微光图像增强代码

# utils.py

import os  
import numpy as np  
import shutil  
from torch.autograd import Variable  
import matplotlib.pyplot as plt  
from PIL import Image  
import torch  
import torch.nn.functional as F  
import logging # 添加logging模块导入  
  
  
def save(model, path):  
 torch.save(model.state\_dict(), path)  
  
  
def count\_parameters\_in\_MB(model):  
 return sum(p.numel() for p in model.parameters()) / 1e6  
  
  
def pair\_downsampler(img):  
 device = img.device  
 dtype = img.dtype  
 B, C, H, W = img.shape  
  
  
  
 # 确保高度和宽度是偶数  
 if H % 2 != 0:  
 img = F.pad(img, (0, 0, 0, 1), mode='reflect')  
 H += 1  
 if W % 2 != 0:  
 img = F.pad(img, (0, 1, 0, 0), mode='reflect')  
 W += 1  
  
 # 确保掩码在相同设备和数据类型上  
 mask1 = torch.tensor([[[[1, 0], [0, 1]]]], dtype=dtype, device=device)  
 mask2 = torch.tensor([[[[0, 1], [1, 0]]]], dtype=dtype, device=device)  
  
 mask1 = mask1.repeat(C, 1, 1, 1)  
 mask2 = mask2.repeat(C, 1, 1, 1)  
  
 output1 = F.conv2d(img, mask1, stride=2, groups=C) \* 2  
 output2 = F.conv2d(img, mask2, stride=2, groups=C) \* 2  
  
  
 return output1, output2  
  
  
def gauss\_cdf(x):  
 return 0.5 \* (1 + torch.erf(x / torch.sqrt(torch.tensor(2.))))  
  
  
# +++ 添加：更真实的微光图像退化函数 +++  
def degrade\_image(image, severity):  
 """  
 根据严重程度应用一系列真实的微光图像退化。  
 severity: 0.0 (无退化) 到 1.0 (严重退化) 的浮点数。  
 """  
 if severity == 0:  
 return image  
  
 device = image.device  
  
 # 1. 模拟欠曝光 (Gamma校正)  
 # 严重程度越高，gamma值越大，图像越暗  
 gamma = 1.0 + severity \* 2.0  
 image\_darkened = image \*\* gamma  
  
 # 2. 模拟色彩偏移 (Color Cast)  
 # 随机选择一个颜色通道进行增强或减弱  
 color\_cast = torch.tensor([1.0, 1.0, 1.0], device=device)  
 channel\_to\_cast = np.random.randint(0, 3)  
 cast\_direction = np.random.choice([-1, 1])  
 # 严重程度越高，色彩偏移越明显  
 cast\_amount = 0.1 + severity \* 0.3  
 color\_cast[channel\_to\_cast] += cast\_direction \* cast\_amount  
 image\_color\_cast = image\_darkened \* color\_cast.view(1, 3, 1, 1)  
  
 # 3. 模拟传感器噪声 (泊松-高斯噪声模型)  
 # 泊松分量 (与信号强度相关)  
 shot\_noise\_strength = 0.1 \* severity  
 image\_with\_shot\_noise = torch.poisson(image\_color\_cast / shot\_noise\_strength) \* shot\_noise\_strength  
  
 # 高斯分量 (读出噪声，与信号无关)  
 read\_noise\_strength = 0.05 \* severity  
 read\_noise = torch.randn\_like(image\_with\_shot\_noise) \* read\_noise\_strength  
  
 image\_noisy = image\_with\_shot\_noise + read\_noise  
  
 return torch.clamp(image\_noisy, 0, 1)  
def gauss\_kernel(kernlen=21, nsig=3, channels=1, device='cpu'):  
 interval = (2 \* nsig + 1.) / kernlen  
 x = torch.linspace(-nsig - interval / 2., nsig + interval / 2., steps=kernlen + 1, device=device)  
 kern1d = torch.diff(0.5 \* (1 + torch.erf(x / torch.sqrt(torch.tensor(2., device=device)))))  
 kernel\_raw = torch.sqrt(torch.outer(kern1d, kern1d))  
 kernel = kernel\_raw / torch.sum(kernel\_raw)  
 out\_filter = kernel.view(1, 1, kernlen, kernlen).repeat(channels, 1, 1, 1)  
 return out\_filter  
  
  
class LocalMean(torch.nn.Module):  
 def \_\_init\_\_(self, patch\_size=5):  
 super(LocalMean, self).\_\_init\_\_()  
 self.patch\_size = patch\_size  
 self.padding = self.patch\_size // 2  
  
 def forward(self, image):  
 image = torch.nn.functional.pad(image, (self.padding, self.padding, self.padding, self.padding), mode='reflect')  
 patches = image.unfold(2, self.patch\_size, 1).unfold(3, self.patch\_size, 1)  
 return patches.mean(dim=(4, 5))  
  
  
def blur(x):  
 device = x.device  
 kernel\_size = 21  
 padding = kernel\_size // 2  
 kernel\_var = gauss\_kernel(kernel\_size, 1, x.size(1), device=device)  
 x\_padded = F.pad(x, (padding, padding, padding, padding), mode='reflect')  
 return F.conv2d(x\_padded, kernel\_var, padding=0, groups=x.size(1))  
  
  
def padr\_tensor(img):  
 pad = 2  
 pad\_mod = torch.nn.ConstantPad2d(pad, 0)  
 img\_pad = pad\_mod(img)  
 return img\_pad  
  
  
def calculate\_local\_variance(train\_noisy):  
 b, c, w, h = train\_noisy.shape  
 avg\_pool = torch.nn.AvgPool2d(kernel\_size=5, stride=1, padding=2)  
 noisy\_avg = avg\_pool(train\_noisy)  
 noisy\_avg\_pad = padr\_tensor(noisy\_avg)  
 train\_noisy = padr\_tensor(train\_noisy)  
 unfolded\_noisy\_avg = noisy\_avg\_pad.unfold(2, 5, 1).unfold(3, 5, 1)  
 unfolded\_noisy = train\_noisy.unfold(2, 5, 1).unfold(3, 5, 1)  
 unfolded\_noisy\_avg = unfolded\_noisy\_avg.reshape(unfolded\_noisy\_avg.shape[0], -1, 5, 5)  
 unfolded\_noisy = unfolded\_noisy.reshape(unfolded\_noisy.shape[0], -1, 5, 5)  
 noisy\_diff\_squared = (unfolded\_noisy - unfolded\_noisy\_avg) \*\* 2  
 noisy\_var = torch.mean(noisy\_diff\_squared, dim=(2, 3))  
 noisy\_var = noisy\_var.view(b, c, w, h)  
  
 # 添加方差统计  
 var\_stats = {  
 'min': noisy\_var.min().item(),  
 'max': noisy\_var.max().item(),  
 'mean': noisy\_var.mean().item(),  
 'std': noisy\_var.std().item()  
 }  
 logging.debug(f"Local variance stats: {var\_stats}")  
  
 return noisy\_var  
  
  
def save\_checkpoint(state, is\_best, save):  
 filename = os.path.join(save, 'checkpoint.pth.tar')  
 torch.save(state, filename)  
 if is\_best:  
 best\_filename = os.path.join(save, 'model\_best.pth.tar')  
 shutil.copyfile(filename, best\_filename)  
  
  
def load(model, model\_path):  
 model.load\_state\_dict(torch.load(model\_path))  
  
  
def drop\_path(x, drop\_prob):  
 if drop\_prob > 0.:  
 keep\_prob = 1. - drop\_prob  
 mask = Variable(torch.cuda.FloatTensor(x.size(0), 1, 1, 1).bernoulli\_(keep\_prob))  
 x.div\_(keep\_prob)  
 x.mul\_(mask)  
 return x  
  
  
def create\_exp\_dir(path, scripts\_to\_save=None):  
 if not os.path.exists(path):  
 os.makedirs(path, exist\_ok=True)  
 print('实验目录 : {}'.format(path))  
  
 if scripts\_to\_save is not None:  
 os.makedirs(os.path.join(path, 'scripts'), exist\_ok=True)  
 for script in scripts\_to\_save:  
 dst\_file = os.path.join(path, 'scripts', os.path.basename(script))  
 shutil.copyfile(script, dst\_file)  
  
  
def show\_pic(pic, name, path):  
 pic\_num = len(pic)  
 for i in range(pic\_num):  
 img = pic[i]  
 image\_numpy = img[0].cpu().float().numpy()  
 if image\_numpy.shape[0] == 3:  
 image\_numpy = np.transpose(image\_numpy, (1, 2, 0))  
 im = Image.fromarray(np.clip(image\_numpy \* 255.0, 0, 255.0).astype('uint8'))  
 img\_name = name[i]  
 plt.subplot(5, 6, i + 1)  
 plt.xlabel(str(img\_name))  
 plt.xticks([])  
 plt.yticks([])  
 plt.imshow(im)  
 elif image\_numpy.shape[0] == 1:  
 im = Image.fromarray(np.clip(image\_numpy[0] \* 255.0, 0, 255.0).astype('uint8'))  
 img\_name = name[i]  
 plt.subplot(5, 6, i + 1)  
 plt.xlabel(str(img\_name))  
 plt.xticks([])  
 plt.yticks([])  
 plt.imshow(im, plt.cm.gray)  
 plt.savefig(path)

# train.py

import os  
  
os.environ["PYTORCH\_CUDA\_ALLOC\_CONF"] = "expandable\_segments:True"  
import os  
  
import matplotlib.pyplot as plt  
import sys  
import time  
import torch  
import math  
import glob  
import numpy as np  
import utils  
from torch.optim.lr\_scheduler import LambdaLR, CosineAnnealingLR  
from PIL import Image  
import logging  
import argparse  
from torch.utils.data import DataLoader as TorchDataLoader  
from model import Network  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.utils  
import torch.backends.cudnn as cudnn  
from torch.autograd import Variable  
from model import \*  
from multi\_read\_data import DataLoader  
import gc  
import pyiqa  
import lpips as lpips\_lib  
from adamp import AdamP  
import pandas as pd  
from torch.optim.lr\_scheduler import ReduceLROnPlateau  
  
# 设置CUDA环境和优化选项  
os.environ["PYTORCH\_CUDA\_ALLOC\_CONF"] = "max\_split\_size\_mb:128"  
torch.backends.cuda.matmul.allow\_tf32 = True  
torch.cuda.empty\_cache()  
torch.backends.cudnn.benchmark = False  
  
# 解析命令行参数  
parser = argparse.ArgumentParser("ZERO-IG")  
parser.add\_argument('--batch\_size', type=int, default=1, help='批处理大小')  
  
parser.add\_argument('--cuda', type=bool, default=True, help='是否使用CUDA训练')  
parser.add\_argument('--gpu', type=str, default='0', help='GPU设备ID')  
parser.add\_argument('--seed', type=int, default=2, help='随机种子')  
parser.add\_argument('--epochs', type=int, default=6001, help='训练轮数')  
parser.add\_argument('--lr\_gen', type=float, default=1e-4, help='生成器学习率')  
parser.add\_argument('--lr\_disc', type=float, default=1e-4, help='判别器学习率')  
parser.add\_argument('--save', type=str, default='./EXP/', help='实验结果保存根目录')  
parser.add\_argument('--model\_pretrain', type=str, default='', help='预训练模型路径')  
parser.add\_argument('--adv\_weight', type=float, default=0.08, help='对抗损失权重')  
parser.add\_argument('--patience', type=int, default=1000, help='早停耐心值')  
parser.add\_argument('--min\_delta', type=float, default=0.002, help='最小提升阈值')  
parser.add\_argument('--disc\_update\_freq', type=int, default=2, help='判别器更新频率')  
parser.add\_argument('--gradient\_penalty\_weight', type=float, default=2.0, help='梯度惩罚权重')  
args = parser.parse\_args()  
  
# 设置可见GPU设备  
os.environ["CUDA\_VISIBLE\_DEVICES"] = args.gpu  
print(f"CUDA 可用: {torch.cuda.is\_available()}")  
print(f"CUDA 设备数量: {torch.cuda.device\_count()}")  
print(f"当前 CUDA 设备: {torch.cuda.current\_device()}")  
  
# 创建实验目录  
args.save = os.path.join(args.save, f"Train-{time.strftime('%Y%m%d-%H%M%S')}")  
utils.create\_exp\_dir(args.save, scripts\_to\_save=glob.glob('\*.py'))  
model\_path = os.path.join(args.save, 'model\_epochs/')  
os.makedirs(model\_path, exist\_ok=True)  
image\_path = os.path.join(args.save, 'image\_epochs/')  
os.makedirs(image\_path, exist\_ok=True)  
  
# 配置日志  
log\_format = '%(asctime)s %(message)s'  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format=log\_format, datefmt='%m/%d %I:%M:%S %p')  
fh = logging.FileHandler(os.path.join(args.save, 'log.txt'))  
fh.setFormatter(logging.Formatter(log\_format))  
logging.getLogger().addHandler(fh)  
logging.info("train file name = %s", os.path.split(\_\_file\_\_)[1])  
  
