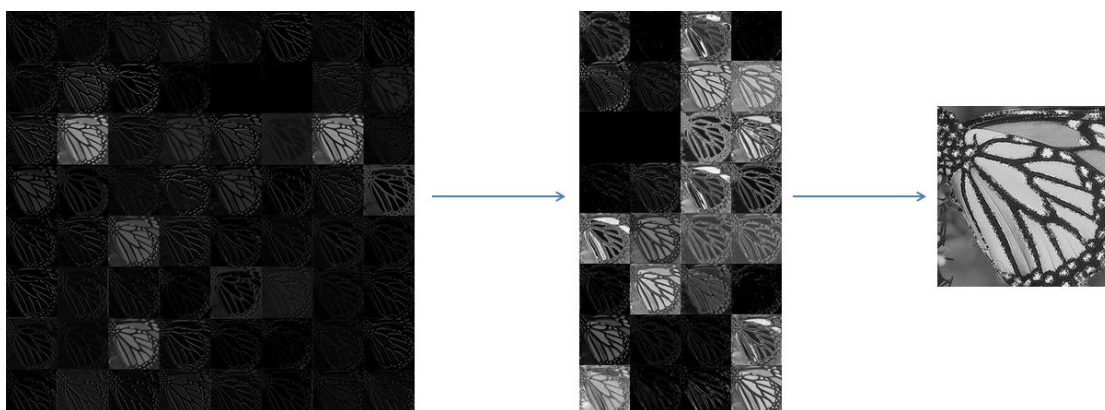


图 3.1-1 SRCNN 特征图

3.2.1 训练参数初始化

```
parser = argparse.ArgumentParser()
parser.add_argument('--train-file', type=str, required=True) #训练文件目录
parser.add_argument('--eval-file', type=str, required=True) #评估文件目录
```

```

parser.add_argument('--outputs-dir', type=str, required=True)#输出文件目录
parser.add_argument('--scale', type=int, default=2)#图片缩放尺度
parser.add_argument('--lr', type=float, default=1e-4)#学习率
parser.add_argument('--batch-size', type=int, default=16)
parser.add_argument('--num-epochs', type=int, default=400)
parser.add_argument('--num-workers', type=int, default=0)
parser.add_argument('--seed', type=int, default=123)

args = parser.parse_args()

#输出路径
args.outputs_dir = os.path.join(args.outputs_dir, 'x{}'.format(args.scale))
#如果不存在, 则创建目录
if not os.path.exists(args.outputs_dir):
    os.makedirs(args.outputs_dir)
#这样可以增加程序的运行效率
cudnn.benchmark = True
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
## 为CPU设置种子用于生成随机数, 以使得结果是确定的
torch.manual_seed(args.seed)
#设置cpu/gpu
model = SRCNN().to(device)

#设置损失函数
criterion = nn.MSELoss()
logmysnr = torch.Tensor()#用于保存psnr
logmyloss = torch.Tensor()#用于保存loss
optimizer = optim.Adam([#优化函数
    {'params': model.conv1.parameters()},
    {'params': model.conv2.parameters()},
    {'params': model.conv3.parameters(), 'lr': args.lr * 0.1}
], lr=args.lr)

```

3.2.2 数据集处理

数据集处理分为三部分。第一部分是对图片的预处理 `prepare.py`, 将图片打包成 `h5` 文件, 加快 `io` 访问速度。第二部分是 `dataset.py`, 封装成为一个类, 向上提供 `hr` 和 `lr` 训练图片。第三部分是训练 `train.py` 文件中截取的关于数据集处理的部分。

#第一部分

```

def train(args)://本函数是对训练集进行提取图片并封装保存到一个h5文件中。
    h5_file = h5py.File(args.output_path, 'w')

```

```

lr_patches = []
hr_patches = []

for image_path in sorted(glob.glob('{}/*'.format(args.images_dir))):
    hr = pil_image.open(image_path).convert('RGB')
    hr_width = (hr.width // args.scale) * args.scale
    hr_height = (hr.height // args.scale) * args.scale
    hr = hr.resize((hr_width, hr_height), resample=pil_image.BICUBIC)
    lr = hr.resize((hr_width // args.scale, hr_height // args.scale),
resample=pil_image.BICUBIC)
    lr = lr.resize((lr.width * args.scale, lr.height * args.scale),
resample=pil_image.BICUBIC)
    hr = np.array(hr).astype(np.float32)
    lr = np.array(lr).astype(np.float32)
    hr = convert_rgb_to_y(hr)
    lr = convert_rgb_to_y(lr)

    for i in range(0, lr.shape[0] - args.patch_size + 1, args.stride):
        for j in range(0, lr.shape[1] - args.patch_size + 1, args.stride):
            lr_patches.append(lr[i:i + args.patch_size, j:j + args.patch_size])
            hr_patches.append(hr[i:i + args.patch_size, j:j + args.patch_size])

lr_patches = np.array(lr_patches)
hr_patches = np.array(hr_patches)

h5_file.create_dataset('lr', data=lr_patches)
h5_file.create_dataset('hr', data=hr_patches)

h5_file.close()

```

#第二部分

#按路径从 h5 文件中读取 hr 图和 lr 图作为训练

```

class TrainDataset(Dataset):
    def __init__(self, h5_file):
        super(TrainDataset, self).__init__()
        self.h5_file = h5_file

    def __getitem__(self, idx):
        with h5py.File(self.h5_file, 'r') as f:
            return np.expand_dims(f['lr'][idx] / 255., 0), np.expand_dims(f['hr'][idx] /

```

```
255., 0)

def __len__(self):
    with h5py.File(self.h5_file, 'r') as f:
        return len(f['lr'])

class EvalDataset(Dataset):
    def __init__(self, h5_file):
        super(EvalDataset, self).__init__()
        self.h5_file = h5_file

    def __getitem__(self, idx):
        with h5py.File(self.h5_file, 'r') as f:
            return np.expand_dims(f['lr'][str(idx)][:, :] / 255., 0),
np.expand_dims(f['hr'][str(idx)][:, :] / 255., 0)

    def __len__(self):
        with h5py.File(self.h5_file, 'r') as f:
            return len(f['lr'])
```

#第三部分

```
#设置dataset
train_dataset = TrainDataset(args.train_file)

#DataLoader加载
train_dataloader = DataLoader(dataset=train_dataset,
                               batch_size=args.batch_size,
                               shuffle=True,
                               num_workers=args.num_workers,
                               pin_memory=True,
                               drop_last=True)

#评测数据集
eval_dataset = EvalDataset(args.eval_file)
eval_dataloader = DataLoader(dataset=eval_dataset, batch_size=1)

best_weights = copy.deepcopy(model.state_dict())
best_epoch = 0
best_psnr = 0.0
```

3.2.3 模型

```
from torch import nn

class SRCNN(nn.Module): #三层结构的SRCNN模型
    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
```

3.2 VDSR

VDSR 在代码方面我参考了 twtygqyy 的 `pytorch-vdsr`。并作出如下改动：首先我在数据集的生成部分参考了 VDSR 的做法，将

3.2.1 训练参数初始化

```
import argparse
from math import log10, sqrt
import time
```

```
import os
import errno
from os.path import join

import torch
import torch.nn as nn
import torch.optim as optim
import torch.backends.cudnn as cudnn
from torch.utils.data import DataLoader
from torch.autograd import Variable
from dataset import DatasetFromHdf5
from model import VDSR
from myutils import AverageMeter

my_cuda = True
#测试设置
my_TorF_test= False
my_test_batch_size=128
my_gpuids=[0]
my_model=""
my_threads = 1
#训练设置
my_start_epoch=1
my_epochs=50
my_gpus=0
my_batch_size=64
my_clip=0.4
my_lr=0.1
# =====>衰减
my_step=10
my_momentum=0.9

my_upscale_factor=2
my_weight_decay=0.0001

#恢复 Path to checkpoint (default: none)
my_resume=""

#预训练模型的路径（默认值：无）
my_pretrained=""
#用来保存loss的
logmyloss=torch.Tensor()
```

```
#优化函数
print("==> Setting Optimizer")
optimizer = optim.SGD(model.parameters(),
                       lr=my_lr,
                       momentum=my_momentum,
                       weight_decay=my_weight_decay)

#加载模型
if my_pretrained:
    if os.path.isfile(my_pretrained):
        print("=> loading model ' {}'.format(my_pretrained))
        weights = torch.load(my_pretrained)
        model.load_state_dict(weights['model'].state_dict())
    else:
        print("=> no model found at ' {}'.format(my_pretrained))

#继续训练
if my_resume:
    if os.path.isfile(my_resume):
        print("=> loading checkpoint ' {}'.format(my_resume))
        checkpoint = torch.load(my_resume)

        my_start_epoch = checkpoint["epoch"] + 1
        print(my_start_epoch)
        model.load_state_dict(checkpoint["model"].state_dict())
    else:
        print("=> no checkpoint found at ' {}'.format(my_resume))

#学习率调整
def adjust_learning_rate(optimizer, epoch):
    lr = my_lr * (0.1 ** (epoch // my_step))
    return lr
```

3.2.2 数据集处理

VDSR 中，数据集处理分为三部分。第一部分是关于训练的主函数中的相关的数据集的

加载和处理部分。第二部分是提供的 **DataSet** 的部分。第三部分是最早需要进行的数据预处理部分。将图片数据处理成 **h5** 的文件，用于后面训练。

第一部分

这部分是 **train** 中关于数据集的加载处理部分。

```
#
global my_start_epoch
cuda = my_cuda
if cuda:
    print("> use gpu id: '{}'".format(my_gpus))
    # os.environ["CUDA_VISIBLE_DEVICES"] = my_gpus
    if not torch.cuda.is_available():
        raise Exception("No GPU found or Wrong gpu id, please run without --cuda")

print("==> Loading datasets")
train_set = DatasetFromHdf5("data/train.h5")#这就是数据集的加载
training_data_loader = DataLoader(dataset=train_set,
                                   num_workers=my_threads,
                                   batch_size=my_batch_size,
                                   shuffle=True)
```

第二部分

这部分是 **DataSet** 部分:

```
import torch.utils.data as data
import torch
import h5py

class DatasetFromHdf5(data.Dataset):
    def __init__(self, file_path):
        super(DatasetFromHdf5, self).__init__()
        hf = h5py.File(file_path)
        self.data = hf.get('data')
        self.target = hf.get('label')
```

```
def __getitem__(self, index):  
    return torch.from_numpy(self.data[index, :, :, :]).float(),  
    torch.from_numpy(self.target[index, :, :, :]).float()  
  
def __len__(self):  
    return self.data.shape[0]
```

第三部分：

这部分是数据预处理部分：

```
clear;close all;  
  
folder = 'path/to/train/folder';  
  
savepath = 'train.h5';  
size_input = 41;  
size_label = 41;  
stride = 41;  
  
created_flag = false;  
  
%% 控制尺度  
scale = [2,3,4];  
  
%% 控制下采样  
downsizes = [1,0.7,0.5];  
  
%% 初始化  
data = zeros(size_input, size_input, 1, 1);  
label = zeros(size_label, size_label, 1, 1);  
  
count = 0;  
margin = 0;  
  
%% 生成数据  
filepaths = [];
```

```

filepaths = [filepaths; dir(fullfile(folder,
'*.*jpg'))];
filepaths = [filepaths; dir(fullfile(folder,
'*.*bmp'))];

for i = 1 : length(filepaths)
    for flip = 1: 3
        for degree = 1 : 4
            for s = 1 : length(scale)
                for downsize = 1 : length(downsizes)
                    image =
imread(fullfile(folder,filepaths(i).name));

                    if flip == 1
                        image = flipdim(image ,1);
                    end
                    if flip == 2
                        image = flipdim(image ,2);
                    end

                    image = imrotate(image, 90 * (degree -
1));

                    image =
imresize(image,downsizes(downsize),'bicubic');

                    if size(image,3)==3
                        image = rgb2ycbcr(image);
                        image = im2double(image(:, :, 1));

                        im_label = modcrop(image,
scale(s));

                        [hei, wid] = size(im_label);
                        im_input =
imresize(imresize(im_label,1/scale(s),'bicubic'), [hei
, wid], 'bicubic');

                        filepaths(i).name
                        for x=1 : stride : hei-size_input+1
                            for y = 1 :stride :
wid-size_input+1

                                subim_input = im_input(x :
x+size_input-1, y : y+size_input-1);
                                subim_label = im_label(x :

```

```

x+size_label-1, y : y+size_label-1);

                                count=count+1;

                                data(:, :, 1, count) =
subim_input;                                label(:, :, 1, count) =
subim_label;

                                end
                                end
                                end
                                end
                                end
                                end
                                end
                                end

order = randperm(count);
data = data(:, :, 1, order);
label = label(:, :, 1, order);

%% 写H5文件
chunksz = 64;

totalct = 0;

for batchno = 1:floor(count/chunksz)
    batchno
    last_read=(batchno-1)*chunksz;
    batchdata =
data(:, :, 1, last_read+1:last_read+chunksz);
    batchlabs =
label(:, :, 1, last_read+1:last_read+chunksz);

    startloc = struct('dat',[1,1,1,totalct+1], 'lab',
[1,1,1,totalct+1]);
    curr_dat_sz = store2hdf5(savepath, batchdata,
batchlabs, ~created_flag, startloc, chunksz);
    created_flag = true;
    totalct = curr_dat_sz(end);
end

h5disp(savepath);

```

3.1.3 模型

#VDSR 的中间单元块

```
class Conv_ReLU_Block(nn.Module):
    def __init__(self):
        super(Conv_ReLU_Block, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=64,
                                out_channels=64,
                                kernel_size=3,
                                stride=1,
                                padding=1,
                                bias=False)

        self.relu = nn.ReLU(inplace = True)

    def forward(self, x):
        return self.relu(self.conv1(x))

class VDSR(nn.Module):
    def __init__(self):
        super(VDSR, self).__init__()
        self.residual_layer = self.make_layer(Conv_ReLU_Block, 18)
        self.input = nn.Conv2d(in_channels=3,
                                out_channels=64,
                                kernel_size=3,
                                stride=1,
                                padding=1,
                                bias=False)

        self.output = nn.Conv2d(in_channels=64,
                                out_channels=3,
                                kernel_size=3,
```

```
        stride=1,
        padding=1,
        bias=False)#bias  piany

self.relu = nn.ReLU(inplace = True)

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
        m.weight.data.normal_(0, sqrt(2. / n))

#使用函数封装，采用循环的方式构建VDSR中间的重复的单元块
def make_layer(self, block, num_of_layer):
    layers = []
    for _ in range(num_of_layer):
        layers.append(block())

    return nn.Sequential(*layers)

def forward(self, x):
#残差 res
    residual = x
    out = self.relu(self.input(x))
    out = self.residual_layer(out)
    out = self.output(out)
#残差相加
    out = torch.add(out, residual)

    return out
```

3.3 RCAN

一开始我借鉴了 RCAN 作者提供的 pytorch 的代码，首先我解决了他的版本适配问题，然后重写了他提供的多尺度的混合训练的部分，改为只在单尺度进行训练。最后我放弃了他的整体的代码。选择重新构建一个框架。保留了 RCAN 的数据预处理和读入部分，以及 rcan

的 model, 训练部分, 和模型的 checkpoint 以及中间的 loss 的保存等我都采用一个自己搭建的流程。这个流程依然适用于 IDN 等模型的训练。在评估部分我依旧采用 RCAN 作者提供的 MATLAB 进行评估。

3.3.1 训练参数初始化

```
#train.py 中部分代码
import argparse
from math import log10, sqrt
import time
import os
import errno
from os.path import join
import copy
import torch
import torch.nn as nn
import torch.optim as optim
import torch.backends.cudnn as cudnn
from torch.utils.data import DataLoader
from torch.autograd import Variable
from dataset import Dataset
from rcnn import make_model
from myutils import AverageMeter, calc_psnr

torch.manual_seed(1)
my_cuda = True
#下面开始参数配置
my_model=""
my_threads=8
#训练设置
my_start_epoch=1#开始的epoch, 用于恢复训练
my_gpus=0#gpu
my_images_dir='data' #数据加载路径
my_batch_size=16#批次
# =====>训练周期与lr衰减
my_epochs=400
my_step=200
#恢复 Path to checkpoint (default: none)
my_resume="."
#预训练模型的路径(默认值: 无)
my_pretrained="" #用于恢复的模型名
```

```
logmyloss=torch.Tensor()
#优化函数参数
my_lr=0.0001
#模型参数，用于构建rcan
class modelargs():
    def __init__(self):
        self.n_resgroups=10
        self.n_resblocks=20
        self.n_feats=64
        self.reduction=16
        self.scale=[2]
        self.rgb_range=255
        self.n_colors=3
        self.res_scale=1

#学习率调整
def adjust_learning_rate(optimizer, epoch):
    """Sets the learning rate to the initial LR decayed by 10 every 10 epochs"""
    lr = my_lr * (0.5 ** (epoch // my_step))
    return lr

#保存模型

def save_checkpoint(model, epoch):
    model_out_path = "checkpoint/" + "model_epoch_{}.pth".format(epoch)

    if not os.path.exists("checkpoint/"):
        os.makedirs("checkpoint/")
    torch.save(model.state_dict(), model_out_path)

    print("Checkpoint saved to {}".format(model_out_path))
```

3.3.2 数据集处理部分

数据集处理分为三部分，分别是数据预处理、DataSet 和 train 中的数据处理。

第一部分

这部分主要是把原始的图像图片转为 npy 的格式，加快 io 读入处理。

```
if args.ext == 'img' or benchmark:
    self.images_hr, self.images_lr = self._scan()
elif args.ext.find('sep') >= 0:
    self.images_hr, self.images_lr = self._scan()
if args.ext.find('reset') >= 0: #原始的参数为: sep_reset时, 会执行这里
    print('Preparing seperated binary files')
    for v in self.images_hr:
        hr = misc.imread(v)
        #print("now hr is :{}".format(hr))
        name_sep = v.replace(self.ext, '.npy')
        np.save(name_sep, hr) #保存hr图为numpy
    for si, s in enumerate(self.scale):
        for v in self.images_lr[si]:
            lr = misc.imread(v)
            name_sep = v.replace(self.ext, '.npy')
            np.save(name_sep, lr) #保存lr 图为 numpy
```

第二部分

第二部分是关于提供 DataSet 部分的。

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import random
import glob
import numpy as np
import PIL.Image as pil_image
import torch

class Dataset(object):
    def __init__(self, images_dir, patch_size, scale, use_fast_loader=False):
```

```
self.image_files_lr = sorted(glob.glob(images_dir + '/LR' + '/*.npy'))
self.image_files_hr = sorted(glob.glob(images_dir + '/HR' + '/*.npy'))
#初始化
self.patch_size = patch_size
self.scale = scale
self.use_fast_loader = use_fast_loader

def __getitem__(self, idx):
    #提供数据
    #print("idx: {}".format(idx))
    idx = self._get_index(idx)
    #print("now is : {}".format(idx))
    lr = np.load(self.image_files_lr[idx])
    hr = np.load(self.image_files_hr[idx])
    lr, hr = get_patch(lr, hr, self.patch_size * self.scale, self.scale)
    lr, hr = augment([lr, hr])
    lr_tensor, hr_tensor = np2Tensor([lr, hr], 255)
    #print("lr: {}".format(lr_tensor.size()))
    #print("hr: {}".format(hr_tensor.size()))

    # normalization
    lr_tensor /= 255.0
    hr_tensor /= 255.0

    return lr_tensor, hr_tensor
#长度
def __len__(self):
    #print(len(self.image_files_hr))
    return len(self.image_files_hr)
def _get_index(self, idx):

    return idx % len(self.image_files_hr)

def np2Tensor(l, rgb_range):
    def _np2Tensor(img):
        np_transpose = np.ascontiguousarray(img.transpose((2, 0, 1)))
        #可以这样认为, ascontiguousarray函数将一个内存不连续存储的数组转换为内存连续存储
        #的数组, 使得运行速度更快。
        #转换为 c height weight
        tensor = torch.from_numpy(np_transpose).float()
```

```
        tensor.mul_(rgb_range / 255)

    return tensor

    return [_np2Tensor(_l) for _l in l]

#提供图片块
def get_patch(img_in, img_tar, patch_size, scale, multi_scale=False):#随机裁剪一块
    ih, iw = img_in.shape[:2]

    p = scale if multi_scale else 1
    tp = p * patch_size
    ip = tp // scale

    ix = random.randrange(0, iw - ip + 1)
    iy = random.randrange(0, ih - ip + 1)
    tx, ty = scale * ix, scale * iy

    img_in = img_in[iy:iy + ip, ix:ix + ip, :]
    img_tar = img_tar[ty:ty + tp, tx:tx + tp, :]

    return img_in, img_tar

#数据增强
def augment(l, hflip=True, rot=True):
    hflip = hflip and random.random() < 0.5
    vflip = rot and random.random() < 0.5
    rot90 = rot and random.random() < 0.5

    def _augment(img):
        if hflip: img = img[:, ::-1, :]
        if vflip: img = img[::-1, :, :]
        if rot90: img = img.transpose(1, 0, 2)

        return img

    return [_augment(_l) for _l in l]
```

第三部分

train 中的关于数据处理的部分。

```
train_set = Dataset(my_images_dir, 48, 2, False)#pacth=48 scale=2
training_data_loader = DataLoader(dataset=train_set,
                                   num_workers=my_threads,
                                   batch_size=my_batch_size,
                                   shuffle=True,
                                   pin_memory=True,
                                   drop_last=True)
```

3.3.3 模型

```
from torch import nn
#用这个函数来返回RCAN
def make_model(args, parent=False):
    return RCAN(args)

#注意力机制的实现
class ChannelAttention(nn.Module):
    def __init__(self, num_features, reduction):
        super(ChannelAttention, self).__init__()
        self.module = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(num_features, num_features // reduction, kernel_size=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num_features // reduction, num_features, kernel_size=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return x * self.module(x)

#RCAB块
class RCAB(nn.Module):
    def __init__(self, num_features, reduction):
        super(RCAB, self).__init__()
```

```

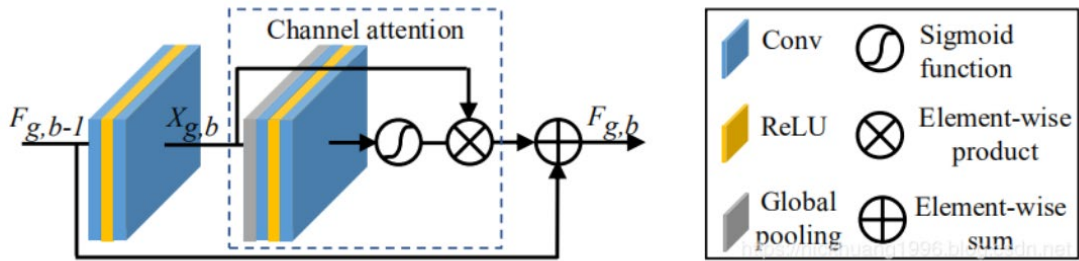
self.module = nn.Sequential(
    nn.Conv2d(num_features, num_features, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(num_features, num_features, kernel_size=3, padding=1),
    ChannelAttention(num_features, reduction)
)

```

```

def forward(self, x):
    return x + self.module(x)

```



RCAB块

#RG是RCAB的Group，一组的意思，RG里面包含若干个RCAB，可以通过RCAN的创建参数来自由的调整。

class RG(nn.Module):#rg模块

```

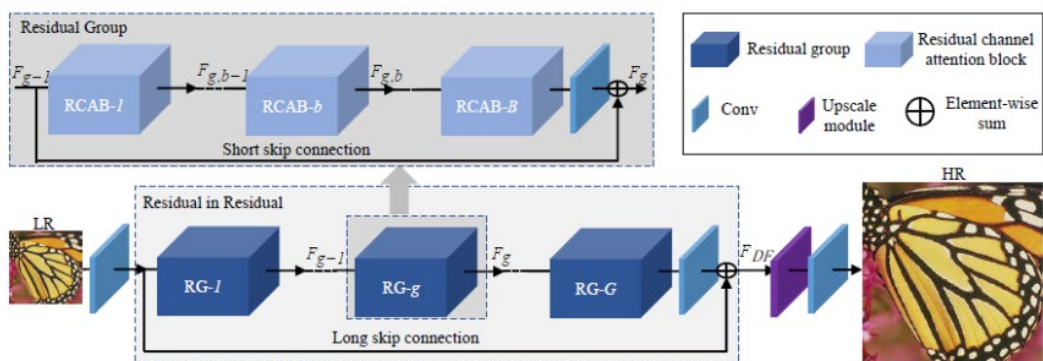
def __init__(self, num_features, num_rcab, reduction):
    super(RG, self).__init__()
    self.module = [RCAB(num_features, reduction) for _ in range(num_rcab)]
    self.module.append(nn.Conv2d(num_features, num_features, kernel_size=3,
padding=1))
    self.module = nn.Sequential(*self.module)

```

```

def forward(self, x):
    return x + self.module(x)

```



RG块

#RCAN的主体部分

```
class RCAN(nn.Module):
    def __init__(self, args):
        super(RCAN, self).__init__()
        scale = args.scale[0]
        num_features = args.n_feats
        num_rg = args.n_resgroups
        num_rcab = args.n_resblocks
        reduction = args.reduction

        self.sf = nn.Conv2d(3, num_features, kernel_size=3, padding=1)
        self.rgs = nn.Sequential(*[RG(num_features, num_rcab, reduction) for _ in
range(num_rg)])
        self.conv1 = nn.Conv2d(num_features, num_features, kernel_size=3, padding=1)
        self.upscale = nn.Sequential(
            nn.Conv2d(num_features, num_features * (scale ** 2), kernel_size=3,
padding=1),
            nn.PixelShuffle(scale)
        )
        self.conv2 = nn.Conv2d(num_features, 3, kernel_size=3, padding=1)

    def forward(self, x):
        x = self.sf(x)
        residual = x
        x = self.rgs(x)
        x = self.conv1(x)
        x += residual
        x = self.upscale(x)
        x = self.conv2(x)
        return x
```

3.3.4 训练框架的 train 函数与 Epoch 循环

与一开始我 3.1 中相似，只不过这里我对 train 进行了封装，使得结构更为清晰，只需要在 Epoch 的循环中调用 train 即可。

Epoch 循环调用实例

```
for epoch in range(my_start_epoch, my_epochs + 1):
    start_time = time.time()
    train(training_data_loader, optimizer, model, criterion, epoch)
    elapsed_time = time.time() - start_time
    print("Now the {} time is :{}".format(epoch, elapsed_time))

    save_checkpoint(model, epoch)
```

训练框架中的 train 函数

```
def train(training_data_loader, optimizer, model, criterion, epoch):

    epoch_losses = AverageMeter()

    lr = adjust_learning_rate(optimizer, epoch-1)

    for param_group in optimizer.param_groups:
        param_group["lr"] = lr

    epoch_loss = 0
    print("Epoch = {}, lr = {}".format(epoch, optimizer.param_groups[0]["lr"]))

    model.train()
    #BatchNormalization 和 Dropout
    for iteration, batch in enumerate(training_data_loader, 1):
        input, target = batch

        if my_cuda:
            input = input.cuda()
            target = target.cuda()