# 设置默认tensor类型  
if torch.cuda.is\_available() and args.cuda:  
 torch.set\_default\_dtype(torch.float32)  
else:  
 torch.set\_default\_tensor\_type('torch.FloatTensor')  
  
  
def save\_images(tensor):  
 if tensor.dim() == 3:  
 tensor = tensor.unsqueeze(0)  
 image\_numpy = tensor[0].cpu().float().numpy()  
 image\_numpy = np.transpose(image\_numpy, (1, 2, 0))  
 im = np.clip(image\_numpy \* 255.0, 0, 255.0).astype('uint8')  
 return im  
  
  
def print\_memory\_usage(stage):  
 alloc = torch.cuda.memory\_allocated() / 1024 \*\* 3 # GB  
 reserved = torch.cuda.memory\_reserved() / 1024 \*\* 3  
 print(f"[{stage}] 已分配: {alloc:.2f}GB, 已预留: {reserved:.2f}GB")  
  
  
def analyze\_training\_metrics(metrics\_path):  
 """分析训练指标"""  
 try:  
 df = pd.read\_csv(metrics\_path)  
 if df.empty:  
 logging.warning("Metrics CSV file is empty.")  
 return None  
  
 # 检查必要的列是否存在  
 required\_columns = ['Epoch', 'PSNR', 'SSIM', 'LPIPS', 'NIQE']  
 for col in required\_columns:  
 if col not in df.columns:  
 logging.warning(f"Column {col} not found in metrics file")  
 return None  
  
 # 处理NaN值 - 使用新方法  
 df = df.ffill().bfill()  
  
 # 确保所有指标列都是数值类型  
 for col in ['PSNR', 'SSIM', 'LPIPS', 'NIQE']:  
 df[col] = pd.to\_numeric(df[col], errors='coerce')  
  
 # 再次处理可能的NaN  
 df = df.ffill().bfill()  
  
 # 按epoch分组计算平均指标  
 epoch\_metrics = df.groupby('Epoch').agg({  
 'PSNR': 'mean',  
 'SSIM': 'mean',  
 'LPIPS': 'mean',  
 'NIQE': 'mean'  
 }).reset\_index()  
  
 # 找到每个指标的最佳epoch  
 best\_psnr\_idx = epoch\_metrics['PSNR'].idxmax()  
 best\_ssim\_idx = epoch\_metrics['SSIM'].idxmax()  
 best\_lpips\_idx = epoch\_metrics['LPIPS'].idxmin()  
 best\_niqe\_idx = epoch\_metrics['NIQE'].idxmin()  
  
 # 获取最佳指标值  
 best\_psnr = epoch\_metrics.loc[best\_psnr\_idx, ['Epoch', 'PSNR']]  
 best\_ssim = epoch\_metrics.loc[best\_ssim\_idx, ['Epoch', 'SSIM']]  
 best\_lpips = epoch\_metrics.loc[best\_lpips\_idx, ['Epoch', 'LPIPS']]  
 best\_niqe = epoch\_metrics.loc[best\_niqe\_idx, ['Epoch', 'NIQE']]  
  
 # 计算综合得分  
 epoch\_metrics['norm\_psnr'] = (epoch\_metrics['PSNR'] - epoch\_metrics['PSNR'].min()) / (  
 epoch\_metrics['PSNR'].max() - epoch\_metrics['PSNR'].min() + 1e-8)  
 epoch\_metrics['norm\_ssim'] = (epoch\_metrics['SSIM'] - epoch\_metrics['SSIM'].min()) / (  
 epoch\_metrics['SSIM'].max() - epoch\_metrics['SSIM'].min() + 1e-8)  
 epoch\_metrics['norm\_lpips'] = 1 - (epoch\_metrics['LPIPS'] - epoch\_metrics['LPIPS'].min()) / (  
 epoch\_metrics['LPIPS'].max() - epoch\_metrics['LPIPS'].min() + 1e-8)  
 epoch\_metrics['norm\_niqe'] = 1 - (epoch\_metrics['NIQE'] - epoch\_metrics['NIQE'].min()) / (  
 epoch\_metrics['NIQE'].max() - epoch\_metrics['NIQE'].min() + 1e-8)  
  
 epoch\_metrics['composite\_score'] = (  
 epoch\_metrics['norm\_psnr'] \* 0.4 +  
 epoch\_metrics['norm\_ssim'] \* 0.4 +  
 epoch\_metrics['norm\_lpips'] \* 0.1 +  
 epoch\_metrics['norm\_niqe'] \* 0.1  
 )  
  
 best\_composite\_idx = epoch\_metrics['composite\_score'].idxmax()  
 best\_composite = epoch\_metrics.loc[best\_composite\_idx, ['Epoch', 'PSNR', 'SSIM', 'LPIPS', 'NIQE']]  
  
 return {  
 'best\_psnr': best\_psnr,  
 'best\_ssim': best\_ssim,  
 'best\_lpips': best\_lpips,  
 'best\_niqe': best\_niqe,  
 'best\_composite': best\_composite  
 }  
 except Exception as e:  
 logging.error(f"分析训练指标时出错: {e}")  
 return None  
  
  
def write\_best\_metrics\_to\_log(metrics\_path, log\_path):  
 """将最佳指标写入日志文件"""  
 best\_metrics = analyze\_training\_metrics(metrics\_path)  
 if not best\_metrics:  
 return  
  
 with open(log\_path, 'r') as f:  
 original\_content = f.read()  
  
 best\_info = f"""最佳指标总结:  
PSNR最佳: epoch {int(best\_metrics['best\_psnr']['Epoch'])} - {best\_metrics['best\_psnr']['PSNR']:.4f}  
SSIM最佳: epoch {int(best\_metrics['best\_ssim']['Epoch'])} - {best\_metrics['best\_ssim']['SSIM']:.4f}  
LPIPS最佳: epoch {int(best\_metrics['best\_lpips']['Epoch'])} - {best\_metrics['best\_lpips']['LPIPS']:.4f}  
NIQE最佳: epoch {int(best\_metrics['best\_niqe']['Epoch'])} - {best\_metrics['best\_niqe']['NIQE']:.4f}  
综合最佳: epoch {int(best\_metrics['best\_composite']['Epoch'])} - PSNR: {best\_metrics['best\_composite']['PSNR']:.4f}, SSIM: {best\_metrics['best\_composite']['SSIM']:.4f}, LPIPS: {best\_metrics['best\_composite']['LPIPS']:.4f}, NIQE: {best\_metrics['best\_composite']['NIQE']:.4f}  
  
"""  
  
 with open(log\_path, 'w') as f:  
 f.write(best\_info + original\_content)  
  
 logging.info(best\_info)  
  
  
def normalize\_for\_discriminator(x):  
 """将输入图像裁剪到[0, 1]范围，匹配真实图像分布"""  
 return torch.clamp(x, 0, 1)  
  
  
# +++ 修改：实现零中心梯度惩罚 (0-GP) 以提升判别器泛化能力 +++  
def compute\_gradient\_penalty(D, real\_samples):  
 """计算应用于真实样本的零中心梯度惩罚 (R1 正则化)"""  
 real\_samples.requires\_grad\_(True)  
 d\_real = D(real\_samples)  
 grad\_outputs = torch.ones\_like(d\_real, requires\_grad=False)  
  
 gradients = torch.autograd.grad(  
 outputs=d\_real,  
 inputs=real\_samples,  
 grad\_outputs=grad\_outputs,  
 create\_graph=True,  
 retain\_graph=True,  
 only\_inputs=True  
 )  
  
 gradients = gradients.view(gradients.size(0), -1)  
 # 惩罚梯度范数的平方，使其趋向于0  
 gradient\_penalty = (gradients.norm(2, dim=1) \*\* 2).mean()  
 return gradient\_penalty  
 # +++ 修改：实现零中心梯度惩罚 (0-GP) 以提升判别器泛化能力 +++  
  
class EMA:  
 """指数移动平均"""  
  
 def \_\_init\_\_(self, model, decay=0.999):  
 self.model = model  
 self.decay = decay  
 self.shadow = {}  
 self.backup = {}  
  
 def register(self):  
 for name, param in self.model.named\_parameters():  
 if param.requires\_grad:  
 self.shadow[name] = param.data.clone()  
  
 def update(self):  
 for name, param in self.model.named\_parameters():  
 if param.requires\_grad:  
 self.shadow[name] = self.decay \* self.shadow[name] + (1 - self.decay) \* param.data  
  
 def apply\_shadow(self):  
 for name, param in self.model.named\_parameters():  
 if param.requires\_grad:  
 self.backup[name] = param.data  
 param.data = self.shadow[name]  
  
 def restore(self):  
 for name, param in self.model.named\_parameters():  
 if param.requires\_grad:  
 param.data = self.backup[name]  
  
  
class EarlyStopping:  
 def \_\_init\_\_(self, patience=2000, min\_delta=0.0005, warmup\_epochs=1000):  
 self.patience = patience  
 self.min\_delta = min\_delta  
 self.warmup\_epochs = warmup\_epochs  
 self.counter = 0  
 self.best\_score = None  
 self.early\_stop = False  
  
 def \_\_call\_\_(self, composite\_score, current\_epoch):  
 if current\_epoch < self.warmup\_epochs:  
 return False  
  
 if self.best\_score is None:  
 self.best\_score = composite\_score  
 elif composite\_score < self.best\_score + self.min\_delta:  
 self.counter += 1  
 if self.counter >= self.patience:  
 self.early\_stop = True  
 else:  
 self.best\_score = composite\_score  
 self.counter = 0  
 return self.early\_stop  
  
  
def adjust\_loss\_weights(epoch):  
 """动态调整损失权重"""  
 # 前500epoch：主要学习基础重建  
 if epoch < 500:  
 weights = {  
 'pixel\_reconstruction': 1.5,  
 'perceptual': 0.2,  
 'texture\_preserve': 0.3,  
 'color\_constancy': 0.1,  
 'histogram\_match': 0.1,  
 'ms\_ssim': 0.8,  
 'frequency': 0.1,  
 'noise\_aware': 0.5,  
 'psnr': 0.1,  
 'overexposure\_weight': 0.4,  
 'adv\_weight': 0.05 # 大幅降低对抗损失权重  
 }  
 # 500-2000epoch：平衡各项损失  
 elif epoch < 1500:  
 weights = {  
 'pixel\_reconstruction': 1.5 - 0.7 \* (epoch - 500) / 1000,  
 'perceptual': min(0.4, 0.2 + 0.6 \* (epoch - 500) / 1000),  
 'texture\_preserve': min(0.5, 0.3 + 0.5 \* (epoch - 500) / 1000),  
 'color\_constancy': 0.1,  
 'histogram\_match': 0.1 + 0.1 \* (epoch - 500) / 1000,  
 'ms\_ssim': min(1.2, 0.8 + 0.4 \* (epoch - 500) / 1000),  
 'frequency': min(0.2, 0.1 + 0.1 \* (epoch - 500) / 1000),  
 'noise\_aware': 0.5 + 0.3 \* (epoch - 500) / 1000,  
 'psnr': min(0.2, 0.1 + 0.1 \* (epoch - 500) / 1500),  
 'overexposure\_weight': min(0.6, 0.3 + 0.3 \* (epoch - 500) / 1000),  
 'adv\_weight': 0.05 + 0.1 \* (epoch - 500) / 1000,  
 }  
 # 2000epoch后：专注于感知质量和细节  
 else:  
 weights = {  
 'pixel\_reconstruction': 0.8,  
 'perceptual': 0.8,  
 'texture\_preserve': 0.8,  
 'color\_constancy': 0.1,  
 'histogram\_match': 0.2,  
 'ms\_ssim': 1.2,  
 'frequency': 0.2,  
 'noise\_aware': 0.8,  
 'psnr': 0.2,  
 'overexposure\_weight': 0.5,  
 'adv\_weight': 0.15  
 }  
  
 return weights  
  
  
  
  
def adaptive\_brightness\_control(image, max\_brightness=0.92, min\_avg\_brightness=0.4):  
 """  
 新增逻辑：  
 - 若平均亮度 < 0.3（正常下限），按比例提升亮度  
 - 过曝处理保留，但降低亮度衰减系数（从 0.9/0.8 改为 0.95/0.9）  
 """  
 # 计算图像平均亮度  
 brightness = 0.299 \* image[:, 0] + 0.587 \* image[:, 1] + 0.114 \* image[:, 2]  
 avg\_brightness = torch.mean(brightness)  
 overexposed = (brightness > max\_brightness).float()  
 overexposed\_ratio = overexposed.mean()  
  
 # 1. 亮度不足时：按比例提升（目标达到 min\_avg\_brightness）  
 if avg\_brightness < min\_avg\_brightness:  
 scale = min\_avg\_brightness / (avg\_brightness + 1e-6) # 提升比例（如 0.3/0.15=2.0）  
 scale = torch.clamp(scale, 1.0, 3.0)  
 image = image \* scale  
  
 # 2. 过曝时：轻微降低亮度（衰减系数从 0.9/0.8 改为 0.95/0.9，减少过度抑制）  
 if overexposed\_ratio > 0.1:  
 image = image \* 0.95 # 原 0.9 → 0.95  
 elif overexposed\_ratio > 0.05:  
 mask = overexposed.unsqueeze(0).expand\_as(image)  
 image = torch.where(mask > 0, image \* 0.9, image) # 原 0.8 → 0.9  
  
 return torch.clamp(image, 0, 1)  
  
  
def check\_nan\_inf(tensor, name):  
 """检查张量中是否有NaN或Inf值"""  
 if torch.isnan(tensor).any() or torch.isinf(tensor).any():  
 logging.warning(f"Warning: {name} contains NaN or Inf values.")  
 return True  
 return False  
  
  
def main():  
 # 启用梯度异常检测，当出现nan/inf梯度时提供详细堆栈跟踪  
 torch.autograd.set\_detect\_anomaly(True)  
 if not torch.cuda.is\_available():  
 logging.info('无可用GPU设备，退出。')  
 sys.exit(1)  
  
 device = torch.device("cuda:0" if args.cuda else "cpu")  
 print(f"使用设备: {device}")  
  
 # 初始化混合精度训练  
 scaler\_gen = torch.amp.GradScaler('cuda', enabled=False, growth\_interval=200)  
 scaler\_disc = torch.amp.GradScaler('cuda', enabled=False, growth\_interval=200)  
  
 # 设置随机种子  
 np.random.seed(args.seed)  
 cudnn.benchmark = True  
 torch.manual\_seed(args.seed)  
 cudnn.enabled = True  
 torch.cuda.manual\_seed(args.seed)  
 logging.info('使用GPU设备 = %s' % args.gpu)  
 logging.info("参数 = %s", args)  
  
 # 初始化模型  
 model = Network()  
 model.enhance.init\_conv.apply(model.enhance\_weights\_init)  
 for block in model.enhance.blocks:  
 for layer in block:  
 if isinstance(layer, nn.Conv2d):  
 layer.apply(model.enhance\_weights\_init)  
 model.enhance.final\_conv.apply(model.enhance\_weights\_init)  
 model = model.to(device)  
 torch.set\_default\_dtype(torch.float32) # 确保默认数据类型为float32  
 # 添加模型参数初始化检查与修正  
 for name, param in model.named\_parameters():  
 if torch.isnan(param).any() or torch.isinf(param).any():  
 logging.warning(f"参数 {name} 包含NaN或Inf值，重新初始化")  
 # 使用xavier均匀分布重新初始化有问题的参数  
 nn.init.xavier\_uniform\_(param.data)  
 print\_memory\_usage("模型初始化后（含参数）")  
  
 # 初始化EMA  
 ema = EMA(model, decay=0.999)  
 ema.register()  
  
 # 初始化指标模型  
 lpips\_model = lpips\_lib.LPIPS(net='alex').to(device)  
 psnr\_metric = pyiqa.create\_metric('psnr', device=device)  
 ssim\_metric = pyiqa.create\_metric('ssim', device=device)  
 niqe\_metric = pyiqa.create\_metric('niqe', device=device)  
 model.\_criterion = model.\_criterion.to(device)  
  
 # 初始化早停机制  
 early\_stopping = EarlyStopping(patience=1000, min\_delta=0.01, warmup\_epochs=500)  
  
 # 优化器 - 修正参数绑定问题  
 generator\_params = []  
 for name, param in model.named\_parameters():  
 if not name.startswith('discriminator'): # 排除判别器参数  
 generator\_params.append(param)  
  
 # 为判别器设置更高的学习率 (TTUR)  
 generator\_optimizer = AdamP(generator\_params, lr=args.lr\_gen, betas=(0.9, 0.999), weight\_decay=1e-4)  
 discriminator\_optimizer = AdamP(model.discriminator.parameters(), lr=args.lr\_disc, betas=(0.5, 0.999),  
 weight\_decay=1e-4)  
  
 # +++ 修改：学习率调度器 - 调整T\_max以加速收敛 +++  
 scheduler\_gen = torch.optim.lr\_scheduler.CosineAnnealingLR(  
 generator\_optimizer,  
 T\_max=1000, # 从2000减少到1000，加速收敛  
 eta\_min=1e-7  
 )  
 scheduler\_disc = torch.optim.lr\_scheduler.CosineAnnealingLR(  
 discriminator\_optimizer,  
 T\_max=500, # 从1000减少到500  
 eta\_min=1e-6  
 )  
  
 # 加载数据集  
 train\_low\_dir = './data/LOL-V1/lol\_dataset/eval15/cs/low'  
 train\_target\_dir = './data/LOL-V1/lol\_dataset/eval15/cs/high'  
 test\_low\_dir = './data/LOL-V1/lol\_dataset/eval15/cs/low'  
 test\_target\_dir = './data/LOL-V1/lol\_dataset/eval15/cs/high'  
  
 TestDataset = DataLoader(img\_dir=test\_low\_dir, target\_dir=test\_target\_dir, task='test')  
 TrainDataset = DataLoader(img\_dir=train\_low\_dir, target\_dir=train\_target\_dir, task='train')  
  
 # 打印模型参数量  
 MB = utils.count\_parameters\_in\_MB(model)  
 logging.info("模型参数量 = %f MB", MB)  
 print(f"Model Parameters: {MB:.2f} MB")  
  
 # 创建数据加载器  
 train\_queue = TorchDataLoader(TrainDataset, batch\_size=args.batch\_size, pin\_memory=False, num\_workers=0,  
 shuffle=True)  
 test\_queue = TorchDataLoader(TestDataset, batch\_size=1, pin\_memory=False, num\_workers=0, shuffle=False)  
  
 # 初始化指标日志文件  
 metrics\_log\_path = os.path.join(args.save, 'training\_metrics.csv')  
 # 确保目录存在  
 metrics\_dir = os.path.dirname(metrics\_log\_path)  
 os.makedirs(metrics\_dir, exist\_ok=True)  
 # 初始化CSV文件并写入表头（仅当文件不存在时）  
 if not os.path.exists(metrics\_log\_path):  
 with open(metrics\_log\_path, 'w') as f:  
 f.write("Epoch,Image\_Name,PSNR,SSIM,LPIPS,NIQE\n")  
  
 # 初始化详细指标日志文件  
 detailed\_metrics\_path = os.path.join(args.save, 'detailed\_metrics.csv')  
 if not os.path.exists(detailed\_metrics\_path):  
 with open(detailed\_metrics\_path, 'w') as f:  
 f.write("Epoch,PSNR,SSIM,LPIPS,NIQE\n")  
  
 total\_step = 0  
 model.train()  
 best\_composite\_score = -float('inf')  
  
 # 添加梯度监控函数  
 def get\_grad\_norms(model, layer\_names):  
 """获取指定层的梯度范数"""  
 grad\_norms = {}  
 for name, param in model.named\_parameters():  
 if param.grad is not None and any(layer\_name in name for layer\_name in layer\_names):  
 grad\_norms[name] = param.grad.data.norm(2).item()  
 return grad\_norms  
  
 # 指定要监控的层  
 monitor\_layers = ['enhance', 'denoise\_1', 'denoise\_2', 'discriminator']  
  
 try:  
 for epoch in range(args.epochs):  
 if epoch < 100:  
 disc\_update\_freq = 5 # 训练初期：每5步更新1次（减少判别器压制）  
 elif epoch < 500:  
 disc\_update\_freq = 3 # 训练中期：每3步更新1次（平衡对抗）  
 else:  
 disc\_update\_freq = 2 # 训练后期：每2步更新1次（正常对抗）  
 # 新增2：记录当前更新频率，方便调试  
 logging.info(f"Epoch {epoch} | 判别器更新频率: 每{disc\_update\_freq}步更新1次")  
  
 # 应用课程学习策略  
 loss\_weights = adjust\_loss\_weights(epoch)  
 # 新增：获取当前噪声水平  
  
  
 # 更新损失函数中的权重 - 使用新的权重字典  
 model.\_criterion.current\_weights = {  
 'pixel\_reconstruction': loss\_weights['pixel\_reconstruction'],  
 'perceptual': loss\_weights['perceptual'],  
 'texture\_preserve': loss\_weights['texture\_preserve'],  
 'color\_constancy': loss\_weights['color\_constancy'],  
 'histogram\_match': loss\_weights['histogram\_match'],  
 'ms\_ssim': loss\_weights['ms\_ssim'],  
 'frequency': loss\_weights['frequency'],  
 'noise\_aware': loss\_weights['noise\_aware'],  
 'psnr': loss\_weights['psnr']  
 }  
 model.\_criterion.overexposure\_weight = loss\_weights['overexposure\_weight']  
 args.adv\_weight = loss\_weights['adv\_weight']  
  
 losses\_gen = []  
 losses\_disc = [0.0]  
  
 for idx, (input, target, img\_name) in enumerate(train\_queue):  
 total\_step += 1  
 input = input.to(device).requires\_grad\_(True)  
 target = target.to(device)  
 # 新增：根据当前噪声水平添加高斯噪声到输入  
 # +++ 修改：应用更真实的退化作为课程学习 +++  
 # 随着epoch增加，退化程度从0线性增加到1（前2000个epoch达到最大）  
 degradation\_severity = min(1.0, epoch / 2000.0)  
 logging.info(f"Epoch {epoch} 退化严重程度: {degradation\_severity:.4f}")  
 input\_degraded = utils.degrade\_image(input, degradation\_severity)  
  
 # 1. 训练判别器 - 每disc\_update\_freq步训练一次  
 # 修改判别器训练部分  
 if (total\_step % disc\_update\_freq == 0) and (len(losses\_disc) > 0 and abs(losses\_disc[-1]) < 100):  
 try:  
 discriminator\_optimizer.zero\_grad()  
  
 with torch.amp.autocast('cuda'):  
 with torch.no\_grad():  
 outputs = model(input\_degraded)  
 pred\_img = outputs['H2'].detach() # 从生成器分离，避免梯度传回  
  
 # 准备判别器的输入，确保范围正确且仅变换一次  
 # 真实图像从 变换到 [-1, 1]  
 real\_input = torch.clamp(target \* 2 - 1, -1.0, 1.0)  
 fake\_input = torch.clamp(pred\_img \* 2 - 1, -1.0, 1.0)  
  
 # 为安全起见，进行钳位操作  
 real\_input = torch.clamp(real\_input, -1.0, 1.0)  
 fake\_input = torch.clamp(fake\_input, -1.0, 1.0)  
  
 # 判别器前向传播  
 real\_pred = model.discriminator(real\_input)  
 fake\_pred = model.discriminator(fake\_input)  
  
 # 添加数值稳定性处理  
 real\_pred = torch.clamp(real\_pred, -10, 10)  
 fake\_pred = torch.clamp(fake\_pred, -10, 10)  
  
 # 记录判别器预测分布  
 disc\_stats = {  
 'real\_mean': real\_pred.mean().item(),  
 'real\_std': real\_pred.std().item(),  
 'real\_range': [real\_pred.min().item(), real\_pred.max().item()],  
 'fake\_mean': fake\_pred.mean().item(),  
 'fake\_std': fake\_pred.std().item(),  
 'fake\_range': [fake\_pred.min().item(), fake\_pred.max().item()]  
 }  
  
 # 记录判别器梯度  
 disc\_grad\_norms = get\_grad\_norms(model.discriminator, monitor\_layers)  
  
 # 记录数据信息  
 data\_info = {  
 'batch\_index': idx,  
 'total\_batches': len(train\_queue),  
 'input\_range': [input.min().item(), input.max().item()],  
 'target\_range': [target.min().item(), target.max().item()],  
 'input\_degraded\_range': [input\_degraded.min().item(), input\_degraded.max().item()],  
  
 }  
  
 # 计算WGAN-GP损失 - 添加数值稳定性处理,绝对值  
 disc\_loss = torch.mean(real\_pred) - torch.mean(fake\_pred)  
 # 添加梯度惩罚  
 # +++ 修改：应用0-GP到真实样本 +++  
 gradient\_penalty = compute\_gradient\_penalty(model.discriminator, real\_input)  
 disc\_loss = disc\_loss + args.gradient\_penalty\_weight \* gradient\_penalty # 使用新的权重参数  
  
 # 添加判别器损失正则化  
 disc\_regularization = 0.001 \* torch.mean(real\_pred \*\* 2)  
 disc\_loss = disc\_loss + disc\_regularization  
  
 # 检查损失有效性  
 if torch.isnan(disc\_loss) or torch.isinf(disc\_loss):  
 logging.warning("Invalid discriminator loss, skipping update")  
 discriminator\_optimizer.zero\_grad()  
 continue  
  
 # 反向传播和优化  
 scaler\_disc.scale(disc\_loss).backward()  
 scaler\_disc.unscale\_(discriminator\_optimizer)  
 # 添加梯度裁剪  
 torch.nn.utils.clip\_grad\_norm\_(model.discriminator.parameters(), max\_norm=1.0)  
 torch.nn.utils.clip\_grad\_value\_(model.discriminator.parameters(), clip\_value=0.5)  
 scaler\_disc.step(discriminator\_optimizer)  
 scaler\_disc.update()  
 # 更严格的梯度裁剪  
  
  
  
  
 losses\_disc.append(disc\_loss.item())  
  