        #梯度清除
        optimizer.zero_grad()
```

```
# print(input.size())
# print(target.size())
# print("mymodel")

loss = criterion(model(input), target)
epoch_loss += loss.item()
#print("training -----")
#print("loss is{} :".format(loss.item()))
epoch_losses.update(loss.item(), len(input))

loss.backward()

optimizer.step()

if iteration%10 == 0:
    print("==> Epoch[{}]({} / {}): Loss: {:.10f}".format(
        epoch, iteration, len(training_data_loader),
        loss.item()))

print("==> Epoch {} Complete: Avg. Loss: {:.8f}".format(
    epoch, epoch_loss / len(training_data_loader)))
print("ok!-----")
print(' {}_epoch avg loss : {:.8f}'
      .format(epoch, epoch_losses.avg))
myloss=torch.full((1,1), epoch_losses.avg)

global logmyloss

logmyloss = torch.cat([logmyloss, myloss])
torch.save(logmyloss, os.path.join("loss", 'last_loss.pth'))
```


3.4 IDN

IDN 的训练代码是我在总结前几个模型的基础上迭代而来的。数据的读入部分采用的依旧是 RCAN 的 npy 的数据处理。整体的框架与前文通用较为类似。

3.4.1 训练参数初始化

```
torch.manual_seed(1)
my_cuda = True
my_loss='l1'
my_model=""
my_threads=8
#训练设置
my_patch=11
my_start_epoch=274
my_gpus=0
my_images_dir='data'
my_batch_size=16

# =====>训练周期与lr衰减
my_epochs=1000
my_step=1000

#恢复 Path to checkpoint (default: none)
my_resume=""

#预训练模型的路径（默认值：无）
my_pretrained="./checkpoint/model_epoch_273.pth"

logmyloss=torch.Tensor()

#优化函数参数
my_lr=0.0001

scale=8
```

```
#模型参数，用于构建IDN
class modelargs():
    def __init__(self):
        self.n_resgroups=10
        self.n_resblocks=20
        self.n_feats=64
        self.reduction=16
        self.scale=8
        self.rgb_range=255
        self.n_colors=3
        self.res_scale=1
        self.d=16
        self.s=4

model = IDN(modelargs())

#加载
if my_pretrained is not None:
    state_dict = model.state_dict()
    for n, p in torch.load(my_pretrained, map_location=lambda storage, loc:
storage).items():
        if n in state_dict.keys():
            state_dict[n].copy_(p)
        else:
            raise KeyError(n)

#损失函数

if my_loss == 'l1':
    criterion = nn.L1Loss()
elif my_loss == 'l2':
    criterion = nn.MSELoss()
else:
    raise Exception(' Loss Error: "{}".format(opt.loss))

#优化函数
print("====> Setting Optimizer")

optimizer = optim.Adam(model.parameters(), lr=my_lr)

print("====> Setting GPU")
```

```
if cuda:
    model = model.cuda()
    criterion = criterion.cuda()

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

best_weights = copy.deepcopy(model.state_dict())
best_epoch = 0
best_psnr = 0.0
logmysnr = torch.Tensor()
logmyloss = torch.Tensor()
```

3.4.2 数据集处理部分

依旧分为三部分。预处理、DataSet、已经 train 中关于数据集的处理部分。但是由于训练框架仍采用与上文相同的框架与处理方式。故预处理与 DataSet 与我 RCAN 的 3.3.2 中第一部分与第二部分相同。下面给出关于 train 中的数据集处理部分：

```
train_set = Dataset(my_images_dir, my_patch, scale, False)#pacth=48 scale=2
training_data_loader = DataLoader(dataset=train_set,
                                   num_workers=my_threads,
                                   batch_size=my_batch_size,
                                   shuffle=True,
                                   pin_memory=True,
                                   drop_last=True)
```

3.4.3 模型

整体的模型描述与每个模块的作用与构成见我 2.4 部分。本节仅指出代码与模块的对应。

```
import torch
from torch import nn
import torch.nn.functional as F

class FBlock(nn.Module):#开头的FB单元
    def __init__(self, num_features):
        super(FBlock, self).__init__()
        self.module = nn.Sequential(
            nn.Conv2d(3, num_features, kernel_size=3, padding=1),
            nn.LeakyReLU(0.05),
            nn.Conv2d(num_features, num_features, kernel_size=3, padding=1),
            nn.LeakyReLU(0.05)
        )

    def forward(self, x):
        return self.module(x)

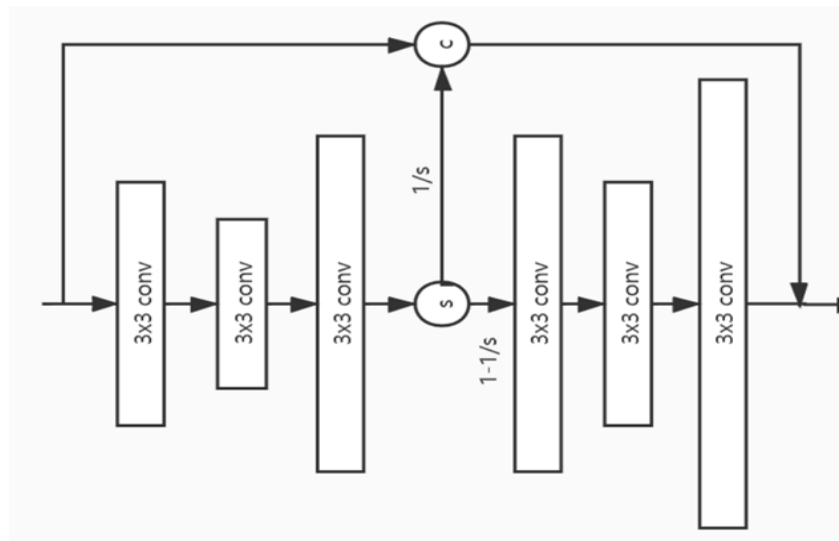
class DBlock(nn.Module):#中间的DB单元
    def __init__(self, num_features, d, s):
        super(DBlock, self).__init__()
        self.num_features = num_features
        self.s = s
        self.enhancement_top = nn.Sequential(
            nn.Conv2d(num_features, num_features - d, kernel_size=3, padding=1),
            nn.LeakyReLU(0.05),
            nn.Conv2d(num_features - d, num_features - 2 * d, kernel_size=3, padding=1,
groups=4),
            nn.LeakyReLU(0.05),
            nn.Conv2d(num_features - 2 * d, num_features, kernel_size=3, padding=1),
            nn.LeakyReLU(0.05)
        )
        self.enhancement_bottom = nn.Sequential(
            nn.Conv2d(num_features - d, num_features, kernel_size=3, padding=1),
            nn.LeakyReLU(0.05),
            nn.Conv2d(num_features, num_features - d, kernel_size=3, padding=1, groups=4),
            nn.LeakyReLU(0.05),
            nn.Conv2d(num_features - d, num_features + d, kernel_size=3, padding=1),
```

```

        nn.LeakyReLU(0.05)
    )
    self.compression = nn.Conv2d(num_features + d, num_features, kernel_size=1)

    def forward(self, x):
        residual = x
        x = self.enhancement_top(x)
        slice_1 = x[:, :int((self.num_features - self.num_features/self.s)), :, :]
        slice_2 = x[:, int((self.num_features - self.num_features/self.s)):, :, :]
        x = self.enhancement_bottom(slice_1)
        x = x + torch.cat((residual, slice_2), 1)
        x = self.compression(x)
    return x

```



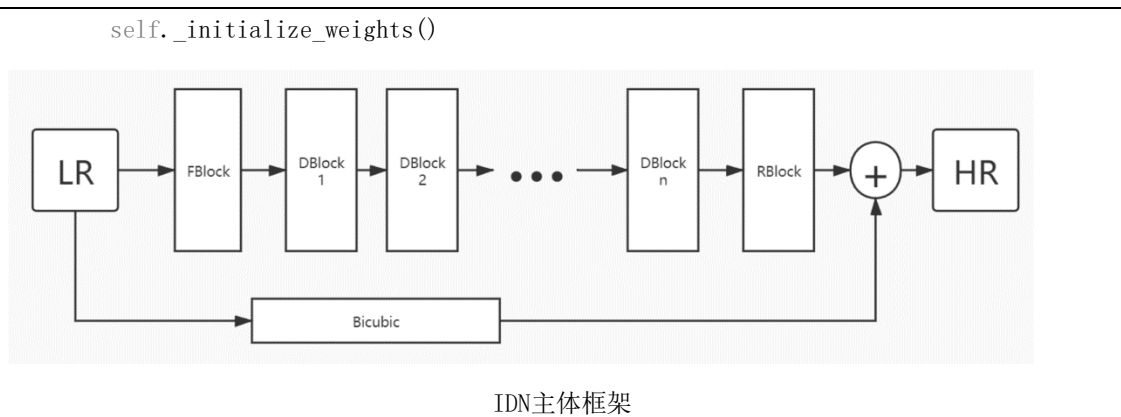
DB单元

```

class IDN(nn.Module):#IDN主程序块
    def __init__(self, args):
        super(IDN, self).__init__()
        self.scale = args.scale
        num_features = args.n_feats
        d = args.d
        s = args.s
        print("my scale is :{}".format(self.scale))

        self.fblock = FBlock(num_features)
        self.dblocks = nn.Sequential(*[DBlock(num_features, d, s) for _ in range(4)])
        self.deconv = nn.ConvTranspose2d(num_features, 3, kernel_size=17,
stride=self.scale, padding=8, output_padding=7)

```



```
def _initialize_weights(self):#初始化模型参数
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight)
            nn.init.zeros_(m.bias)
        if isinstance(m, nn.ConvTranspose2d):
            nn.init.kaiming_normal_(m.weight)
            nn.init.zeros_(m.bias)

    def forward(self, x):
        bicubic = F.interpolate(x, scale_factor=self.scale, mode='bicubic',
                                align_corners=False)

        x = self.fblock(x)
        x = self.dblocks(x)
        x = self.deconv(x)
        # print("x size is :{}".format(x.size()))
        # print("bi size is :{}".format(bicubic.size()))

        return bicubic + x
```

3.4.4 训练

同我 3.3.4 部分。

3.5 SwinIR

SwinIR 的代码主要参考了论文作者的开源代码。我在源代码的基础上进行了精简，去掉了一些由于原论文作者为了复用而构建的框架。我仅实现了 SR 部分的重建与训练。下文主要介绍 SwinIR 的结构。

3.5.1 SwinIR 主体

SwinIR 主要由浅层特征提取，深层特征提取和高质量图像重建模块组成。

```
# SwinIR
class SwinIR(nn.Module):
    """ SwinIR
        基于 Swin Transformer 的图像恢复网络.

    输入:
        img_size (int | tuple(int)): 输入图像的大小, 默认为 64*64.
        patch_size (int | tuple(int)): patch 的大小, 默认为 1.
        in_chans (int): 输入图像的通道数, 默认为 3.
        embed_dim (int): Patch embedding 的维度, 默认为 96.
        depths (tuple(int)): Swin Transformer 层的深度.
        num_heads (tuple(int)): 在不同层注意力头的个数.
        window_size (int): 窗口大小, 默认为 7.
        mlp_ratio (float): MLP 隐藏层特征图通道与嵌入层特征图通道的比, 默认为 4.
        qkv_bias (bool): 给 query, key, value 添加可学习的偏置, 默认为 True.
        qk_scale (float): 重写默认的缩放因子, 默认为 None.
        drop_rate (float): 随机丢弃神经元, 丢弃率默认为 0.
        attn_drop_rate (float): 注意力权重的丢弃率, 默认为 0.
        drop_path_rate (float): 深度随机丢弃率, 默认为 0.1.
        norm_layer (nn.Module): 归一化操作, 默认为 nn.LayerNorm.
        ape (bool): patch embedding 添加绝对位置 embedding, 默认为 False.
        patch_norm (bool): 在 patch embedding 后添加归一化操作, 默认为 True.
        use_checkpoint (bool): 是否使用 checkpointing 来节省显存, 默认为 False.
        upscale: 放大因子, 2/3/4/8 适合图像超分, 1 适合图像去噪和 JPEG 压缩去伪影
        img_range: 灰度值范围, 1 或者 255.
        upsampler: 图像重建方法的选择模块, 可选择 pixelshuffle, pixelshuffledirect, nearest+conv 或 None.
        resi_connection: 残差连接之前的卷积块, 可选择 1conv 或 3conv.
    """
```

```

def __init__(self, img_size=64, patch_size=1, in_chans=3,
              embed_dim=96, depths=[6, 6, 6, 6], num_heads=[6, 6, 6, 6],
              window_size=7, mlp_ratio=4., qkv_bias=True, qk_scale=None,
              drop_rate=0., attn_drop_rate=0., drop_path_rate=0.1,
              norm_layer=nn.LayerNorm, ape=False, patch_norm=True,
              use_checkpoint=False, upscale=2, img_range=1., upsampler="", resi_connection='1conv',
              **kwargs):
    super(SwinIR, self).__init__()
    num_in_ch = in_chans # 输入图片通道数
    num_out_ch = in_chans # 输出图片通道数
    num_feat = 64 # 特征图通道数
    self.img_range = img_range # 灰度值范围:[0, 1] or [0, 255]
    if in_chans == 3: # 如果输入是 RGB 图像
        rgb_mean = (0.4488, 0.4371, 0.4040) # 数据集 RGB 均值
        self.mean = torch.Tensor(rgb_mean).view(1, 3, 1, 1) # 转为[1, 3, 1, 1]的张量
    else: # 否则灰度图
        self.mean = torch.zeros(1, 1, 1, 1) # 构造[1, 1, 1, 1]的张量
    self.upscale = upscale # 图像放大倍数, 超分(2/3/4/8),去噪(1)
    self.upsampler = upsampler # 上采样方法
    self.window_size = window_size # 注意力窗口的大小

    #####
    ##### 1, 浅层特征提取 #####
    self.conv_first = nn.Conv2d(num_in_ch, embed_dim, 3, 1, 1) # 输入卷积层

    #####
    ##### 2, 深层特征提取 #####
    self.num_layers = len(depths) # Swin Transformer 层的个数
    self.embed_dim = embed_dim # 嵌入层特征图的通道数
    self.ape = ape # patch embedding 添加绝对位置 embedding, 默认为 False.
    self.patch_norm = patch_norm # 在 patch embedding 后添加归一化操作, 默认为 True.
    self.num_features = embed_dim # 特征图的通道数
    self.mlp_ratio = mlp_ratio # MLP 隐藏层特征图通道与嵌入层特征图通道的比

    # 将图像分割成多个不重叠的 patch
    self.patch_embed = PatchEmbed(
        img_size=img_size, patch_size=patch_size, in_chans=embed_dim, embed_dim=embed_dim,
        norm_layer=norm_layer if self.patch_norm else None)
    num_patches = self.patch_embed.num_patches # 分割得到 patch 的个数
    patches_resolution = self.patch_embed.patches_resolution # 分割得到 patch 的分辨率
    self.patches_resolution = patches_resolution

    # 将多个不重叠的 patch 合并成图像

```



```

self.patch_unembed = PatchUnEmbed(
    img_size=img_size, patch_size=patch_size, in_chans=embed_dim, embed_dim=embed_dim,
    norm_layer=norm_layer if self.patch_norm else None)

# 绝对位置嵌入
if self.ape:
    # 结构为 [1, patch 个数, 嵌入层特征图的通道数] 的参数
    self.absolute_pos_embed = nn.Parameter(torch.zeros(1, num_patches, embed_dim))
    trunc_normal_(self.absolute_pos_embed, std=.02) # 截断正态分布, 限制标准差为 0.02

self.pos_drop = nn.Dropout(p=drop_rate) # 以 drop_rate 为丢弃率随机丢弃神经元, 默认不丢弃

# 随机深度衰减规律, 默认为 [0, 0.1] 进行 24 等分后的列表
dpr = [x.item() for x in torch.linspace(0, drop_path_rate, sum(depths))]

# Residual Swin Transformer blocks (RSTB)
# 残差 Swin Transformer 块 (RSTB)
self.layers = nn.ModuleList() # 创建一个 ModuleList 实例对象, 也就是多个 RSTB
for i_layer in range(self.num_layers): # 循环 Swin Transformer 层的个数次
    # 实例化 RSTB
    layer = RSTB(dim=embed_dim,
                  input_resolution=(patches_resolution[0],
                                   patches_resolution[1]),
                  depth=depths[i_layer],
                  num_heads=num_heads[i_layer],
                  window_size=window_size,
                  mlp_ratio=self.mlp_ratio,
                  qkv_bias=qkv_bias, qk_scale=qk_scale,
                  drop=drop_rate, attn_drop=attn_drop_rate,
                  drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])], # no impact on SR results
                  norm_layer=norm_layer,
                  downsample=None,
                  use_checkpoint=use_checkpoint,
                  img_size=img_size,
                  patch_size=patch_size,
                  resi_connection=resi_connection
                  )

    self.layers.append(layer) # 将 RSTB 对象插入 ModuleList 中
self.norm = norm_layer(self.num_features) # 归一化操作, 默认 LayerNorm

# 在深层特征提取网络中加入卷积块, 保持特征图通道数不变
if resi_connection == '1conv': # 1 层卷积
    self.conv_after_body = nn.Conv2d(embed_dim, embed_dim, 3, 1, 1)
elif resi_connection == '3conv': # 3 层卷积

```

```

# 为了减少参数使用和节约显存, 采用瓶颈结构
self.conv_after_body = nn.Sequential(nn.Conv2d(embed_dim, embed_dim // 4, 3, 1, 1), # 降维
                                     nn.LeakyReLU(negative_slope=0.2, inplace=True),
                                     nn.Conv2d(embed_dim // 4, embed_dim // 4, 1, 1, 0),
                                     nn.LeakyReLU(negative_slope=0.2, inplace=True),
                                     nn.Conv2d(embed_dim // 4, embed_dim, 3, 1, 1)) # 升维

# 高质量图像重建模块
if self.upsampler == 'pixelshuffle': # pixelshuffle 上采样
    # 适合经典超分
    self.conv_before_upsample = nn.Sequential(nn.Conv2d(embed_dim, num_feat, 3, 1, 1),
                                              nn.LeakyReLU(inplace=True))

    self.upsample = Upsample(upscale, num_feat) # 上采样
    self.conv_last = nn.Conv2d(num_feat, num_out_ch, 3, 1, 1) # 输出卷积层
elif self.upsampler == 'pixelshuffledirect': # 一步是实现既上采样也降维
    # 适合轻量级充分, 可以减少参数量(一步是实现既上采样也降维)
    self.upsample = UpsampleOneStep(upscale, embed_dim, num_out_ch,
                                    (patches_resolution[0], patches_resolution[1]))
elif self.upsampler == 'nearest+conv': # 最近邻插值上采样
    # 适合真实图像超分
    assert self.upscale == 4, 'only support x4 now.' # 声明目前仅支持 4 倍超分重建
    # 上采样之前的卷积层
    self.conv_before_upsample = nn.Sequential(nn.Conv2d(embed_dim, num_feat, 3, 1, 1),
                                              nn.LeakyReLU(inplace=True))

    # 第一次上采样卷积(直接对输入做最近邻插值变为 2 倍图像)
    self.conv_up1 = nn.Conv2d(num_feat, num_feat, 3, 1, 1)
    # 第二次上采样卷积(直接对输入做最近邻插值变为 2 倍图像)
    self.conv_up2 = nn.Conv2d(num_feat, num_feat, 3, 1, 1)
    self.conv_hr = nn.Conv2d(num_feat, num_feat, 3, 1, 1) # 对上采样完成的图像再做卷积
    self.lrelu = nn.LeakyReLU(negative_slope=0.2, inplace=True) # 激活层
    self.conv_last = nn.Conv2d(num_feat, num_out_ch, 3, 1, 1) # 输出卷积层
else:
    # 适合图像去噪和 JPEG 压缩去伪影
    self.conv_last = nn.Conv2d(embed_dim, num_out_ch, 3, 1, 1)

self.apply(self._init_weights) # 初始化网络参数

# 初始化网络参数
def _init_weights(self, m):
    if isinstance(m, nn.Linear): # 判断是否为线性 Linear 层
        trunc_normal_(m.weight, std=.02) # 截断正态分布, 限制标准差为 0.02
        if m.bias is not None: # 如果设置了偏置
            nn.init.constant_(m.bias, 0) # 初始化偏置为 0
    elif isinstance(m, nn.LayerNorm): # 判断是否为归一化 LayerNorm 层

```

```

nn.init.constant_(m.bias, 0) # 初始化偏置为 0
nn.init.constant_(m.weight, 1.0) # 初始化权重系数为 1

# 检查图片(准确说是张量)的大小
def check_image_size(self, x):
    _, h, w = x.size() # 张量 x 的高和宽
    # h 维度要填充的个数
    mod_pad_h = (self.window_size - h % self.window_size) % self.window_size
    # w 维度要填充的个数
    mod_pad_w = (self.window_size - w % self.window_size) % self.window_size
    # 右填充 mod_pad_w 个值, 下填充 mod_pad_h 个值, 模式为反射(可以理解为以 x 的维度末尾为轴对折)
    x = F.pad(x, (0, mod_pad_w, 0, mod_pad_h), 'reflect')
    return x

# 深层特征提取网络的前向传播
def forward_features(self, x):
    x_size = (x.shape[2], x.shape[3]) # 张量 x 的高和宽
    x = self.patch_embed(x) # 分割 x 为多个不重叠的 patch embeddings
    if self.ape: # 绝对位置 embedding
        x = x + self.absolute_pos_embed # x 加上对应的绝对位置 embedding
    x = self.pos_drop(x) # 随机将 x 中的部分元素置 0

    for layer in self.layers:
        x = layer(x, x_size) # x 通过多个串联的 RSTB

    x = self.norm(x) # 对 RSTB 的输出进行归一化
    x = self.patch_unembed(x, x_size) # 将多个不重叠的 patch 合并成图像

    return x

# SWinIR 的前向传播
def forward(self, x):
    H, W = x.shape[2:] # 输入图片的高和宽
    x = self.check_image_size(x) # 检查图片的大小, 使高宽满足 window_size 的整数倍

    self.mean = self.mean.type_as(x) # RGB 均值的类型同 x 一致
    x = (x - self.mean) * self.img_range # x 减去 RGB 均值再乘以输入的最大灰度值

    if self.upsampler == 'pixelshuffle': # pixelshuffle 上采样方法
        # 适合经典超分
        x = self.conv_first(x) # 输入卷积层
        x = self.conv_after_body(self.forward_features(x)) + x # 深度特征提取网络, 引入残差
        x = self.conv_before_upsample(x) # 上采样前进行卷积
        x = self.conv_last(self.upsample(x)) # 上采样后再通过输出卷积层

```

```

elif self.upsampler == 'pixelshuffledirect': # 一步是实现既上采样也降维
    # 适合轻量级超分
    x = self.conv_first(x) # 输入卷积层
    x = self.conv_after_body(self.forward_features(x)) + x # 深度特征提取网络，引入残差
    x = self.upsample(x) # 上采样并降维后输出

elif self.upsampler == 'nearest+conv': # 最近邻插值上采样方法
    # 适合真实图像超分，只适合 4 倍超分
    x = self.conv_first(x) # 输入卷积层
    x = self.conv_after_body(self.forward_features(x)) + x # 深度特征提取网络，引入残差
    x = self.conv_before_upsample(x) # 上采样前进行卷积
    # 第一次上采样 2 倍
    x = self.lrelu(self.conv_up1(torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest'))))
    # 第二次上采样 2 倍
    x = self.lrelu(self.conv_up2(torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest'))))
    x = self.conv_last(self.lrelu(self.conv_hr(x))) # 输出卷积层

else:
    # 适合图像去噪和 JPEG 压缩去伪影
    x_first = self.conv_first(x) # 输入卷积层
    res = self.conv_after_body(self.forward_features(x_first)) + x_first # 深度特征提取网络，引入残差
    x = x + self.conv_last(res) # 输出卷积层，引入残差

x = x / self.img_range + self.mean # 最后的 x 除以灰度值范围再加上 RGB 均值

return x[:, :, :H*self.upscale, :W*self.upscale] # 返回输出 x

```

3.5.2 MLP

```

# 多层感知机
class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU, drop=0.):
        super().__init__()
        out_features = out_features or in_features # 输入特征的维度
        hidden_features = hidden_features or in_features # 隐藏特征维度
        self.fc1 = nn.Linear(in_features, hidden_features) # 线性层
        self.act = act_layer() # 激活函数
        self.fc2 = nn.Linear(hidden_features, out_features) # 线性层
        self.drop = nn.Dropout(drop) # 随机丢弃神经元，丢弃率默认为 0

    # 定义前向传播
    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)

```

```

x = self.drop(x)
x = self.fc2(x)
x = self.drop(x)
return x

```

3.5.3 Patch Embedding

主要的操作就是将原始 2 维图像 (特征图的一个 plane 或者说一个 channel) 转变为 1 维的 patch embeddings, 通过 Swin Transformer 学习处理之后再重新组合成与原来特征图结构一致的新特征图。

3.5.3.1 将 2 维图像转变成 1 维 patch embeddings

```

# 图像转成 Patch Embeddings
class PatchEmbed(nn.Module):
    """ Image to Patch Embedding

    输入:
        img_size (int): 图像的大小, 默认为 224*224.
        patch_size (int): Patch token 的大小, 默认为 4*4.
        in_chans (int): 输入图像的通道数, 默认为 3.
        embed_dim (int): 线性 projection 输出的通道数, 默认为 96.
        norm_layer (nn.Module, optional): 归一化层, 默认为 None.
    """

    def __init__(self, img_size=224, patch_size=4, in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        self.img_size = to_2tuple(img_size) # 图像的大小, 默认为 224*224
        self.patch_size = to_2tuple(patch_size) # Patch token 的大小, 默认为 4*4
        self.patches_resolution = [img_size[0] // patch_size[0], img_size[1] // patch_size[1]] # patch 的分辨率
        self.num_patches = self.patches_resolution[0] * self.patches_resolution[1] # patch 的个数, num_patches

        self.in_chans = in_chans # 输入图像的通道数
        self.embed_dim = embed_dim # 线性 projection 输出的通道数

        if norm_layer is not None:
            self.norm = norm_layer(embed_dim) # 归一化
        else:
            self.norm = None

```

```

# 定义前向传播
def forward(self, x):
    x = x.flatten(2).transpose(1, 2) # 结构为 [B, num_patches, C]
    if self.norm is not None:
        x = self.norm(x) # 归一化
    return x

```

3.5.3.2 将 1 维 patch embeddings 转变为 2 维图像

```

# 从 Patch Embeddings 组合图像
class PatchUnEmbed(nn.Module):
    r""" Image to Patch Unembedding

    输入:

        img_size (int): 图像的大小, 默认为 224*224.
        patch_size (int): Patch token 的大小, 默认为 4*4.
        in_chans (int): 输入图像的通道数, 默认为 3.
        embed_dim (int): 线性 projection 输出的通道数, 默认为 96.
        norm_layer (nn.Module, optional): 归一化层, 默认为 None.