 # 每10步记录详细信息  
 if total\_step % 10 == 0:  
 logging.info(f"Discriminator Stats: {disc\_stats}")  
 logging.info(f"Discriminator Grad Norms: {disc\_grad\_norms}")  
 logging.info(f"Data Info: {data\_info}")  
  
 logging.info(f"Discriminator trained successfully, loss: {disc\_loss.item():.6f}")  
 if gen\_loss\_val > 1000:  
 logging.warning(  
 f"Anomaly detected at step {total\_step}: Gen loss = {gen\_loss\_val}, Image = {img\_name}")  
 # Also log the breakdown during anomalies  
 detailed\_losses = model.\_criterion.get\_detailed\_loss\_components()  
 logging.warning(f"Anomaly Loss Breakdown: {detailed\_losses}")  
  
 except Exception as e:  
 logging.error(f"Error training discriminator: {e}")  
 # 重置梯度，防止累积  
 discriminator\_optimizer.zero\_grad()  
 # 跳过本次更新但记录一个合理值  
 losses\_disc.append(1.0) # 使用中性值而不是0  
 continue  
 else:  
 # 即使不更新判别器，也记录上一次损失（避免列表为空）  
 if losses\_disc: # 列表非空时记录上一次值  
 losses\_disc.append(losses\_disc[-1])  
 else:  
 losses\_disc.append(0.0) # 初始值  
 # 2. 训练生成器  
 generator\_optimizer.zero\_grad()  
  
 # 初始化变量，避免未定义错误  
 pred = None  
 fake\_pred = None  
 content\_loss = None  
 adv\_loss = None  
 gen\_loss = None  
 outputs = None  
  
 # 仅调用一次model(input)，复用输出  
 with torch.amp.autocast('cuda'):  
 outputs = model(input\_degraded)  
 for key, tensor in outputs.items():  
 if torch.is\_tensor(tensor) and (torch.isnan(tensor).any() or torch.isinf(tensor).any()):  
 logging.error(f"NaN/Inf found in {key} at step {total\_step}. Skipping batch.")  
 continue # 跳过这个batch  
  
  
 # 直接使用修复后的outputs，不再重复调用  
 pred = outputs['H2']  
 pred = torch.clamp(pred, 0, 1) # 确保在[0,1]范围  
 # 转换到[-1,1]范围再输入判别器  
 pred\_disc = pred \* 2 - 1  
 fake\_pred\_g = model.discriminator(pred\_disc)  
  
 logging.info(  
 f"[DEBUG] H2 - min: {pred.min().item():.6f}, max: {pred.max().item():.6f}, mean: {pred.mean().item():.6f}")  
  
 gen\_content\_loss = model.\_loss(input, target, epoch=epoch, \*\*outputs) # 内部已按current\_weights加权  
 if torch.isnan(gen\_content\_loss) or torch.isinf(gen\_content\_loss):  
 logging.warning("Invalid content loss, skipping batch")  
 continue  
 # 使用新的权重键  
 gen\_content\_loss = gen\_content\_loss \* loss\_weights['pixel\_reconstruction']  
  
 pred\_disc = torch.clamp(pred \* 2 - 1, -1.0, 1.0)  
 fake\_pred\_g = model.discriminator(pred\_disc)  
  
 # 添加数值稳定性处理  
 fake\_pred\_g = torch.clamp(fake\_pred\_g, -10, 10)  
 if torch.isnan(fake\_pred\_g).any() or torch.isinf(fake\_pred\_g).any():  
 logging.warning("NaN or Inf in fake\_pred\_g")  
 continue  
 gen\_adv\_loss = -torch.mean(fake\_pred\_g)  
  
 # 总生成器损失（使用动态调整的对抗权重）  
 gen\_loss = gen\_content\_loss + args.adv\_weight \* gen\_adv\_loss  
  
 # 检查生成器损失是否有NaN或Inf  
 if check\_nan\_inf(gen\_loss, "gen\_loss"):  
 generator\_optimizer.zero\_grad()  
 continue # 跳过这个batch  
  
 # 确保gen\_loss是张量  
 if not torch.is\_tensor(gen\_loss):  
 gen\_loss = torch.tensor(gen\_loss, device=device, dtype=torch.float32, requires\_grad=True)  
  
 # 获取详细损失分量（如果可用）  
 try:  
 detailed\_loss = model.\_criterion.get\_detailed\_loss\_components()  
 except:  
 detailed\_loss = "Not available"  
  
 # 获取中间层输出（如果可用）  
 try:  
 intermediate\_outputs = model.get\_intermediate\_outputs()  
 except:  
 intermediate\_outputs = "Not available"  
  
 # 获取生成器梯度  
 gen\_grad\_norms = get\_grad\_norms(model, monitor\_layers)  
  
 # 记录学习率  
 lr\_info = {  
 'gen\_expected': scheduler\_gen.get\_last\_lr()[0],  
 'gen\_actual': generator\_optimizer.param\_groups[0]['lr'],  
 'disc\_expected': scheduler\_disc.get\_last\_lr()[0],  
 'disc\_actual': discriminator\_optimizer.param\_groups[0]['lr']  
 }  
  
 # 每10步记录详细信息  
 if total\_step % 10 == 0:  
 logging.info(f"Generator Loss Breakdown: {detailed\_loss}")  
 logging.info(f"Intermediate Outputs: {intermediate\_outputs}")  
 logging.info(f"Generator Grad Norms: {gen\_grad\_norms}")  
 logging.info(f"Learning Rate Info: {lr\_info}")  
  
 # 记录参数更新量  
 param\_update\_norms = {}  
 for name, param in model.named\_parameters():  
 if param.grad is not None and any(layer\_name in name for layer\_name in monitor\_layers):  
 update\_norm = (param.grad.data \* generator\_optimizer.param\_groups[0]['lr']).norm(  
 2).item()  
 param\_update\_norms[name] = update\_norm  
 logging.info(f"Parameter Update Norms: {param\_update\_norms}")  
  
 # 保存损失值用于日志  
 gen\_loss\_val = gen\_loss.item()  
 # +++ 新增：在反向传播前检查gen\_loss的有效性 +++  
 if check\_nan\_inf(gen\_loss, "gen\_loss"):  
 generator\_optimizer.zero\_grad()  
 continue # 跳过这个batch  
 # 反向传播和优化生成器  
 scaler\_gen.scale(gen\_loss).backward()  
 scaler\_gen.unscale\_(generator\_optimizer)  
  
 # 使用更温和的梯度裁剪  
 torch.nn.utils.clip\_grad\_norm\_(generator\_params, max\_norm=0.8) # 收紧最大范数  
  
  
 # 设置更保守的自动混合精度  
  
 scaler\_gen.step(generator\_optimizer) # 优化器步骤  
 scaler\_gen.update() # 混合精度更新  
  
 losses\_gen.append(gen\_loss\_val)  
  
 # 每10步打印详细日志  
 if total\_step % 10 == 0:  
 # 记录判别器损失（如果已计算）  
 disc\_loss\_val = losses\_disc[-1] if losses\_disc else 0.0  
 logging.info('epoch %d step %d gen\_loss %f disc\_loss %f',  
 epoch, total\_step, gen\_loss\_val, disc\_loss\_val)  
  
 # 记录各损失组件  
 if hasattr(model.\_criterion, 'current\_weights'):  
 logging.info(f"损失权重: {model.\_criterion.current\_weights}")  
  
 # 记录亮度统计  
 if hasattr(model.\_criterion, 'avg\_brightness'):  
 logging.info(  
 f"平均亮度: {model.\_criterion.avg\_brightness:.4f}, 过曝比例: {model.\_criterion.overexposure\_ratio:.4f}")  
  
 # 记录学习率  
 current\_lr\_gen = generator\_optimizer.param\_groups[0]['lr']  
 current\_lr\_disc = discriminator\_optimizer.param\_groups[0]['lr']  
 logging.info(f"学习率 - 生成器: {current\_lr\_gen:.2e}, 判别器: {current\_lr\_disc:.2e}")  
  
 # 记录噪声分类结果  
 if 'noise\_prob' in outputs:  
 noise\_prob = outputs['noise\_prob']  
 logging.info(  
 f"噪声概率 - 高斯: {noise\_prob[0, 0]:.3f}, 泊松: {noise\_prob[0, 1]:.3f}, 椒盐: {noise\_prob[0, 2]:.3f}")  
  
 # 更新EMA  
 ema.update()  
  
 # 清理显存  
 if total\_step % 2 == 0:  
 torch.cuda.empty\_cache()  
 gc.collect()  
  
 # 每50步监控梯度  
 if total\_step % 50 == 0:  
 # 监控梯度范数  
 total\_grad\_norm = 0  
 grad\_norms = []  
 for name, param in model.named\_parameters():  
 if param.grad is not None and "generator" in name:  
 param\_grad\_norm = param.grad.data.norm(2).item()  
 grad\_norms.append((name, param\_grad\_norm))  
 total\_grad\_norm += param\_grad\_norm \*\* 2  
  
 # 修复过小的梯度（梯度消失）  
 if param\_grad\_norm < 1e-8:  
 logging.warning(f"梯度消失检测: {name}, 范数: {param\_grad\_norm:.8f}")  
 # 添加少量噪声重启梯度  
 param.grad.data += torch.randn\_like(param.grad.data) \* 1e-6  
  
 # 修复过大的梯度（梯度爆炸）  
 if param\_grad\_norm > 1000:  
 logging.warning(f"梯度爆炸检测: {name}, 范数: {param\_grad\_norm:.2f}")  
 torch.nn.utils.clip\_grad\_norm\_([param], max\_norm=10.0)  
  
 total\_grad\_norm = total\_grad\_norm \*\* 0.5  
 logging.info(f'总梯度范数: {total\_grad\_norm:.6f}')  
  
 # 记录前5个最大梯度  
 grad\_norms.sort(key=lambda x: x[1], reverse=True)  
 for i, (name, norm) in enumerate(grad\_norms[:5]):  
 logging.info(f'梯度TOP{i + 1}: {name} = {norm:.6f}')  
  
 # 监控参数更新量  
 param\_update\_norm = 0  
 for p in generator\_params:  
 if p.grad is not None:  
 param\_update\_norm += (p.grad.data \* generator\_optimizer.param\_groups[0]['lr']).norm(  
 2).item() \*\* 2  
 param\_update\_norm = param\_update\_norm \*\* 0.5  
  
 logging.info(f'参数更新量: {param\_update\_norm:.8f}')  
 try:  
 # 判别器对真实样本的输出范围  
 if 'real\_pred' in locals():  
 logging.info(f"Real pred range: [{real\_pred.min():.3f}, {real\_pred.max():.3f}]")  
 # 判别器对生成样本的输出范围  
 if 'fake\_pred' in locals():  
 logging.info(f"Fake pred range: [{fake\_pred.min():.3f}, {fake\_pred.max():.3f}]")  
 # 生成器输出的数值范围  
 if 'pred' in locals():  
 logging.info(f"Gen output range: [{pred.min():.3f}, {pred.max():.3f}]")  
 except Exception as e:  
 logging.warning(f"监控输出范围时出错: {e}")  
  
 # 每50步记录batch级指标  
 if total\_step % 50 == 0 and target is not None:  
 with torch.no\_grad():  
 psnr\_val = psnr\_metric(outputs['H2'], target)  
 ssim\_val = ssim\_metric(outputs['H2'], target)  
 logging.info(f"Batch {idx} Metrics - PSNR: {psnr\_val.item():.4f}, SSIM: {ssim\_val.item():.4f}")  
  
 # 异常检测  
 if gen\_loss\_val > 1000: # 异常阈值  
 logging.warning(  
 f"Anomaly detected at step {total\_step}: Gen loss = {gen\_loss\_val}, Image = {img\_name}")  
  
 # 每100步检查参数和梯度的数值稳定性（NaN/Inf/范围）  
 if total\_step % 100 == 0:  
 # 检查参数NaN/Inf  
 for name, param in model.named\_parameters():  
 if torch.isnan(param).any():  
 logging.warning(f"NaN detected in parameter: {name}")  
 if torch.isinf(param).any():  
 logging.warning(f"Inf detected in parameter: {name}")  
  
 # 检查梯度NaN/Inf  
 for name, param in model.named\_parameters():  
 if param.grad is not None:  
 if torch.isnan(param.grad).any():  
 logging.warning(f"NaN detected in gradient: {name}")  
 if torch.isinf(param.grad).any():  
 logging.warning(f"Inf detected in gradient: {name}")  
  
 # 检查参数范围（避免数值爆炸）  
 for name, param in model.named\_parameters():  
 if param.numel() > 0: # 跳过空参数  
 param\_min = param.min().item()  
 param\_max = param.max().item()  
 if abs(param\_max) > 1e4 or abs(param\_min) > 1e4:  
 logging.warning(  
 f"Parameter {name} has large values: min={param\_min:.4f}, max={param\_max:.4f}")  
  
 # 清理变量，只删除已定义的变量  
 variables\_to\_delete = ['pred', 'fake\_pred', 'content\_loss', 'adv\_loss', 'gen\_loss', 'outputs']  
 for var\_name in variables\_to\_delete:  
 if var\_name in locals():  
 del locals()[var\_name]  
 torch.cuda.empty\_cache()  
 gc.collect()  
  
 # 添加损失列表空值检查  
 if not losses\_gen:  
 logging.warning(f"Epoch {epoch}: 生成器损失列表为空，可能训练步骤被跳过")  
 continue  
  
 if not losses\_disc:  
 logging.warning(f"Epoch {epoch}: 判别器损失列表为空，可能训练步骤被跳过")  
 continue  
  
 # 更新学习率  
 mean\_gen\_loss = np.mean(losses\_gen) if losses\_gen else 0.0  
 mean\_disc\_loss = np.mean(losses\_disc) if losses\_disc else 0.0  
  
 current\_lr\_gen = generator\_optimizer.param\_groups[0]['lr']  
 current\_lr\_disc = discriminator\_optimizer.param\_groups[0]['lr']  
  
 logging.info(f"Current Learning Rates - Gen: {current\_lr\_gen}, Disc: {current\_lr\_disc}")  
  
 # 使用生成器损失作为监控指标  
 scheduler\_gen.step() # 余弦退火调度器在每个epoch结束时更新，无需传入参数  
 scheduler\_disc.step()  
  
 # 验证和保存  
 if total\_step != 0 and epoch % 2 == 0: # 每2个epoch验证一次以节省时间  
 ema.apply\_shadow()  
 model.eval()  
 epoch\_metrics = []  
 composite\_scores = []  
  
 # 记录更详细的验证指标  
 epoch\_metrics\_detailed = []  
  
 with torch.no\_grad():  
 for idx, (input, target, img\_name) in enumerate(test\_queue):  
 input = Variable(input).to(device)  
 target = Variable(target).to(device)  
  
 image\_name = os.path.splitext(os.path.basename(img\_name[0]))[0]  
  
 outputs = model(input)  
 enhanced\_H3 = outputs['H3']  
 enhanced\_H2 = outputs['H2']  
  
 # 应用自适应亮度控制并确保范围正确  
 enhanced\_H2 = adaptive\_brightness\_control(enhanced\_H2)  
 enhanced\_H3 = torch.clamp(enhanced\_H3, 0, 1)  
 enhanced\_H2 = torch.clamp(enhanced\_H2, 0, 1)  
  
 # 确保目标图像也在正确范围内  
 target = torch.clamp(target, 0, 1)  
  
 # 计算指标 - 确保输入范围正确  
 # PSNR和SSIM需要确保输入在[0,1]范围内  
 psnr\_value = psnr\_metric(enhanced\_H2, target)  
 ssim\_value = ssim\_metric(enhanced\_H2, target)  
  
 # LPIPS需要将输入从[0,1]转换到[-1,1]  
 lpips\_input = enhanced\_H2 \* 2 - 1 # [0,1] -> [-1,1]  
 lpips\_target = target \* 2 - 1 # [0,1] -> [-1,1]  
 lpips\_value = lpips\_model(lpips\_input, lpips\_target).mean()  
  
 # NIQE只需要增强后的图像  
 niqe\_value = niqe\_metric(enhanced\_H2)  
  
 epoch\_metrics.append({  
 'name': image\_name,  
 'psnr': psnr\_value.item(),  
 'ssim': ssim\_value.item(),  
 'lpips': lpips\_value.item(),  
 'niqe': niqe\_value.item()  
 })  
  
 metrics = {  
 'name': image\_name,  
 'psnr': psnr\_value.item(),  
 'ssim': ssim\_value.item(),  
 'lpips': lpips\_value.item(),  
 'niqe': niqe\_value.item(),  
 # 记录验证时的中间输出（如果可用）  
 'intermediate\_outputs': model.get\_intermediate\_outputs() if hasattr(model,  
 'get\_intermediate\_outputs') else "Not available"  
 }  
 epoch\_metrics\_detailed.append(metrics)  
  
 # 计算综合得分  
 composite\_score = (psnr\_value.item() / 40 \* 0.4 + # PSNR归一化  
 ssim\_value.item() \* 0.4 + # SSIM  
 (1 - lpips\_value.item()) \* 0.1 + # LPIPS反向  
 (1 - min(niqe\_value.item() / 10, 1)) \* 0.1) # NIQE归一化  
 composite\_scores.append(composite\_score)  
  
 with open(metrics\_log\_path, 'a') as f:  
 f.write(  
 f"{epoch},{image\_name},{psnr\_value.item():.4f},{ssim\_value.item():.4f},{lpips\_value.item():.4f},{niqe\_value.item():.4f}\n")  
 f.flush() # 确保数据立即写入磁盘，避免缓存导致的数据丢失  
  
 # 在验证循环中添加以下调试代码  
 if epoch % 2 == 0:  
 # 添加输入和目标图像的统计信息  
 logging.info(f"输入图像范围: [{input.min().item():.4f}, {input.max().item():.4f}]")  
 logging.info(f"目标图像范围: [{target.min().item():.4f}, {target.max().item():.4f}]")  
 logging.info(  
 f"增强图像范围: [{enhanced\_H2.min().item():.4f}, {enhanced\_H2.max().item():.4f}]")  
  
 # 检查PSNR计算是否正确  
 mse = F.mse\_loss(enhanced\_H2, target)  
 manual\_psnr = 10 \* torch.log10(1.0 / (mse + 1e-10))  
 if abs(psnr\_value.item() - manual\_psnr.item()) > 0.1:  
 logging.warning(  
 f"PSNR计算不一致: 库计算={psnr\_value.item():.4f}, 手动计算={manual\_psnr.item():.4f}")  
 # 定期保存图像  
 if epoch % 50 == 0:  
 H3\_img = save\_images(enhanced\_H3)  
 denoise\_dir = os.path.join(args.save, 'result/denoise')  
 os.makedirs(denoise\_dir, exist\_ok=True)  
 Image.fromarray(H3\_img).save(os.path.join(denoise\_dir, f"{image\_name}\_denoise\_{epoch}.png"),  
 'PNG')  
  
 H2\_img = save\_images(enhanced\_H2)  
 enhance\_dir = os.path.join(args.save, 'result/enhance')  
 os.makedirs(enhance\_dir, exist\_ok=True)  
 Image.fromarray(H2\_img).save(os.path.join(enhance\_dir, f"{image\_name}\_enhance\_{epoch}.png"),  
 'PNG')  
  
 # 记录平均指标  
 avg\_metrics = {  
 'psnr': np.mean([m['psnr'] for m in epoch\_metrics\_detailed]),  
 'ssim': np.mean([m['ssim'] for m in epoch\_metrics\_detailed]),  
 'lpips': np.mean([m['lpips'] for m in epoch\_metrics\_detailed]),  
 'niqe': np.mean([m['niqe'] for m in epoch\_metrics\_detailed])  
 }  
  
 logging.info(f"Epoch {epoch} Detailed Metrics: {avg\_metrics}")  
  
 # 保存详细指标到文件  
 with open(detailed\_metrics\_path, 'a') as f:  
 f.write(  
 f"{epoch},{avg\_metrics['psnr']},{avg\_metrics['ssim']},{avg\_metrics['lpips']},{avg\_metrics['niqe']}\n")  
  
 ema.restore() # 恢复原始权重  
 model.train() # 切换回训练模式  
 # 计算平均指标  
 avg\_psnr = np.mean([m['psnr'] for m in epoch\_metrics]) if epoch\_metrics else 0  
 avg\_ssim = np.mean([m['ssim'] for m in epoch\_metrics]) if epoch\_metrics else 0  
 avg\_lpips = np.mean([m['lpips'] for m in epoch\_metrics]) if epoch\_metrics else 0  
 avg\_niqe = np.mean([m['niqe'] for m in epoch\_metrics]) if epoch\_metrics else 0  
 avg\_composite = np.mean(composite\_scores) if composite\_scores else 0  
  
 logging.info(  
 f"Epoch {epoch} Metrics - PSNR: {avg\_psnr:.4f}, SSIM: {avg\_ssim:.4f}, LPIPS: {avg\_lpips:.4f}, NIQE: {avg\_niqe:.4f}, Composite: {avg\_composite:.4f}")  
  
 # 检查早停 - 添加current\_epoch参数  
 if early\_stopping(avg\_composite, epoch):  
 logging.info(f"早停触发于 epoch {epoch}")  
 break  
  
 # 保存模型（当PSNR指标大于20时保存）  
 if avg\_psnr > 20:  
 torch.save(model.state\_dict(), os.path.join(model\_path, f'best\_model.pt'))  
 logging.info(f"PSNR大于20，保存模型，当前PSNR: {avg\_psnr:.4f}")  
  
 ema.restore()  
 model.train()  
  
 gc.collect()  
 torch.cuda.empty\_cache()  
 alloc = torch.cuda.memory\_allocated() / 1024 \*\* 3 # 已分配显存（GB）  
 reserved = torch.cuda.memory\_reserved() / 1024 \*\* 3 # 已预留显存（GB）  
 logging.info('epoch %d GPU Memory - Allocated: %.2fGB, Reserved: %.2fGB',  
 epoch, alloc, reserved)  
  
 # 修改检查点保存条件  
 avg\_psnr = np.mean([m['psnr'] for m in epoch\_metrics])  
 if epoch % 100 == 0 and avg\_psnr > 18: # 只在PSNR>18时保存  
 # 保存完整训练状态（包含模型、优化器、调度器等）  
 checkpoint = {  
 'epoch': epoch,  
 'model\_state': model.state\_dict(),  
 'gen\_optimizer': generator\_optimizer.state\_dict(),  
 'disc\_optimizer': discriminator\_optimizer.state\_dict(),  
 'gen\_scaler': scaler\_gen.state\_dict(), # 混合精度缩放器状态  
 'disc\_scaler': scaler\_disc.state\_dict(),  
 'gen\_scheduler': scheduler\_gen.state\_dict(), # 学习率调度器状态  
 'disc\_scheduler': scheduler\_disc.state\_dict(),  
 'gen\_losses': losses\_gen, # 当前epoch生成器损失  
 'disc\_losses': losses\_disc, # 当前epoch判别器损失  
 'ema\_state': ema.shadow, # EMA模型状态  
 'best\_composite\_score': best\_composite\_score # 最佳综合得分  
 }  
 # 确保保存路径存在  
 checkpoint\_path = os.path.join(model\_path, f'checkpoint\_epoch\_{epoch}.pt')  
 torch.save(checkpoint, checkpoint\_path)  
 logging.info(f"Saved full checkpoint to {checkpoint\_path}")  
 # 内存监控  
 alloc = torch.cuda.memory\_allocated() / 1024 \*\* 3  
 reserved = torch.cuda.memory\_reserved() / 1024 \*\* 3  
 print(f"Epoch {epoch} | 已分配: {alloc:.2f}GB | 已预留: {reserved:.2f}GB")  
  
 except KeyboardInterrupt:  
 logging.info("训练被用户中断")  
 except Exception as e:  
 logging.error(f"训练过程中发生错误: {e}")  
 finally:  
 # 保存最终模型  
 torch.save(model.state\_dict(), os.path.join(model\_path, 'final\_model.pt'))  
  
 # 分析指标并写入日志  
 log\_path = os.path.join(args.save, 'log.txt')  
 write\_best\_metrics\_to\_log(metrics\_log\_path, log\_path)  
 logging.info("训练结束，最佳指标已写入日志文件首行")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# test.py

import os  
import sys  
import numpy as np  
import torch  
import torch.nn.functional as F  
import argparse  
  
import logging  
import lpips as lpips\_lib  
import pyiqa  
from PIL import Image  
from torch.autograd import Variable  
from model import Finetunemodel # 导入微调模型类  
from multi\_read\_data import DataLoader # 导入自定义数据加载器  
from thop import profile # 用于计算模型FLOPs  
from ultralytics import YOLO  
import torchvision.transforms as T  
import cv2  
  
# 设置根目录路径，确保可以正确导入项目模块  
root\_dir = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), "../"))  
sys.path.append(root\_dir)  
  
# 解析命令行参数  
parser = argparse.ArgumentParser("ZERO-IG")  
parser.add\_argument('--data\_path\_test\_low', type=str, default='./data/LOL-V1/lol\_dataset/eval15/cs/low',  
 help='测试低光图像路径')  
parser.add\_argument('--data\_path\_test\_target', type=str, default='./data/LOL-V1/lol\_dataset/eval15/cs/high',  
 help='（可选）测试目标图像路径，用于计算指标')  
parser.add\_argument('--save', type=str, default='./results/', help='结果保存目录')  
parser.add\_argument('--model\_test', type=str, default='./EXP/Train-20250729-002842/model\_epochs/weights\_800.pt',  
 help='预训练模型权重路径')  
parser.add\_argument('--gpu', type=int, default=0, help='使用的GPU设备ID')  
parser.add\_argument('--seed', type=int, default=2, help='随机种子（保证结果可复现）')  
parser.add\_argument('--yolo\_size', type=int, default=640, help='YOLO输入尺寸')  
parser.add\_argument('--tta', action='store\_true', help='启用测试时增强')  
args = parser.parse\_args()  
  
# 创建结果保存目录  
save\_path = args.save  
os.makedirs(save\_path, exist\_ok=True)  
  
# 配置日志输出到控制台和文件  
log\_format = '%(asctime)s %(message)s'  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format=log\_format, datefmt='%m/%d %I:%M:%S %p')  
metric\_log = logging.FileHandler(os.path.join(save\_path, 'log.txt'))  
metric\_log.setFormatter(logging.Formatter(log\_format))  
logging.getLogger().addHandler(metric\_log)  
logging.info("test file name = %s", os.path.split(\_\_file\_\_)[1])  
  