    """

    def __init__(self, img_size=224, patch_size=4, in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        self.img_size = to_2tuple(img_size) # 图像的大小, 默认为 224*224
        self.patch_size = to_2tuple(patch_size) # Patch token 的大小, 默认为 4*4
        self.patches_resolution = [img_size[0] // patch_size[0], img_size[1] // patch_size[1]] # patch 的分辨率
        self.in_chans = in_chans # 输入图像的通道数
        self.embed_dim = embed_dim # 线性 projection 输出的通道数

    def forward(self, x, x_size):
        B, HW, C = x.shape # 输入 x 的结构
        x = x.transpose(1, 2).view(B, self.embed_dim, x_size[0], x_size[1]) # 输出结构为 [B, Ph*Pw, C]
        return x

```

3.5.4 Window Attention

采用窗口注意力来减轻传统 Transformer 的全局注意力带来的计算负担，将注意力的计算限制在每一个窗口里，在每个窗口里其实还是原始的多头自注意力。

3.5.4.1 窗口分割

```
# 将输入分割为多个不重叠窗口
def window_partition(x, window_size):
    """
    输入:
        x: (B, H, W, C)
        window_size (int): window size # 窗口的大小
    返回:
        windows: (num_windows*B, window_size, window_size, C) # 每一个 batch 有单独的 windows
    """
    B, H, W, C = x.shape # 输入的 batch 个数, 高, 宽, 通道数
    # 将输入 x 重构为结构 [batch 个数, 高方向的窗口个数, 窗口大小, 宽方向的窗口个数, 窗口大小, 通道数] 的张量
    x = x.view(B, H // window_size, window_size, W // window_size, window_size, C)
    # 交换重构后 x 的第 3 和 4 维度, 5 和 6 维度, 再次重构为结构 [高和宽方向的窗口个数乘以 batch 个数, 窗口大小, 窗口大小, 通道数] 的张量
    windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window_size, window_size, C)
    return windows
```

3.5.4.2 窗口注意力

```
# 窗口注意力
class WindowAttention(nn.Module):
    r""" 基于有相对位置偏差的多头自注意力窗口, 支持移位的(shifted)或者不移位的(non-shifted)窗口.

    输入:
        dim (int): 输入特征的维度.
        window_size (tuple[int]): 窗口的大小.
        num_heads (int): 注意力头的个数.
        qkv_bias (bool, optional): 给 query, key, value 添加可学习的偏置, 默认为 True.
        qk_scale (float | None, optional): 重写默认的缩放因子 scale.
        attn_drop (float, optional): 注意力权重的丢弃率, 默认为 0.0.
        proj_drop (float, optional): 输出的丢弃率, 默认为 0.0.
    """

    def __init__(self, dim, window_size, num_heads, qkv_bias=True, qk_scale=None, attn_drop=0., proj_drop=0.):
```

```

super().__init__()
self.dim = dim # 输入特征的维度
self.window_size = window_size # 窗口的高 Wh,宽 Ww
self.num_heads = num_heads # 注意力头的个数
head_dim = dim // num_heads # 注意力头的维度
self.scale = qk_scale or head_dim ** -0.5 # 缩放因子 scale

# 定义相对位置偏移的参数表, 结构为 [2*Wh-1 * 2*Ww-1, num_heads]
self.relative_position_bias_table = nn.Parameter(
    torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1), num_heads))

# 获取窗口内每个 token 的成对的相对位置索引
coords_h = torch.arange(self.window_size[0]) # 高维度上的坐标 (0, 7)
coords_w = torch.arange(self.window_size[1]) # 宽维度上的坐标 (0, 7)
coords = torch.stack(torch.meshgrid([coords_h, coords_w])) # 坐标, 结构为 [2, Wh, Ww]
coords_flatten = torch.flatten(coords, 1) # 重构张量结构为 [2, Wh*Ww]
relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :] # 相对坐标, 结构为 [2, Wh*Ww, Wh*Ww]
relative_coords = relative_coords.permute(1, 2, 0).contiguous() # 交换维度, 结构为 [Wh*Ww, Wh*Ww, 2]
relative_coords[:, :, 0] += self.window_size[0] - 1 # 第 1 个维度移位
relative_coords[:, :, 1] += self.window_size[1] - 1 # 第 1 个维度移位
relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1 # 第 1 个维度的值乘以 2 倍的 Ww, 再减 1
relative_position_index = relative_coords.sum(-1) # 相对位置索引, 结构为 [Wh*Ww, Wh*Ww]
self.register_buffer("relative_position_index", relative_position_index) # 保存数据, 不再更新

self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias) # 线性层, 特征维度变为原来的 3 倍
self.attn_drop = nn.Dropout(attn_drop) # 随机丢弃神经元, 丢弃率默认为 0.0
self.proj = nn.Linear(dim, dim) # 线性层, 特征维度不变

self.proj_drop = nn.Dropout(proj_drop) # 随机丢弃神经元, 丢弃率默认为 0.0

trunc_normal_(self.relative_position_bias_table, std=.02) # 截断正态分布, 限制标准差为 0.02
self.softmax = nn.Softmax(dim=-1) # 激活函数 softmax

# 定义前向传播
def forward(self, x, mask=None):
    """
    输入:
        x: 输入特征图, 结构为 [num_windows*B, N, C]
        mask: (0/-inf) mask, 结构为 [num_windows, Wh*Ww, Wh*Ww] 或者没有 mask
    """
    B_, N, C = x.shape # 输入特征图的结构
    # 将特征图的通道维度按照注意力头的个数重新划分, 并再做交换维度操作
    qkv = self.qkv(x).reshape(B_, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)

```



```

q, k, v = qkv[0], qkv[1], qkv[2] # 方便后续写代码, 重新赋值

# q 乘以缩放因子
q = q * self.scale

# @ 代表常规意义上的矩阵相乘
attn = (q @ k.transpose(-2, -1)) # q 和 k 相乘后并交换最后两个维度

# 相对位置偏移, 结构为 [Wh*Ww, Wh*Ww, num_heads]
relative_position_bias = self.relative_position_bias_table[self.relative_position_index.view(-1)].view(
    self.window_size[0] * self.window_size[1], self.window_size[0] * self.window_size[1], -1)
# 相对位置偏移交换维度, 结构为 [num_heads, Wh*Ww, Wh*Ww]
relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()
attn = attn + relative_position_bias.unsqueeze(0) # 带相对位置偏移的注意力图

if mask is not None: # 判断是否有 mask
    nW = mask.shape[0] # mask 的宽
    # 注意力图与 mask 相加
    attn = attn.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
    attn = attn.view(-1, self.num_heads, N, N) # 恢复注意力图原来的结构
    attn = self.softmax(attn) # 激活注意力图 [0, 1] 之间
else:
    attn = self.softmax(attn)

attn = self.attn_drop(attn) # 随机设置注意力图中的部分值为 0
# 注意力图与 v 相乘得到新的注意力图
x = (attn @ v).transpose(1, 2).reshape(B_, N, C)
x = self.proj(x) # 通过线性层
x = self.proj_drop(x) # 随机设置新注意力图中的部分值为 0
return x

```

3.5.4.3 窗口合并

```

# 将多个不重叠窗口重新合并
def window_reverse(windows, window_size, H, W):
    """
    输入:
        windows: (num_windows*B, window_size, window_size, C) # 分割得到的窗口(已处理)
        window_size (int): Window size # 窗口大小
        H (int): Height of image # 原分割窗口前特征图的高
        W (int): Width of image # 原分割窗口前特征图的宽
    返回:
        x: (B, H, W, C) # 返回与分割前特征图结构一样的结果
    """
    # 以下就是分割窗口的逆向操作, 不多解释

```

```

B = int(windows.shape[0] / (H * W / window_size / window_size))
x = windows.view(B, H // window_size, W // window_size, window_size, window_size, -1)
x = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(B, H, W, -1)
return x

```

3.5.5 残差 Swin Transformer 块 (RSTB)

SwinIR 主要是使用 Swin Transformer 的思想来实现, 残差 Swin Transformer 块 (RSTB) 可以理解为:

- (1) Swin Transformer 块是 RSTB 的基础组件;
- (2) 多个 Swin Transformer 块构成基础网络;
- (3) 基础网络结尾处加上卷积操作后再引入残差构成 RSTB。

3.5.5.1 Swin Transformer 块

```

# Swin Transformer 块
class SwinTransformerBlock(nn.Module):
    """
    输入:
        dim (int): 输入特征的维度.
        input_resolution (tuple[int]): 输入特征图的分辨率.
        num_heads (int): 注意力头的个数.
        window_size (int): 窗口的大小.
        shift_size (int): SW-MSA 的移位值.
        mlp_ratio (float): 多层感知机隐藏层的维度和嵌入层的比.
        qkv_bias (bool, optional): 给 query, key, value 添加一个可学习偏置, 默认为 True.
        qk_scale (float | None, optional): 重写默认的缩放因子 scale.
        drop (float, optional): 随机神经元丢弃率, 默认为 0.0.
        attn_drop (float, optional): 注意力图随机丢弃率, 默认为 0.0.
        drop_path (float, optional): 深度随机丢弃率, 默认为 0.0.
        act_layer (nn.Module, optional): 激活函数, 默认为 nn.GELU.
        norm_layer (nn.Module, optional): 归一化操作, 默认为 nn.LayerNorm.
    """

    def __init__(self, dim, input_resolution, num_heads, window_size=7, shift_size=0,
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0., drop_path=0.,
                 act_layer=nn.GELU, norm_layer=nn.LayerNorm):
        super().__init__()
        self.dim = dim # 输入特征的维度
        self.input_resolution = input_resolution # 输入特征图的分辨率
        self.num_heads = num_heads # 注意力头的个数
        self.window_size = window_size # 窗口的大小

```

```

self.shift_size = shift_size # SW-MSA 的移位大小
self.mlp_ratio = mlp_ratio # 多层感知机隐藏层的维度和嵌入层的比
if min(self.input_resolution) <= self.window_size: # 如果输入分辨率小于等于窗口大小
    self.shift_size = 0 # 移位大小为 0
    self.window_size = min(self.input_resolution) # 窗口大小等于输入分辨率大小
# 断言移位值必须小于等于窗口的大小
assert 0 <= self.shift_size < self.window_size, "shift_size must in 0-window_size"

self.norm1 = norm_layer(dim) # 归一化层
# 窗口注意力
self.attn = WindowAttention(
    dim, window_size=to_2tuple(self.window_size), num_heads=num_heads,
    qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=drop)

# 如果丢弃率大于 0 则进行随机丢弃, 否则进行占位(不做任何操作)
self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
self.norm2 = norm_layer(dim) # 归一化层
mlp_hidden_dim = int(dim * mlp_ratio) # 多层感知机隐藏层维度
# 多层感知机
self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

if self.shift_size > 0: # 如果移位值大于 0
    attn_mask = self.calculate_mask(self.input_resolution) # 计算注意力 mask
else:
    attn_mask = None # 注意力 mask 赋空

self.register_buffer("attn_mask", attn_mask) # 保存注意力 mask, 不参与更新

# 计算注意力 mask
def calculate_mask(self, x_size):
    H, W = x_size # 特征图的高宽
    img_mask = torch.zeros((1, H, W, 1)) # 新建张量, 结构为 [1, H, W, 1]
    # 以下两 slices 中的数据是索引, 具体缘由尚未搞懂
    h_slices = (slice(0, -self.window_size), # 索引 0 到索引倒数第 window_size
                slice(-self.window_size, -self.shift_size), # 索引倒数第 window_size 到索引倒数第 shift_size
                slice(-self.shift_size, None)) # 索引倒数第 shift_size 后所有索引
    w_slices = (slice(0, -self.window_size),
                slice(-self.window_size, -self.shift_size),
                slice(-self.shift_size, None))

    cnt = 0
    for h in h_slices:
        for w in w_slices:
            img_mask[:, h, w, :] = cnt # 将 img_mask 中 h, w 对应索引范围的值置为 cnt
            cnt += 1 # 加 1

```

```

        mask_windows = window_partition(img_mask, self.window_size) # 窗口分割, 返回值结构为 [nW, window_size,
window_size, 1]

        mask_windows = mask_windows.view(-1, self.window_size * self.window_size) # 重构结构为二维张量, 列数为
>window_size*window_size]

        attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2) # 增加第 2 维度减去增加第 3 维度的
注意力 mask

        # 用浮点数 -100. 填充注意力 mask 中值不为 0 的元素, 再用浮点数 0. 填充注意力 mask 中值为 0 的元素
        attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).masked_fill(attn_mask == 0, float(0.0))

        return attn_mask

# 定义前向传播
def forward(self, x, x_size):
    H, W = x_size # 输入特征图的分辨率
    B, L, C = x.shape # 输入特征的 batch 个数, 长度和维度
    # assert L == H * W, "input feature has wrong size"

    shortcut = x
    x = self.norm1(x) # 归一化
    x = x.view(B, H, W, C) # 重构 x 为结构 [B, H, W, C]

    # 循环移位
    if self.shift_size > 0: # 如果移位值大于 0
        # 第 0 维度上移 shift_size 位, 第 1 维度左移 shift_size 位
        shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
    else:
        shifted_x = x # 不移位