# 加载测试数据集  
TestDataset = DataLoader(img\_dir=args.data\_path\_test\_low,  
 target\_dir=(args.data\_path\_test\_target if args.data\_path\_test\_target else None),  
 task='test')  
test\_queue = torch.utils.data.DataLoader(TestDataset, batch\_size=1, pin\_memory=True, num\_workers=0, shuffle=False)  
  
  
def save\_images(tensor):  
 # 将模型输出的张量转为可保存图像格式（支持单张）  
 if tensor.dim() == 3:  
 tensor = tensor.unsqueeze(0)  
 image\_numpy = tensor[0].cpu().float().numpy()  
 image\_numpy = np.transpose(image\_numpy, (1, 2, 0))  
 im = np.clip(image\_numpy \* 255.0, 0, 255.0).astype('uint8')  
 return im  
  
  
def calculate\_model\_parameters(model):  
 # 计算模型参数总量  
 return sum(p.numel() for p in model.parameters())  
  
  
def calculate\_model\_flops(model, input\_tensor):  
 # 计算模型FLOPs（浮点运算次数）  
 flops, \_ = profile(model, inputs=(input\_tensor,))  
 return flops / 1e9 # 转换为GFLOPs  
  
  
def adaptive\_brightness\_control(image, max\_brightness=0.92, min\_avg\_brightness=0.35):  
 brightness = 0.299 \* image[:, 0] + 0.587 \* image[:, 1] + 0.114 \* image[:, 2]  
 avg\_brightness = torch.mean(brightness)  
  
 # 更保守的亮度调整  
 if avg\_brightness < min\_avg\_brightness:  
 scale = min\_avg\_brightness / (avg\_brightness + 1e-6)  
 scale = torch.clamp(scale, 1.0, 2.0) # 限制最大提升2倍  
 image = image \* scale  
  
 # 更精细的过曝处理  
 overexposed = (brightness > max\_brightness).float()  
 overexposed\_ratio = overexposed.mean()  
  
 if overexposed\_ratio > 0.08:  
 # 使用平滑的过曝修复  
 correction\_mask = torch.clamp((brightness - max\_brightness) / (1 - max\_brightness), 0, 1)  
 correction\_strength = 0.1 + 0.4 \* correction\_mask # 动态调整修复强度  
 image = image \* (1 - correction\_strength.unsqueeze(1))  
  
 return torch.clamp(image, 0, 1)  
  
  
def estimate\_noise\_level(image):  
 """估计图像噪声水平"""  
 if image.dim() == 3:  
 image = image.unsqueeze(0)  
  
 # 转换为灰度图  
 gray = 0.299 \* image[:, 0] + 0.587 \* image[:, 1] + 0.114 \* image[:, 2]  
  
 # 计算局部方差  
 local\_var = torch.var(gray.unfold(2, 5, 1).unfold(3, 5, 1), dim=(-2, -1))  
  
 # 噪声水平估计为局部方差的平均值  
 noise\_level = torch.mean(torch.sqrt(local\_var + 1e-6))  
  
 return noise\_level.item()  
  
  
def adaptive\_denoise(enhanced\_image, noise\_level=0.1):  
 """  
 自适应后处理降噪  
 noise\_level: 0-1之间，控制降噪强度  
 """  
 # 转换为numpy格式  
 if isinstance(enhanced\_image, torch.Tensor):  
 enhanced\_image = enhanced\_image.cpu().numpy()  
 if enhanced\_image.shape[0] == 3:  
 enhanced\_image = np.transpose(enhanced\_image, (1, 2, 0))  
  
 # 根据噪声水平选择降噪参数  
 h = 3 + int(15 \* noise\_level) # h值从3到18  
 template\_window\_size = 7  
 search\_window\_size = 21  
  
 # 将 RGB 转换为 BGR 以供 OpenCV 处理  
 enhanced\_image\_bgr = cv2.cvtColor((enhanced\_image \* 255).astype(np.uint8), cv2.COLOR\_RGB2BGR)  
  
 # 应用非局部均值降噪  
 denoised\_bgr = cv2.fastNlMeansDenoisingColored(  
 enhanced\_image\_bgr,  
 None,  
 h, h, template\_window\_size, search\_window\_size  
 )  
  
 # 将 BGR 结果转换回 RGB  
 denoised = cv2.cvtColor(denoised\_bgr, cv2.COLOR\_BGR2RGB)  
  
 return denoised.astype(np.float32) / 255.0  
  
  
def evaluate\_detection\_mAP(enhanced\_images, target\_images, model, orig\_size=(600, 400)):  
 """使用 YOLO 计算增强图像和目标图像上的 mAP"""  
 # 转换张量为YOLO可接受的输入格式 (0-255范围的RGB图像)  
 transform = T.Compose([  
 T.Normalize(mean=[0, 0, 0], std=[255, 255, 255]),  
 T.ToPILImage()  
 ])  
  
 # 处理增强图像  
 enhanced\_img = transform(enhanced\_images.squeeze(0).cpu())  
 # 处理目标图像  
 target\_img = transform(target\_images.squeeze(0).cpu())  
  
 # 保存原始尺寸  
 enhanced\_orig\_size = enhanced\_img.size  
 target\_orig\_size = target\_img.size  
  
 # 执行检测  
 enhanced\_results = model(enhanced\_img, verbose=False)  
 target\_results = model(target\_img, verbose=False)  
  
 # 计算mAP@0.5  
 enhanced\_map = enhanced\_results[0].boxes.map50 if enhanced\_results[0].boxes is not None else 0.0  
 target\_map = target\_results[0].boxes.map50 if target\_results[0].boxes is not None else 0.0  
  
 return enhanced\_map, target\_map  
  
def calculate\_metrics(enhanced, target, device, psnr\_metric, ssim\_metric, lpips\_model, niqe\_metric, yolo\_model, noise\_level):  
 """  
 统一计算所有评估指标  
 Args:  
 enhanced: 增强后的图像张量 (已归一化到[0,1])  
 target: 目标图像张量 (若存在，已归一化到[0,1])  
 device: 计算设备  
 psnr\_metric/ssim\_metric/lpips\_model/niqe\_metric: 指标计算模型  
 yolo\_model: YOLO检测模型  
 noise\_level: 噪声水平估计值  
 Returns:  
 包含所有指标的字典  
 """  
 metrics = {  
 'psnr': None,  
 'ssim': None,  
 'lpips': None,  
 'niqe': None,  
 'noise\_level': noise\_level,  
 'enhance\_map': None,  
 'target\_map': None  
 }  
  
 # 计算NIQE（无参考指标，始终计算）  
 metrics['niqe'] = niqe\_metric(enhanced).item()  
  
 # 若存在目标图像，计算全参考指标  
 if target is not None:  
 # 确保数据在相同设备  
 enhanced = enhanced.to(device)  
 target = target.to(device)  
  
 # 计算PSNR、SSIM、LPIPS  
 metrics['psnr'] = psnr\_metric(enhanced, target).item()  
 metrics['ssim'] = ssim\_metric(enhanced, target).item()  
 metrics['lpips'] = lpips\_model(enhanced, target).mean().item()  
  
 # 计算目标检测mAP  
 enhance\_map, target\_map = evaluate\_detection\_mAP(enhanced, target, yolo\_model)  
 metrics['enhance\_map'] = enhance\_map  
 metrics['target\_map'] = target\_map  
  
 return metrics  
def resize\_for\_yolo(image\_pil, target\_size=640):  
 """  
 调整图像尺寸以适应YOLO输入，保持宽高比并进行填充  
 """  
 # 计算缩放比例  
 orig\_width, orig\_height = image\_pil.size  
 scale = min(target\_size / orig\_width, target\_size / orig\_height)  
  
 # 计算新尺寸  
 new\_width = int(orig\_width \* scale)  
 new\_height = int(orig\_height \* scale)  
  
 # 调整图像大小  
 resized = image\_pil.resize((new\_width, new\_height), Image.BILINEAR)  
  
 # 创建新图像并进行填充  
 new\_image = Image.new('RGB', (target\_size, target\_size), (114, 114, 114))  
 new\_image.paste(resized, ((target\_size - new\_width) // 2, (target\_size - new\_height) // 2))  
  
 return new\_image, scale, (target\_size - new\_width) // 2, (target\_size - new\_height) // 2  
  
  
def main():  
 if not torch.cuda.is\_available():  
 print('无可用GPU设备，测试终止。')  
 sys.exit(1)  
 # 设置所用设备和随机种子  
 device = torch.device(f"cuda:{args.gpu}" if torch.cuda.is\_available() else "cpu")  
 torch.manual\_seed(args.seed)  
 np.random.seed(args.seed)  
  
 # 初始化指标模型  
 psnr\_metric = pyiqa.create\_metric('psnr', device=device)  
 ssim\_metric = pyiqa.create\_metric('ssim', device=device)  
 niqe\_metric = pyiqa.create\_metric('niqe', device=device)  
 lpips\_model = lpips\_lib.LPIPS(net='alex').to(device)  
  
 # 创建指标日志文件  
 metric\_log\_path = os.path.join(save\_path, 'metrics\_log.txt')  
 with open(metric\_log\_path, 'w') as f:  
 f.write("Image Name, PSNR, SSIM, LPIPS, NIQE, Noise\_Level, mAP(enhance), mAP(target)\n")  
  
 # 加载预训练模型权重  
 model = Finetunemodel(args.model\_test)  
 model = model.to(device)  
 model.eval()  
 model.use\_tta = args.tta # 根据参数启用TTA  
  
 # 计算模型参数量并输出  
 total\_params = calculate\_model\_parameters(model)  
 logging.info("总参数量: %f M", total\_params / 1e6)  
 # 冻结模型参数  
 for p in model.parameters():  
 p.requires\_grad = False  
  
 # 加载YOLO模型（只需加载一次）  
 yolo\_model = YOLO('yolov5s.pt').to(device)  
  
 # YOLO模型预热  
 logging.info("预热YOLO模型...")  
 dummy\_input = torch.randn(1, 3, args.yolo\_size, args.yolo\_size).to(device)  
 \_ = yolo\_model(dummy\_input)  
  
 # 无梯度计算的推理  
 with torch.no\_grad():  
 for \_, batch in enumerate(test\_queue):  
 # 根据是否有目标图像，解析 batch  
 if args.data\_path\_test\_target:  
 input\_tensor, target\_tensor, img\_name = batch  
 target\_tensor = target\_tensor.to(device)  
 else:  
 input\_tensor, img\_name = batch  
 target\_tensor = None  
 input\_tensor = input\_tensor.to(device)  
 input\_name = os.path.splitext(os.path.basename(img\_name[0]))[0]  
  
 # 执行模型推理  
 result = model(input\_tensor)  
 enhance\_tensor = result['H2'] # 增强图像张量  
 output\_tensor = result['H3'] # 去噪图像张量  
  
 # 应用自适应亮度控制后处理  
 output\_tensor = adaptive\_brightness\_control(output\_tensor)  
  
 # 估计噪声水平  
 noise\_level = estimate\_noise\_level(output\_tensor)  
  
 # 应用自适应降噪  
 if noise\_level > 0.05: # 仅在噪声水平较高时应用降噪  
 output\_tensor\_denoised = adaptive\_denoise(output\_tensor, noise\_level)  
 # 转换为Tensor  
 if isinstance(output\_tensor\_denoised, np.ndarray):  
 output\_tensor\_denoised = torch.from\_numpy(  
 np.transpose(output\_tensor\_denoised, (2, 0, 1))  
 ).unsqueeze(0).to(device)  
 output\_tensor = output\_tensor\_denoised  
  
 # 保存输出图像  
 enhance\_img = save\_images(enhance\_tensor)  
 output\_img = save\_images(output\_tensor)  
 os.makedirs(os.path.join(save\_path, 'result'), exist\_ok=True)  
 Image.fromarray(output\_img).save(os.path.join(save\_path, 'result', f'{input\_name}\_denoise.png'), 'PNG')  
 Image.fromarray(enhance\_img).save(os.path.join(save\_path, 'result', f'{input\_name}\_enhance.png'), 'PNG')  
  
 # 确保输出张量在0-1范围内  
 output\_tensor\_norm = torch.clamp(output\_tensor, 0, 1)  
 enhance\_tensor\_norm = torch.clamp(enhance\_tensor, 0, 1)  
  
 # 确保输出张量在0-1范围内  
 output\_tensor\_norm = torch.clamp(output\_tensor, 0, 1)  
 enhance\_tensor\_norm = torch.clamp(enhance\_tensor, 0, 1)  
  
 # 计算指标（调用封装函数）  
 metrics = calculate\_metrics(  
 enhanced=output\_tensor\_norm, # 最终输出的增强图像（去噪后）  
 target=target\_tensor, # 目标图像（可能为None）  
 device=device,  
 psnr\_metric=psnr\_metric,  
 ssim\_metric=ssim\_metric,  
 lpips\_model=lpips\_model,  
 niqe\_metric=niqe\_metric,  
 yolo\_model=yolo\_model,  
 noise\_level=noise\_level # 之前估计的噪声水平  
 )  
  
 # 记录指标到日志文件  
 with open(metric\_log\_path, 'a') as f:  
 if target\_tensor is not None:  
 f.write(  
 f"{input\_name}\_denoise.png, {metrics['psnr']:.4f}, {metrics['ssim']:.4f}, "  
 f"{metrics['lpips']:.4f}, {metrics['niqe']:.4f}, {metrics['noise\_level']:.4f}, "  
 f"{metrics['enhance\_map']:.4f}, {metrics['target\_map']:.4f}\n"  
 )  
 else:  
 f.write(  
 f"{input\_name}\_denoise.png, N/A, N/A, N/A, {metrics['niqe']:.4f}, {metrics['noise\_level']:.4f}, N/A, N/A\n")  
  
 # 打印指标到控制台/日志  
 if target\_tensor is not None:  
 logging.info(  
 f"Image {input\_name} - PSNR: {metrics['psnr']:.4f}, SSIM: {metrics['ssim']:.4f}, "  
 f"LPIPS: {metrics['lpips']:.4f}, NIQE: {metrics['niqe']:.4f}, Noise: {metrics['noise\_level']:.4f}, "  
 f"mAP(enhance): {metrics['enhance\_map']:.4f}, mAP(target): {metrics['target\_map']:.4f}"  
 )  
 else:  
 logging.info(  
 f"Image {input\_name} - NIQE: {metrics['niqe']:.4f}, Noise: {metrics['noise\_level']:.4f} (no target provided)")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# multi\_read\_data.py

import numpy as np  
import torch  
import torch.utils.data  
from PIL import Image  
import torchvision.transforms as transforms  
import os  
import logging  
  
# 配置日志  
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class DataLoader(torch.utils.data.Dataset):  
 def \_\_init\_\_(self, img\_dir, task, target\_dir=None):  
 self.target\_dir = target\_dir  
 self.low\_img\_dir = img\_dir  
 self.task = task  
 self.train\_low\_data\_names = []  
  
 # 收集图像路径  
 for root, dirs, names in os.walk(self.low\_img\_dir):  
 for name in names:  
 if name.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tiff')):  
 self.train\_low\_data\_names.append(os.path.join(root, name))  
 self.train\_low\_data\_names.sort()  
 self.count = len(self.train\_low\_data\_names)  
  
 # 定义图像转换  
 if self.task == 'train':  
 # 训练时：添加颜色增强及其他数据增强  
 # 注意：为了保持一致性，对输入和目标应用相同的随机变换  
 self.transform = transforms.Compose([  
 transforms.Resize((300, 200)),  
 transforms.RandomHorizontalFlip(p=0.5),  
  
 transforms.RandomVerticalFlip(p=0.3),  
 transforms.RandomRotation(15),  
 transforms.ColorJitter(  
 brightness=0.2,  
 contrast=0.2,  
 saturation=0.2,  
 hue=0.05  
 ),  
 transforms.RandomApply([transforms.GaussianBlur(kernel\_size=3)], p=0.1),  
 transforms.RandomGrayscale(p=0.01),  
 transforms.ToTensor()  
  
 ])  
 else:  
 # 测试时保持不变（不做增强，只转Tensor）  
 self.transform = transforms.Compose([  
 transforms.ToTensor()  
 ])  
  
 # 记录数据集统计信息  
 logging.info(f"Dataset initialized - Task: {task}, Low images: {self.count}")  
 if target\_dir:  
 logging.info(f"Target directory: {target\_dir}")  
  
 def load\_images\_transform(self, file):  
 im = Image.open(file).convert('RGB')  
 im = self.transform(im)  
 im = torch.clamp(im, 0.0, 1.0) # 确保数据在[0,1]范围内  
 return im  
  
  
 def \_\_getitem\_\_(self, index):  
 img\_path = self.train\_low\_data\_names[index]  
 img\_name = os.path.basename(img\_path)  
 low\_img = self.load\_images\_transform(img\_path)  
  
 # 记录图像加载信息  
 logging.debug(f"Loading image: {img\_path}, Shape: {low\_img.shape}")  
  
 # 添加目标图像加载逻辑  
 target\_img = None  
 if self.task == 'test' and self.target\_dir:  
 # 构建目标图像路径  
 target\_path = os.path.join(self.target\_dir, img\_name)  
 if os.path.exists(target\_path):  
 target\_img = self.load\_images\_transform(target\_path)  
 logging.debug(f"Loading target image: {target\_path}, Shape: {target\_img.shape}")  
 else:  
 # 如果找不到对应图像，创建全黑占位符并记录警告  
 logger.warning(f"Target image not found: {target\_path}. Using zero tensor.")  
 target\_img = torch.zeros\_like(low\_img)  
  
 if self.task == 'train' and self.target\_dir:  
 # 训练模式：使用相对路径构建目标图像路径  
 # 注意：这里假设低光图像和目标图像在相同目录结构下  
 rel\_path = os.path.relpath(img\_path, self.low\_img\_dir)  
 target\_path = os.path.join(self.target\_dir, rel\_path)  
 if os.path.exists(target\_path):  
 target\_img = self.load\_images\_transform(target\_path)  
 logging.debug(f"Loading target image: {target\_path}, Shape: {target\_img.shape}")  
 else:  
 logger.warning(f"Target image not found: {target\_path}. Using zero tensor.")  
 target\_img = torch.zeros\_like(low\_img)  
 return low\_img, target\_img, img\_name  
 elif self.task == 'test' and self.target\_dir:  
 # 测试模式返回目标图像  
 return low\_img, target\_img, img\_name  
 else:  
 # 如果没有提供目标目录，则只返回低光图像和名称  
 return low\_img, img\_name  
  
 def \_\_len\_\_(self):  
 return self.count

# model.py

import numpy as np  
import torch  
  
import torch.nn as nn  
import torch.nn.functional as F  
  
from utils import blur, pair\_downsampler # 导入工具函数：模糊处理、下采样  
from torch.utils.checkpoint import checkpoint  
from loss import LossFunction, TextureDifference, Discriminator # 导入损失函数相关类  
from utils import gauss\_kernel  
  
  
# 噪声分类器：识别噪声类型（高斯/泊松/椒盐）  
class NoiseClassifier(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.cnn = nn.Sequential(  
 nn.Conv2d(3, 16, 3, padding=1),  
 nn.ReLU(),  
 nn.MaxPool2d(2),  
 nn.Conv2d(16, 32, 3, padding=1),  
 nn.ReLU(),  
 nn.AdaptiveAvgPool2d(1),  
 nn.Flatten()  
 )  
 # 添加额外的全连接层  
 self.extra\_fc1 = nn.Linear(32, 16)  
 self.extra\_fc2 = nn.Linear(16, 3)  
  
 def forward(self, noise\_residual):  
 # 使用完整CNN序列  
 x = self.cnn(noise\_residual)  
 x = F.relu(self.extra\_fc1(x))  
 x = self.extra\_fc2(x)  
 prob = F.softmax(x, dim=1)  
 # 添加clone以防止inplace修改  
 return prob.clone()  
  
  
# 改进的IE-Net（Enhancer类），引入注意力模块  
class Enhancer(nn.Module):  
 def \_\_init\_\_(self, layers=8, channels=64):  
 super().\_\_init\_\_()  
 self.init\_conv = nn.Conv2d(5, channels, 3, padding=1)  
  
 # 添加空间-通道注意力模块  
 self.attention\_modules = nn.ModuleList()  
 for \_ in range(layers):  
 self.attention\_modules.append(AttentionModule(channels))  
  
 # 添加亮度约束模块  
 self.brightness\_control = nn.Sequential(  
 nn.AdaptiveAvgPool2d(1),  
 nn.Conv2d(channels, 32, 1), # 64→32（中间维度按比例增加）  
 nn.ReLU(),  
 nn.Conv2d(32, 1, 1),  
 nn.Sigmoid()  
 )  
 self.blocks = nn.ModuleList()  
 for i in range(layers):  
 self.blocks.append(nn.Sequential(  
 nn.Conv2d(channels, channels, 3, padding=1),  
 nn.ReLU(),  
 nn.Conv2d(channels, channels, 3, padding=1), # 增加一层卷积增强特征  
 nn.ReLU(),  
 self.attention\_modules[i] # 使用注意力模块  
 ))  
 self.final\_conv = nn.Sequential(  
 nn.Conv2d(channels, 3, 3, padding=1),  
 nn.Sigmoid() # 增加 Sigmoid 激活，将输出压缩到 [0, 1]  
 )  
  
 def forward(self, input, alpha\_pred, beta\_pred):  
 B, C, H, W = input.shape  
 alpha\_map = alpha\_pred.view(B, 1, 1, 1).expand(B, 1, H, W)  
 beta\_map = beta\_pred.view(B, 1, 1, 1).expand(B, 1, H, W)  
  
 conditioned\_input = torch.cat([input, alpha\_map, beta\_map], dim=1)  
 fea = self.init\_conv(conditioned\_input)  
  
 # 亮度控制  
 brightness\_factor = self.brightness\_control(fea)  
 brightness\_factor = torch.clamp(brightness\_factor, 0.8, 3.5) # 限制亮度调整范围  
  
 # 应用带注意力的块  
 for i, block in enumerate(self.blocks):  
 fea = fea + block(fea)  
 # 在特定层后应用注意力  
 if i % 2 == 1: # 每隔一层应用额外注意力  
 fea = self.attention\_modules[i](fea)  
  
 fea = self.final\_conv(fea)  
 brightness\_factor = torch.clamp(brightness\_factor, 1.5, 6.0) # 原0.8-3.5  
 fea = fea \* brightness\_factor  
 fea = torch.clamp(fea, 0, 1.0) # 允许轻微过曝（1.2），避免过暗  
  
 return fea  
  
  
# 动态参数预测器：根据亮度直方图和噪声水平预测α和β  
class DynamicParamPredictor(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 # 输入：亮度直方图（假设100 bins）+ 噪声水平（1个值）  
 self.fc = nn.Sequential(  
 nn.Linear(101, 64), # 100 bins + 1噪声水平  
 nn.ReLU(),  
 nn.Linear(64, 32),  
 nn.ReLU(),  
 nn.Linear(32, 2) # 输出α\_pred和β\_pred  
 )  
  
 def forward(self, hist, noise\_level):  
 # 确保输入在同一设备上  
 assert hist.device == noise\_level.device, "Hist and noise\_level must be on the same device"  
 # 输入处理：直方图展平 + 噪声水平拼接  
 hist\_flat = hist.view(hist.shape[0], -1) # [B, 100]  
 input\_feat = torch.cat([hist\_flat, noise\_level.unsqueeze(1)], dim=1) # [B, 101]  
 params = self.fc(input\_feat) # [B, 2]  
  
 # +++ 新增：约束alpha和beta为非负 +++  
 alpha\_pred\_raw, beta\_pred\_raw = params.split(1, dim=1)  
 alpha\_pred = torch.relu(alpha\_pred\_raw).squeeze(1) # 确保α≥0  
 beta\_pred = torch.relu(beta\_pred\_raw).squeeze(1) # 确保β≥0（避免负光照）  
 return alpha\_pred, beta\_pred  
  
  
  
  
  
# 多尺度空间-通道注意力模块  
class AttentionModule(nn.Module):  
 def \_\_init\_\_(self, channels):  
 super().\_\_init\_\_()  
 # 通道注意力分支  
 self.channel\_att = nn.Sequential(  
 nn.AdaptiveAvgPool2d(1),  
 nn.Conv2d(channels, channels // 4, 1, bias=False),  
 nn.ReLU(),  
 nn.Conv2d(channels // 4, channels, 1, bias=False),  
 nn.Sigmoid()  
 )  
 # 空间注意力分支  
 self.spatial\_att = nn.Sequential(  
 nn.Conv2d(2, 1, 3, padding=1, bias=False),  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 # 通道注意力  
 channel\_weight = self.channel\_att(x) # [B, C, 1, 1]  
 x = x \* channel\_weight  
 # 空间注意力  
 max\_pool = torch.max(x, dim=1, keepdim=True)[0]  
 avg\_pool = torch.mean(x, dim=1, keepdim=True)  
 spatial\_feat = torch.cat([max\_pool, avg\_pool], dim=1)  
 spatial\_weight = self.spatial\_att(spatial\_feat) # [B, 1, H, W]  
 x = x \* spatial\_weight  
 return x  
  
  
class Denoise\_1(nn.Module):  
 # 第一级去噪模块（轻量级卷积网络）  
 def \_\_init\_\_(self, chan\_embed=48):  
 super(Denoise\_1, self).\_\_init\_\_()  
 self.act = nn.LeakyReLU(negative\_slope=0.2, inplace=True)  
 self.conv1 = nn.Conv2d(3, chan\_embed, 3, padding=1)  
 self.conv2 = nn.Conv2d(chan\_embed, chan\_embed, 3, padding=1)  
 self.conv3 = nn.Conv2d(chan\_embed, 3, 1)  
  
 def forward(self, x):  
 x = checkpoint(self.conv1, x, use\_reentrant=False)  
 x = self.act(x)  
 x = checkpoint(self.conv2, x, use\_reentrant=False)  
 x = self.act(x)  
 x = checkpoint(self.conv3, x, use\_reentrant=False)  
 return x  
  
  
# +++ 添加：Transformer编码器（全局特征建模） +++  
class TransformerEncoder(nn.Module):  
 def \_\_init\_\_(self, embed\_dim, max\_seq\_len=16384):  
 super().\_\_init\_\_()  
 self.attention = nn.MultiheadAttention(embed\_dim=embed\_dim, num\_heads=4, batch\_first=True)  
 # 添加可学习的位置编码  
 self.pos\_encoding = nn.Parameter(torch.zeros(1, max\_seq\_len, embed\_dim))  
  
 def forward(self, x):  
 # x shape:  
 seq\_len = x.size(1)  
 # 添加位置编码  
 x = x + self.pos\_encoding[:, :seq\_len, :]  
 attn\_output, \_ = self.attention(x, x, x)  
 return attn\_output  
  