    # 对移位操作得到的特征图分割窗口, nW 是窗口的个数
    x_windows = window_partition(shifted_x, self.window_size) # 结构为 [nW*B, window_size, window_size, C]
    x_windows = x_windows.view(-1, self.window_size * self.window_size, C) # 结构为 [nW*B,
>window_size*window_size, C]

    # W-MSA/SW-MSA, 用在分辨率是窗口大小的整数倍的图像上进行测试
    if self.input_resolution == x_size: # 输入分辨率与设定一致, 不需要重新计算注意力 mask
        attn_windows = self.attn(x_windows, mask=self.attn_mask) # 注意力窗口, 结构为 [nW*B,
>window_size*window_size, C]
    else: # 输入分辨率与设定不一致, 需要重新计算注意力 mask
        attn_windows = self.attn(x_windows, mask=self.calculate_mask(x_size).to(x.device))

    # 合并窗口
    attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C) # 结构为 [-1, window_size,
>window_size, C]

```

```

shifted_x = window_reverse(attn_windows, self.window_size, H, W) # 结构为 [B, H', W', C]

# 逆向循环移位
if self.shift_size > 0:
    # 第 0 维度下移 shift_size 位, 第 1 维度右移 shift_size 位
    x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
else:
    x = shifted_x # 不逆向移位
x = x.view(B, H * W, C) # 结构为 [B, H*W, C]

# FFN
x = shortcut + self.drop_path(x) # 对 x 做 dropout, 引入残差
x = x + self.drop_path(self.mlp(self.norm2(x))) # 归一化后通过 MLP, 再做 dropout, 引入残差

return x

```

3.5.5.2 基础网络

```

# 单阶段的 SWin Transformer 基础层
class BasicLayer(nn.Module):
    """
    输入:
        dim (int): 输入特征的维度.
        input_resolution (tuple[int]): 输入分辨率.
        depth (int): SWin Transformer 块的个数.
        num_heads (int): 注意力头的个数.
        window_size (int): 本地(当前块中)窗口的大小.
        mlp_ratio (float): MLP 隐藏层特征维度与嵌入层特征维度的比.
        qkv_bias (bool, optional): 给 query, key, value 添加一个可学习偏置, 默认为 True.
        qk_scale (float | None, optional): 重写默认的缩放因子 scale.
        drop (float, optional): 随机丢弃神经元, 丢弃率默认为 0.0.
        attn_drop (float, optional): 注意力图随机丢弃率, 默认为 0.0.
        drop_path (float | tuple[float], optional): 深度随机丢弃率, 默认为 0.0.
        norm_layer (nn.Module, optional): 归一化操作, 默认为 nn.LayerNorm.
        downsample (nn.Module | None, optional): 结尾处的下采样层, 默认没有.
        use_checkpoint (bool): 是否使用 checkpointing 来节省显存, 默认为 False.
    """

    def __init__(self, dim, input_resolution, depth, num_heads, window_size,
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., norm_layer=nn.LayerNorm, downsample=None, use_checkpoint=False):

        super().__init__()

```

```

self.dim = dim # 输入特征的维度
self.input_resolution = input_resolution # 输入分辨率
self.depth = depth # SWin Transformer 块的个数
self.use_checkpoint = use_checkpoint # 是否使用 checkpointing 来节省显存, 默认为 False

# 创建 Swin Transformer 网络
self.blocks = nn.ModuleList([
    SwinTransformerBlock(dim=dim, input_resolution=input_resolution,
                          num_heads=num_heads, window_size=window_size,
                          shift_size=0 if (i % 2 == 0) else window_size // 2,
                          mlp_ratio=mlp_ratio,
                          qkv_bias=qkv_bias, qk_scale=qk_scale,
                          drop=drop, attn_drop=attn_drop,
                          drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
                          norm_layer=norm_layer)
    for i in range(depth)])

# patch 合并层
if downsample is not None: # 如果有下采样
    self.downsample = downsample(input_resolution, dim=dim, norm_layer=norm_layer) # 下采样
else:
    self.downsample = None # 不做下采样

#定义前向传播
def forward(self, x, x_size):
    for blk in self.blocks: # x 输入串联的 Swin Transformer 块
        if self.use_checkpoint:
            x = checkpoint.checkpoint(blk, x, x_size) # 使用 checkpoint
        else:
            x = blk(x, x_size) # 直接输入网络
    if self.downsample is not None:
        x = self.downsample(x) # 下采样
    return x

```

3.5.5.3 残差 Swin Transformer 块 (RSTB)

```

# 残差 Swin Transformer 块 (RSTB)
class RSTB(nn.Module):
    """
    输入:
        dim (int): 输入特征的维度.
        input_resolution (tuple[int]): 输入分辨率.
        depth (int): SWin Transformer 块的个数.
    """

```

```

num_heads (int): 注意力头的个数.
window_size (int): 本地(当前块中)窗口的大小.
mlp_ratio (float): MLP 隐藏层特征维度与嵌入层特征维度的比.
qkv_bias (bool, optional): 给 query, key, value 添加一个可学习偏置, 默认为 True.
qk_scale (float | None, optional): 重写默认的缩放因子 scale.
drop (float, optional): D 随机丢弃神经元, 丢弃率默认为 0.0.
attn_drop (float, optional): 注意力图随机丢弃率, 默认为 0.0.
drop_path (float | tuple[float], optional): 深度随机丢弃率, 默认为 0.0.
norm_layer (nn.Module, optional): 归一化操作, 默认为 nn.LayerNorm.
downsample (nn.Module | None, optional): 结尾处的下采样层, 默认没有.
use_checkpoint (bool): 是否使用 checkpointing 来节省显存, 默认为 False.
img_size: 输入图片的大小.
patch_size: Patch 的大小.
resi_connection: 残差连接之前的卷积块.
"""

def __init__(self, dim, input_resolution, depth, num_heads, window_size,
             mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
             drop_path=0., norm_layer=nn.LayerNorm, downsample=None, use_checkpoint=False,
             img_size=224, patch_size=4, resi_connection='1conv'):
    super(RSTB, self).__init__()

    self.dim = dim # 输入特征的维度
    self.input_resolution = input_resolution # 输入分辨率

    # SWin Transformer 基础层
    self.residual_group = BasicLayer(dim=dim,
                                     input_resolution=input_resolution,
                                     depth=depth,
                                     num_heads=num_heads,
                                     window_size=window_size,
                                     mlp_ratio=mlp_ratio,
                                     qkv_bias=qkv_bias, qk_scale=qk_scale,
                                     drop=drop, attn_drop=attn_drop,
                                     drop_path=drop_path,
                                     norm_layer=norm_layer,
                                     downsample=downsample,
                                     use_checkpoint=use_checkpoint)

    if resi_connection == '1conv': # 结尾用 1 个卷积层
        self.conv = nn.Conv2d(dim, dim, 3, 1, 1)
    elif resi_connection == '3conv': # 结尾用 3 个卷积层
        # 为了减少参数使用和节约显存, 采用瓶颈结构
        self.conv = nn.Sequential(nn.Conv2d(dim, dim // 4, 3, 1, 1), nn.LeakyReLU(negative_slope=0.2, inplace=True),

```

```

nn.Conv2d(dim // 4, dim // 4, 1, 1, 0),
nn.LeakyReLU(negative_slope=0.2, inplace=True),
nn.Conv2d(dim // 4, dim, 3, 1, 1))

# 图像转成 Patch Embeddings
self.patch_embed = PatchEmbed(
    img_size=img_size, patch_size=patch_size, in_chans=0, embed_dim=dim,
    norm_layer=None)

# 从 Patch Embeddings 组合图像
self.patch_unembed = PatchUnEmbed(
    img_size=img_size, patch_size=patch_size, in_chans=0, embed_dim=dim,
    norm_layer=None)

# 定义前向传播
def forward(self, x, x_size):
    return self.patch_embed(self.conv(self.patch_unembed(self.residual_group(x, x_size), x_size))) + x # 引入残差

```

3.5.6 HQ Image Reconstruction

高质量图像重建模块其实就是卷积和上采样操作的组合

3.5.6.1

先卷积再使用 `pixelshuffle` 上采样，特征图维度不是 3

```

# 上采样
class Upsample(nn.Sequential):
    """
    输入:
        scale (int): 缩放因子, 支持 2^n and 3.
        num_feat (int): 中间特征的通道数.
    """

    def __init__(self, scale, num_feat):
        m = []
        if (scale & (scale - 1)) == 0: # 缩放因子等于 2^n
            for _ in range(int(math.log(scale, 2))): # 循环 n 次
                m.append(nn.Conv2d(num_feat, 4 * num_feat, 3, 1, 1)) # 卷积层
                m.append(nn.PixelShuffle(2)) # pixelshuffle 上采样 2 倍
            elif scale == 3: # 缩放因子等于 3
                m.append(nn.Conv2d(num_feat, 9 * num_feat, 3, 1, 1)) # 卷积层
                m.append(nn.PixelShuffle(3)) # pixelshuffle 上采样 3 倍

```



```

else:
    # 报错，缩放因子不对
    raise ValueError(f'scale {scale} is not supported. ' 'Supported scales: 2^n and 3.')
super(Upsample, self).__init__(*m)

```

3.5.6.2

一步既上采样也实现输出降维，特征图维度是 3，即最后的恢复图像

```

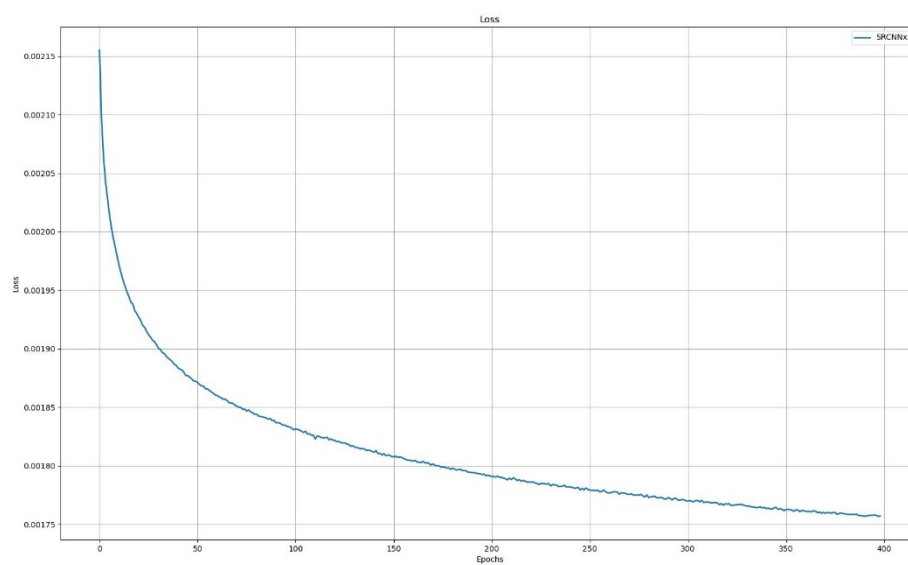
# 一步实现既上采样也降维
class UpsampleOneStep(nn.Sequential):
    """一步上采样与前边上采样模块不同之处在于该模块只有一个卷积层和一个 pixelshuffle 层

    输入:
        scale (int): 缩放因子，支持 2^n and 3.
        num_feat (int): 中间特征的通道数.
    """

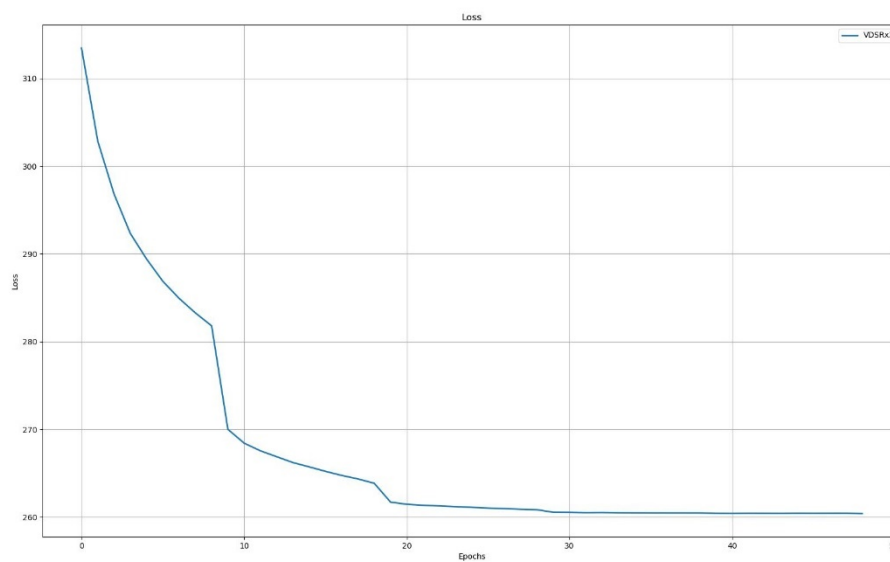
    def __init__(self, scale, num_feat, num_out_ch, input_resolution=None):
        self.num_feat = num_feat # 中间特征的通道数
        self.input_resolution = input_resolution # 输入分辨率
        m = []
        m.append(nn.Conv2d(num_feat, (scale ** 2) * num_out_ch, 3, 1, 1)) # 卷积层
        m.append(nn.PixelShuffle(scale)) # pixelshuffle 上采样 scale 倍
        super(UpsampleOneStep, self).__init__(*m)

```

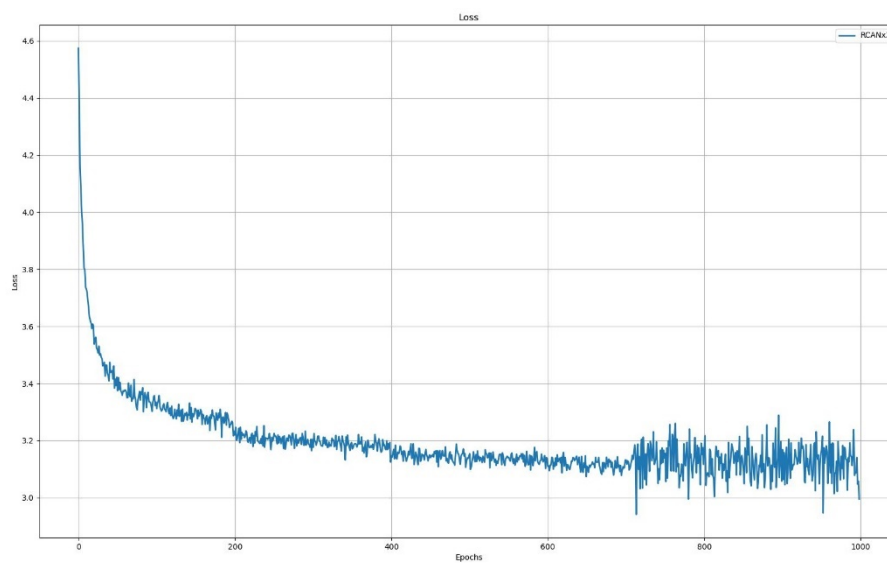
4. 训练曲线 (Loss-Epoch)

SRCNN:

SRCNN 的 loss-epoch 图

VDSR:

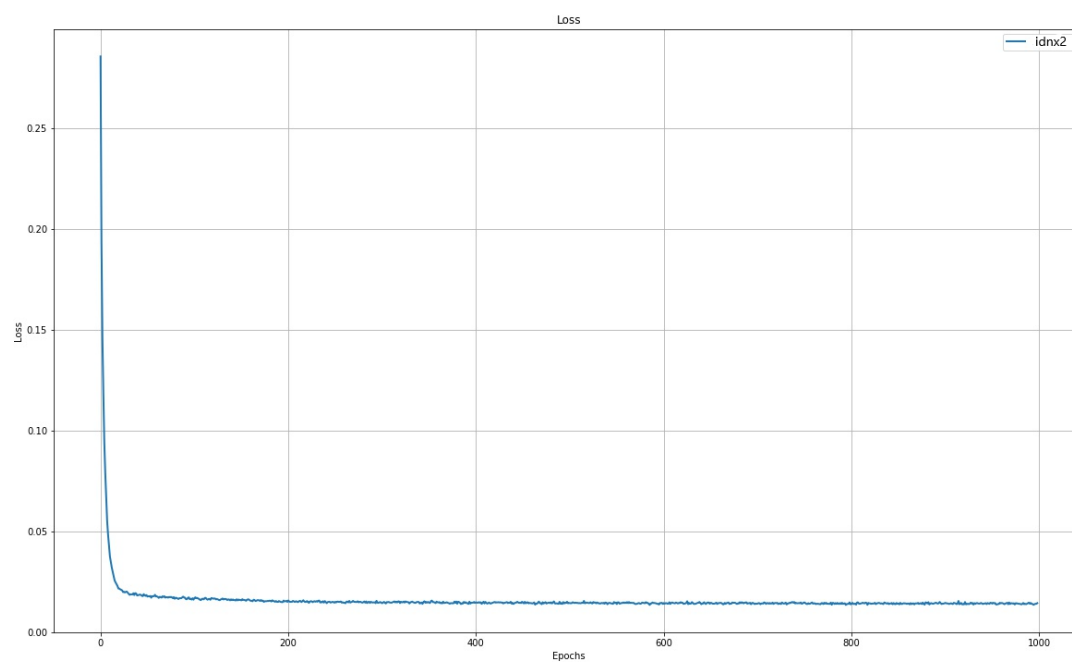
VDSR 的 loss-epoch 图

RCAN:

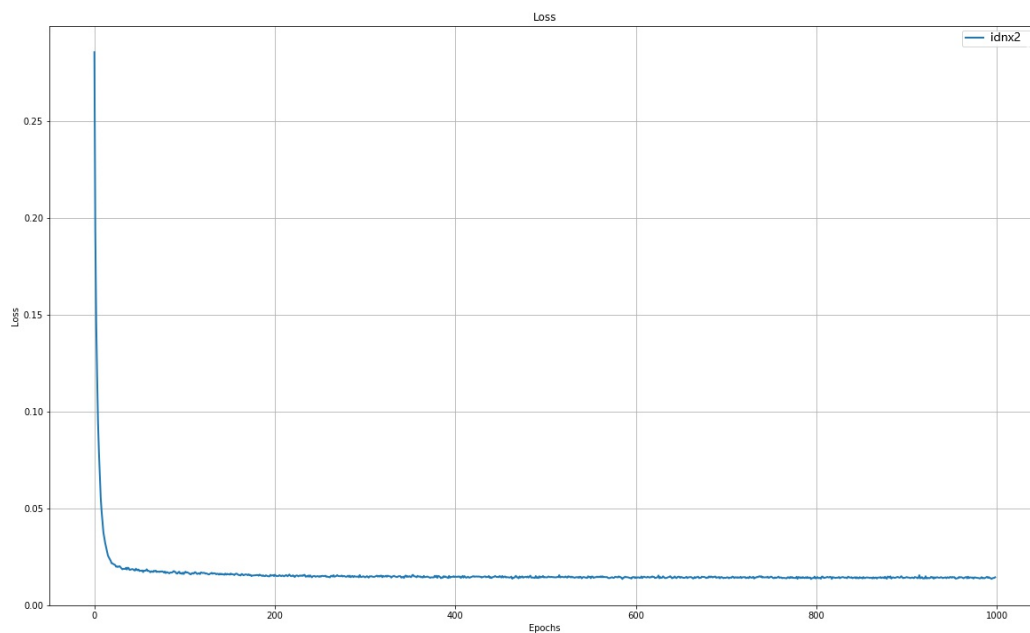
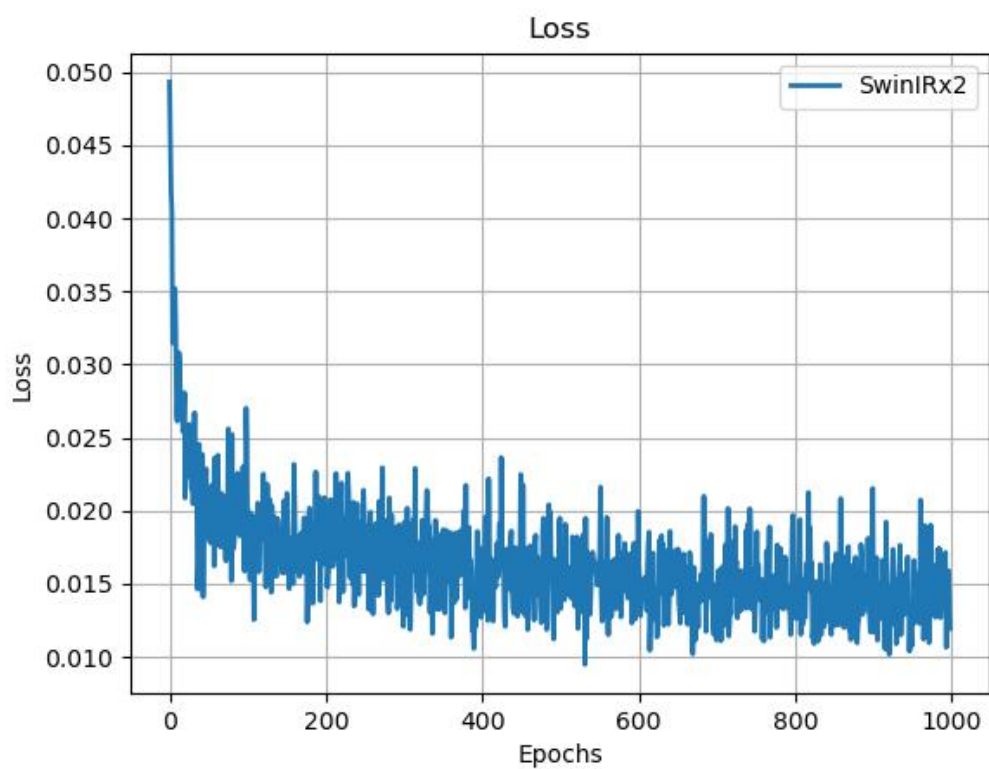
RCAN 的 loss-epoch 图

IDN

IDN 采用两阶段的训练第一阶段采用 L1 损失函数，它的训练曲线：



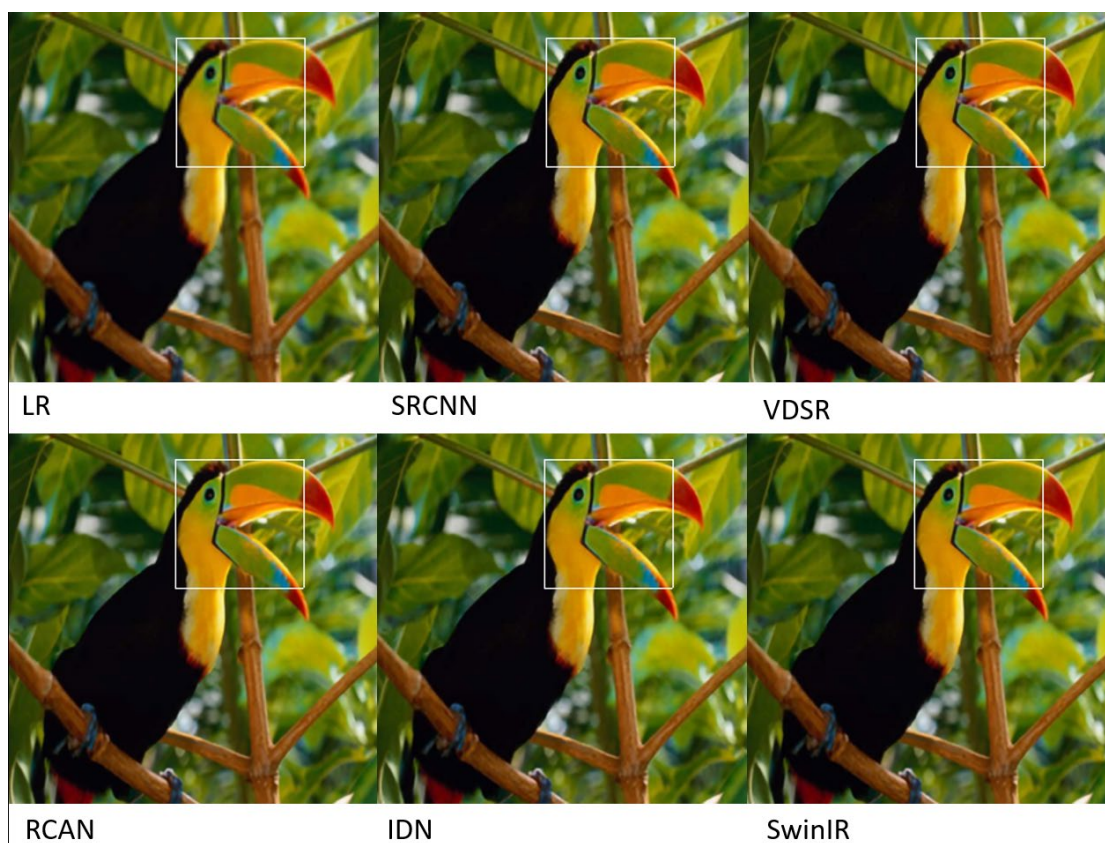
第二阶段采用 L2 损失函数进行微调，它的训练曲线：

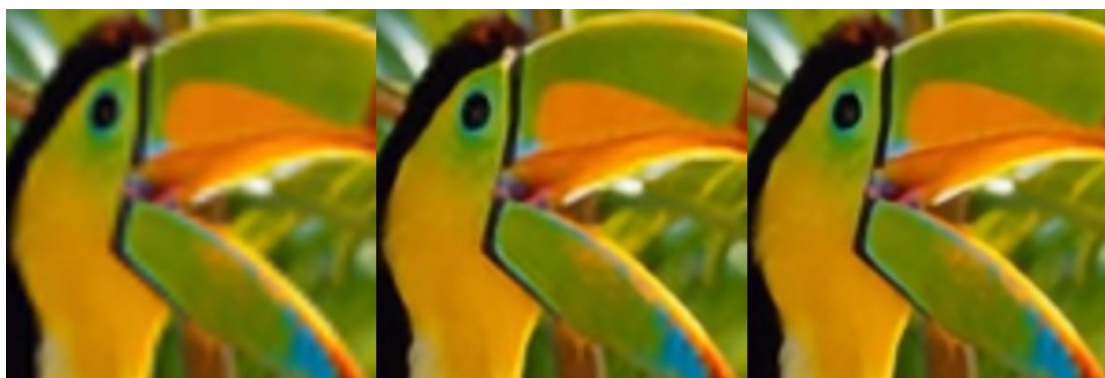
**SwinIR:**

SwinIR 的 loss-epoch 图

5. 运行或运行结果截图

5.1 模型效果图

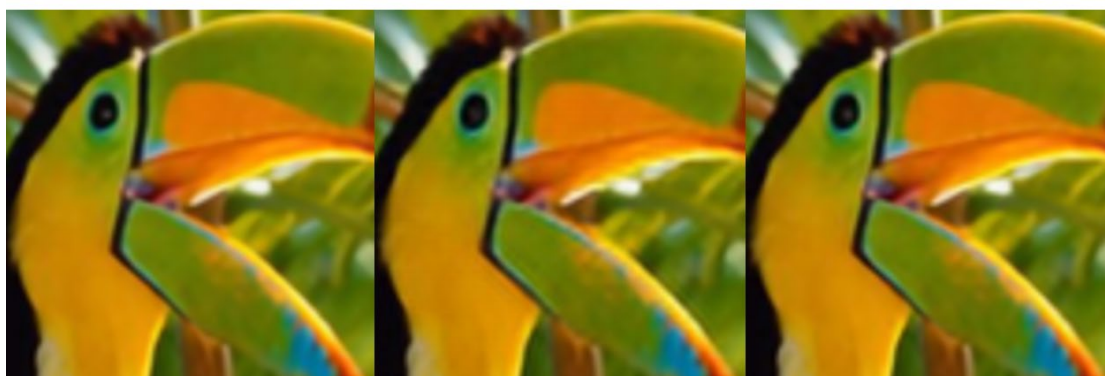




LR

SRCNN

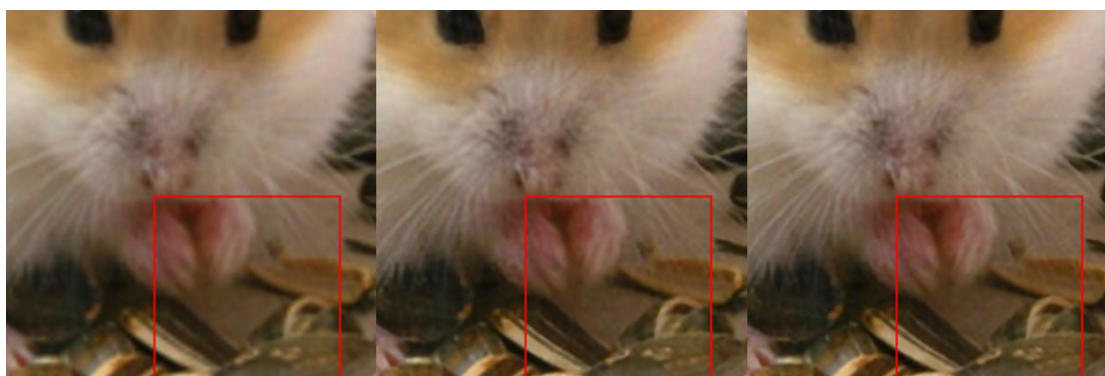
VDSR



RCAN

IDN

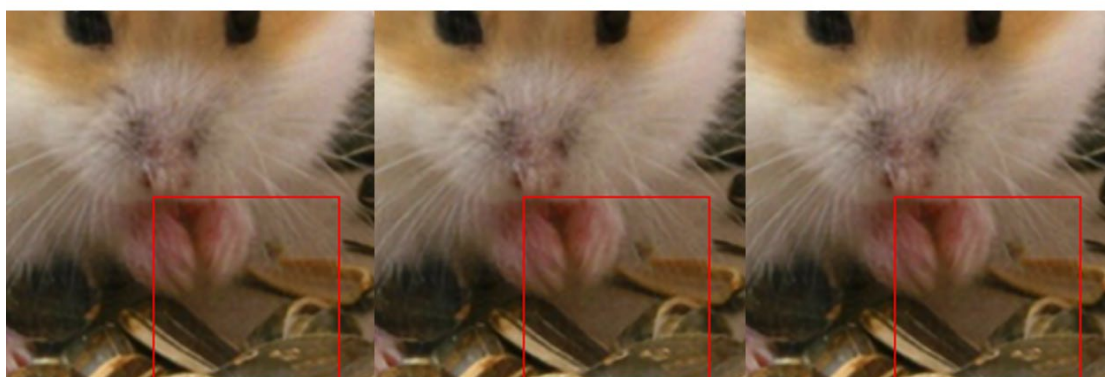
SwinIR



LR

SRCNN

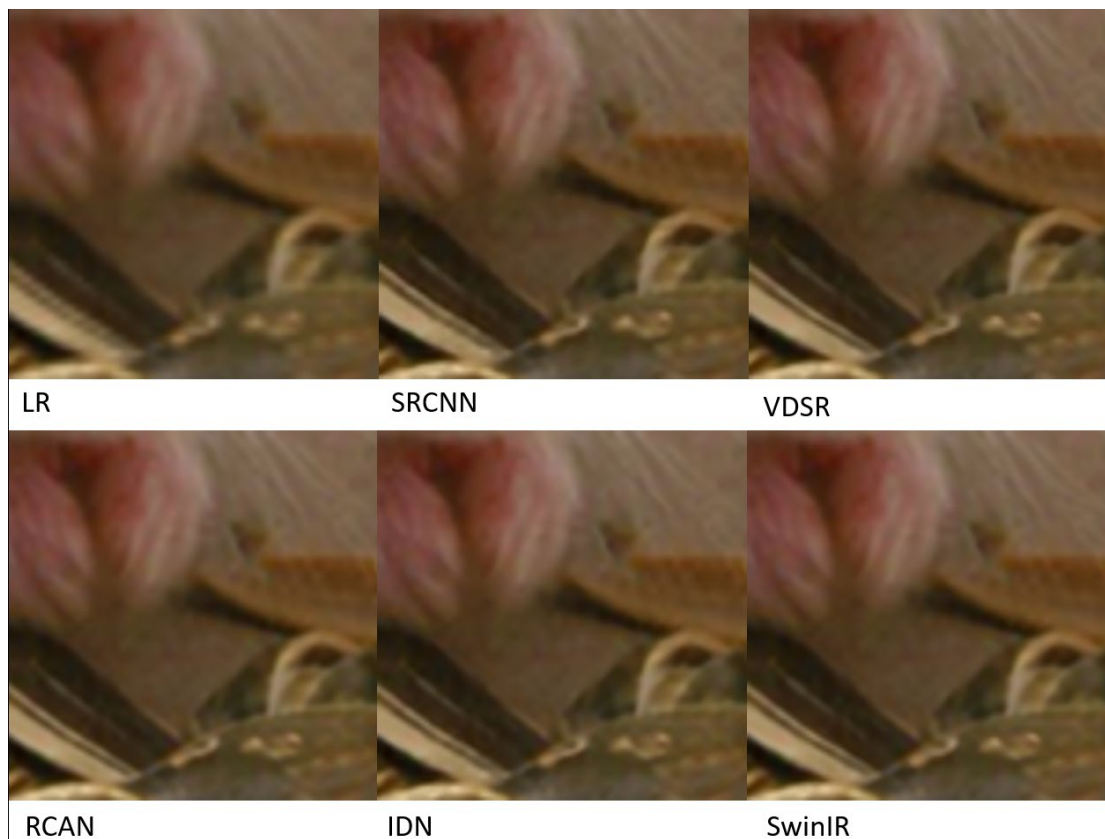
VDSR



RCAN

IDN

SwinIR



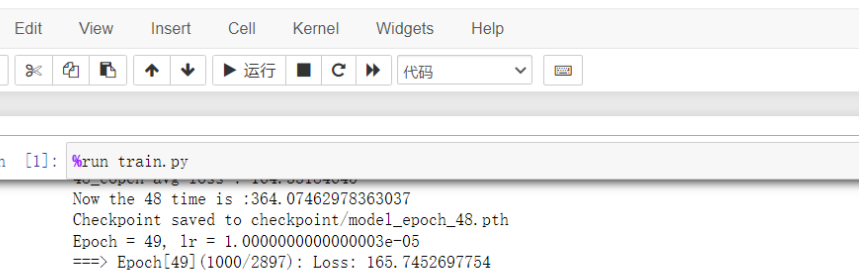
5.2 训练截图

5.2.1 SRCNN

```
In [10]: %run train.py --train-file "BLAH_BLAH/91-image_x2.h5" \
          --eval-file "BLAH_BLAH/Set5_x2.h5" \
          --outputs-dir "BLAH_BLAH/outputs" \
          --scale 2 \
          --lr 1e-4 \
          --batch-size 16 \
          --num-epochs 400 \
          --num-workers 0 \
          --seed 123

390_epoch avg loss : 0.00050709
391_epoch avg loss : 0.00050709
391_epoch eval psnr: 36.57
392_epoch avg loss : 0.00050679
392_epoch eval psnr: 36.60
393_epoch avg loss : 0.00050668
393_epoch eval psnr: 36.58
394_epoch avg loss : 0.00050702
394_epoch eval psnr: 36.59
395_epoch avg loss : 0.00050651
395_epoch eval psnr: 36.60
396_epoch avg loss : 0.00050673
396_epoch eval psnr: 36.57
397_epoch avg loss : 0.00050624
397_epoch eval psnr: 36.60
398_epoch avg loss : 0.00050637
398_epoch eval psnr: 36.61
399_epoch avg loss : 0.00050616
399_epoch eval psnr: 36.60
best epoch: 385, psnr: 36.61
```

5.2.2 VDSR



Jupyter VDSRnaturex2 (更改未保存)

File Edit View Insert Cell Kernel Widgets Help

Run 代码

```
In [1]: %run train.py
```

```
Now the 48 time is :364.07462978363037
Checkpoint saved to checkpoint/model_epoch_48.pth
Epoch = 49, lr = 1.0000000000000003e-05
==> Epoch[49](1000/2897): Loss: 165.7452697754
==> Epoch[49](2000/2897): Loss: 123.6311492920
==> Epoch 49 Complete: Avg. Loss: 164.33136665
ok!-----
49_eopch avg loss : 164.33136665
Now the 49 time is :363.80894804000854
Checkpoint saved to checkpoint/model_epoch_49.pth
Epoch = 50, lr = 1.0000000000000003e-05
==> Epoch[50](1000/2897): Loss: 144.5718688965
==> Epoch[50](2000/2897): Loss: 168.2134399414
==> Epoch 50 Complete: Avg. Loss: 164.33104041
ok!-----
50_eopch avg loss : 164.33104041
Now the 50 time is :363.9061059951782
Checkpoint saved to checkpoint/model_epoch_50.pth
All is ok!
```

5.2.3 RCAN

The screenshot shows the JupyterLab application window. At the top, there's a menu bar with options: File, Edit, View, Insert, Cell, Kernel, Help. Below the menu bar is a toolbar with icons for file operations, running, and code execution. The main area displays a code editor with the following Python code:

```
In [1]: #!python
run main.py --model RCAN --load RCAN_BIX2_G10R20P4S_0422 --resume -1 --pre_train ../experiment/RCAN_BIX2_G10R20P4S_0422/model/model_late
```

Below the code editor, a terminal window is open, displaying the output of the command. The output includes a progress bar at 100%, PSNR values, total time, learning rate, and epoch-wise results.

```
Evaluation:
100% |████████████████████████████████████████████████████████████████████████████████| 5/5 [00:36<00:00, 7.36s/it]

[DIV2K x2] PSNR: 40.442 (Best: 40.471 @epoch 680)
Total time: 36.80s

[Epoch 1000] Learning rate: 1.56e-6
[200/1600] [L1: 3.1117] 57.1+1.1s
[400/1600] [L1: 3.0236] 56.1+1.0s
[600/1600] [L1: 2.9912] 56.2+1.1s
[800/1600] [L1: 2.9823] 57.2+1.0s
[1000/1600] [L1: 3.0012] 56.3+1.0s
[1200/1600] [L1: 2.9462] 55.7+1.0s
[1400/1600] [L1: 2.9559] 60.2+1.2s

0% |████████████████████████████████████████████████████████████████████████████████| 0/5 [00:00<?, ?it/s]

[1600/1600] [L1: 2.9947] 59.6+1.2s

Evaluation:
```


5.2.4 IDN

```
1000

In [1]: %run train

998_epoch avg loss : 0.01882341
Now the 998 time is :1.7789790630340576
Checkpoint saved to checkpoint/model_epoch_998.pth
Epoch = 999, lr = 0.0001
==> Epoch[999](10/10): Loss: 0.0216203574
==> Epoch 999 Complete: Avg. Loss: 0.02021740
ok!-----
999_epoch avg loss : 0.02021740
Now the 999 time is :2.682588577270508
Checkpoint saved to checkpoint/model_epoch_999.pth
Epoch = 1000, lr = 0.0001
==> Epoch[1000](10/10): Loss: 0.0201155599
==> Epoch 1000 Complete: Avg. Loss: 0.01748385
ok!-----
1000_epoch avg loss : 0.01748385
Now the 1000 time is :1.941525936126709
Checkpoint saved to checkpoint/model_epoch_1000.pth
All is ok!
```

5.2.5 SwinIR

```
jupyter SwinIR训练 最新检查点: 几秒前 (已自动保存)

File Edit View Insert Cell Kernel Widgets Help 检查点已创建: 18:13:43 不可信

avg_psnr = avg_psnr / idx

# testing log
logger.info(' <epoch:{:3d}, iter:{:8,d}, Average PSNR : {:.<.2f>dB\n'.format(epoch, current_step, avg_psnr

22-12-14 09:29:42.093 : Saving the model.

superresolution/swinir_sr_lightweight_x2/images/baby
22-12-14 09:29:43.125 : ---1--> baby.png | 37.14dB
superresolution/swinir_sr_lightweight_x2/images/bird
22-12-14 09:29:43.343 : ---2--> bird.png | 39.88dB
22-12-14 09:29:43.495 : ---3--> butterfly.png | 32.72dB
superresolution/swinir_sr_lightweight_x2/images/butterfly
superresolution/swinir_sr_lightweight_x2/images/head
22-12-14 09:29:43.762 : ---4--> head.png | 32.25dB
superresolution/swinir_sr_lightweight_x2/images/woman
22-12-14 09:29:43.979 : ---5--> woman.png | 34.73dB
22-12-14 09:29:44.033 : <epoch:999, iter: 20,000, Average PSNR : 35.35dB
```