  
# +++ 修改：改进的RD-Net（混合架构，替代原Denoise\_2） +++  
class Denoise2(nn.Module):  
 def \_\_init\_\_(self, channels=64):  
 super().\_\_init\_\_()  
 # CNN局部特征提取  
 self.texture\_extractor = nn.Sequential(  
 nn.Conv2d(3, channels // 2, 3, padding=1),  
 nn.LeakyReLU(0.2, inplace=True),  
 nn.Conv2d(channels // 2, channels // 2, 3, padding=1),  
 nn.LeakyReLU(0.2, inplace=True)  
 )  
 self.texture\_proj = nn.Conv2d(channels // 2, channels, 1) # 1x1卷积调整通道数  
 self.cnn = nn.Sequential(  
 nn.Conv2d(6, channels, 3, padding=1), # 输入：反射图(3)+光照图(3)  
 nn.LeakyReLU(negative\_slope=0.2, inplace=True),  
 nn.Conv2d(channels, channels, 3, padding=1),  
 nn.LeakyReLU(negative\_slope=0.2, inplace=True),  
 nn.Conv2d(channels, channels, 3, padding=1)  
 )  
 self.down\_ratio = 4 # 从8减少到4，保留更多细节  
 self.transformer\_norm = nn.LayerNorm(channels)  
 self.transformer = TransformerEncoder(embed\_dim=channels, max\_seq\_len=16384)  
 self.fusion = nn.Conv2d(channels \* 2, channels, 1)  
 self.attn = AttentionModule(channels)  
 self.final\_conv = nn.Conv2d(channels, 6, 1)  
 self.noise\_classifier = NoiseClassifier()  
 self.gauss\_conv = nn.Conv2d(channels, 6, 1)  
 self.poisson\_conv = nn.Conv2d(channels, 6, 1)  
 self.salt\_conv = nn.Conv2d(channels, 6, 1)  
  
 def \_gaussian\_blur(self, x, kernel\_size=3, sigma=1.0):  
 channels = x.shape[1]  
 kernel = gauss\_kernel(kernel\_size, sigma, channels, device=x.device)  
 padding = kernel\_size // 2  
 return F.conv2d(x, kernel, padding=padding, groups=channels)  
  
 def \_resize\_if\_needed(self, tensor, target):  
 if tensor.shape[2:] != target.shape[2:]:  
 return F.interpolate(tensor, size=target.shape[2:], mode='bilinear', align\_corners=False)  
 return tensor  
  
 def forward(self, r, s, noise\_residual):  
 noise\_prob = self.noise\_classifier(noise\_residual)  
 gauss\_prob, poisson\_prob, salt\_prob = noise\_prob[:, 0], noise\_prob[:, 1], noise\_prob[:, 2]  
  
 texture\_feat = self.texture\_extractor(r)  
 texture\_feat = self.texture\_proj(texture\_feat)  
 texture\_feat = self.\_resize\_if\_needed(texture\_feat, r)  
  
 x = torch.cat([r, s], dim=1)  
 cnn\_feat = self.cnn(x)  
 cnn\_feat = cnn\_feat + 0.3 \* texture\_feat  
  
 cnn\_feat\_down = F.avg\_pool2d(cnn\_feat, kernel\_size=self.down\_ratio, stride=self.down\_ratio)  
  
 B, C, H\_down, W\_down = cnn\_feat\_down.shape  
 seq\_len = H\_down \* W\_down  
 max\_seq\_len = self.transformer.pos\_encoding.size(1)  
  
 if seq\_len > max\_seq\_len:  
 additional\_down\_ratio = int(np.ceil(np.sqrt(seq\_len / max\_seq\_len)))  
 cnn\_feat\_down = F.avg\_pool2d(cnn\_feat\_down, kernel\_size=additional\_down\_ratio, stride=additional\_down\_ratio)  
  
 transformer\_input = cnn\_feat\_down.flatten(2).permute(0, 2, 1)  
 transformer\_feat = self.transformer(transformer\_input)  
  
 B, \_, C = transformer\_feat.shape  
 H\_new, W\_new = H\_down, W\_down  
  
 transformer\_feat = transformer\_feat.permute(0, 2, 1).reshape(B, C, H\_new, W\_new)  
  
 transformer\_feat = F.interpolate(transformer\_feat, size=cnn\_feat.shape[2:], mode='bilinear',  
 align\_corners=False)  
  
 B, C, H, W = transformer\_feat.shape  
 transformer\_feat\_norm\_input = transformer\_feat.reshape(B, C, H \* W).permute(0, 2, 1)  
 transformer\_feat\_norm = self.transformer\_norm(transformer\_feat\_norm\_input)  
 transformer\_feat = transformer\_feat\_norm.permute(0, 2, 1).reshape(B, C, H, W)  
  
 fused = self.fusion(torch.cat([cnn\_feat, transformer\_feat], dim=1))  
 fused = self.attn(fused)  
  
 gauss\_out = self.gauss\_conv(fused)  
 poisson\_feat = torch.sqrt(F.relu(fused) + 1e-6)  
 poisson\_out = self.poisson\_conv(poisson\_feat)  
 salt\_mid = self.\_gaussian\_blur(fused, kernel\_size=3, sigma=1.0)  
 salt\_out = self.salt\_conv(salt\_mid)  
  
 gauss\_out = self.\_resize\_if\_needed(gauss\_out, r)  
 poisson\_out = self.\_resize\_if\_needed(poisson\_out, r)  
 salt\_out = self.\_resize\_if\_needed(salt\_out, r)  
  
 gauss\_weight = gauss\_prob.view(-1, 1, 1, 1)  
 poisson\_weight = poisson\_prob.view(-1, 1, 1, 1)  
 salt\_weight = salt\_prob.view(-1, 1, 1, 1)  
  
 total\_weight = gauss\_weight + poisson\_weight + salt\_weight + 1e-6  
 gauss\_weight = gauss\_weight / total\_weight  
 poisson\_weight = poisson\_weight / total\_weight  
 salt\_weight = salt\_weight / total\_weight  
  
 combined = (gauss\_out \* gauss\_weight + poisson\_out \* poisson\_weight + salt\_out \* salt\_weight)  
 return torch.clamp(combined, 0, 1.0)  
# 可学习亮度校正模块（替代启发式亮度调整）  
class LearnableBrightnessCorrection(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 # 可学习的全局缩放因子和偏移量（初始值设为1.0和0.0，即不改变原始亮度）  
 self.scale = nn.Parameter(torch.tensor(1.2))  
 self.shift = nn.Parameter(torch.tensor(0.05))  
  
 def forward(self, x):  
 # 仿射变换调整亮度，确保输出在[0,1]范围内  
 corrected = x \* self.scale + self.shift  
 return torch.clamp(corrected, 0.0, 1.0)  
  
  
# +++ 增强版颜色校正模块 +++  
class EnhancedColorCorrection(nn.Module):  
 def \_\_init\_\_(self, channels=64):  
 super().\_\_init\_\_()  
 self.conv = nn.Sequential(  
 nn.Conv2d(6, channels, 3, padding=1),  
 nn.ReLU(),  
 nn.Conv2d(channels, channels, 3, padding=1),  
 nn.ReLU(),  
 nn.Conv2d(channels, channels, 3, padding=1),  
 nn.ReLU(),  
 nn.Conv2d(channels, 3, 1)  
 )  
 self.attention = nn.Sequential(  
 nn.AdaptiveAvgPool2d(1),  
 nn.Conv2d(3, 16, 1),  
 nn.ReLU(),  
 nn.Conv2d(16, 3, 1),  
 nn.Sigmoid()  
 )  
  
 def forward(self, input, enhanced):  
 # 确保输入在有效范围内  
 input = torch.clamp(input, 0, 1)  
 enhanced = torch.clamp(enhanced, 1e-4, 1) # 避免除零  
  
 # 连接特征  
 concat\_feat = torch.cat([input, enhanced], dim=1)  
  
 # 应用校正  
 correction = self.conv(concat\_feat)  
 attn\_map = self.attention(input)  
 corrected = enhanced + correction \* attn\_map  
  
 # 确保输出在有效范围内  
 return torch.clamp(corrected, 0, 1)  
  
  
# 相机响应函数校正模块  
  
class CRFCorrection(nn.Module):  
 """相机响应函数校正模块"""  
 def \_\_init\_\_(self, init\_gamma=0.45, learnable=True):  
 super().\_\_init\_\_()  
 if learnable:  
 self.gamma = nn.Parameter(torch.tensor(init\_gamma))  
 else:  
 self.register\_buffer('gamma', torch.tensor(init\_gamma))  
  
 # 可学习的色调映射曲线  
 # 使用 LeakyReLU 替代 ReLU，防止梯度消失  
 self.curve = nn.Sequential(  
 nn.Linear(1, 8),  
 nn.LeakyReLU(0.01, inplace=True), # 替换 nn.ReLU()  
 nn.Linear(8, 1),  
 nn.Sigmoid()  
 )  
 # 初始化曲线网络的权重  
 self.\_initialize\_weights()  
  
 def \_initialize\_weights(self):  
 """  
 专门为这个小MLP设计初始化。  
 使用Xavier初始化对于Linear层搭配LeakyReLU是较好的选择。  
 """  
 for m in self.curve.modules():  
 if isinstance(m, nn.Linear):  
 nn.init.xavier\_uniform\_(m.weight)  
 # 将偏置初始化为一个小的正值，增加初始阶段的活性  
 nn.init.constant\_(m.bias, 0.1)  
  
 def forward(self, x):  
 # Gamma校正  
 x\_gamma = torch.pow(x, self.gamma)  
  
 # 可学习的曲线调整（逐像素）  
 B, C, H, W = x.shape  
 x\_flat = x.reshape(B \* C \* H \* W, 1)  
 x\_curve = self.curve(x\_flat).reshape(B, C, H, W)  
  
 # 混合输出  
 return 0.7 \* x\_gamma + 0.3 \* x\_curve  
  
  
  
  
class Network(nn.Module):  
 # 主网络（训练时使用）  
 def \_\_init\_\_(self, debug=False):  
 super(Network, self).\_\_init\_\_()  
 self.debug = debug # 调试模式标志  
 self.enhance = Enhancer(layers=8, channels=64) # 增强模块  
 self.denoise\_1 = Denoise\_1(chan\_embed=16) # 第一级去噪  
 self.denoise\_2 = Denoise2(channels=64) # 第二级去噪  
 self.param\_predictor = DynamicParamPredictor() # 动态参数预测器  
 self.noise\_classifier = NoiseClassifier() # 噪声分类器  
 # 判别器及损失函数  
 self.discriminator = Discriminator()  
 self.\_criterion = LossFunction()  
 # 其它辅助模块  
 self.avgpool = nn.AvgPool2d(kernel\_size=3, stride=1, padding=1)  
 self.TextureDifference = TextureDifference()  
 self.color\_correct = EnhancedColorCorrection() # 使用增强版颜色校正模块  
 # 添加CRF校正模块  
 self.crf\_correction = CRFCorrection(learnable=True)  
 self.brightness\_correction = LearnableBrightnessCorrection()  
  
 # 添加中间层监控  
 self.intermediate\_outputs = {}  
 self.\_register\_hooks()  
 # 检查参数初始化  
 for name, param in self.named\_parameters():  
 if torch.isnan(param).any() or torch.isinf(param).any():  
 print(f"参数 {name} 包含NaN或Inf值，重新初始化")  
 nn.init.xavier\_uniform\_(param.data)  
  
 def \_register\_hooks(self):  
 """注册前向钩子来监控中间层输出"""  
  
 def get\_activation(name):  
 def hook(model, input, output):  
 self.intermediate\_outputs[name] = {  
 'min': output.min().item(),  
 'max': output.max().item(),  
 'mean': output.mean().item(),  
 'std': output.std().item()  
 }  
  
 return hook  
  
 # 监控关键层  
 layers\_to\_monitor = {  
 'enhance.init\_conv': self.enhance.init\_conv,  
 'enhance.blocks.0': self.enhance.blocks[0],  
 'denoise\_1.conv1': self.denoise\_1.conv1,  
 'denoise\_1.conv2': self.denoise\_1.conv2,  
 'denoise\_1.conv3': self.denoise\_1.conv3,  
 'denoise\_2.cnn': self.denoise\_2.cnn,  
 'denoise\_2.transformer': self.denoise\_2.transformer,  
 'param\_predictor.fc.0': self.param\_predictor.fc[0],  
 'param\_predictor.fc.2': self.param\_predictor.fc[2],  
 'param\_predictor.fc.4': self.param\_predictor.fc[4],  
 'noise\_classifier.cnn': self.noise\_classifier.cnn,  
 'color\_correct.conv.0': self.color\_correct.conv[0],  
 'color\_correct.conv.3': self.color\_correct.conv[3],  
 'color\_correct.conv.6': self.color\_correct.conv[6],  
 'crf\_correction.curve.0': self.crf\_correction.curve[0],  
 'crf\_correction.curve.2': self.crf\_correction.curve[2],  
 }  
  
 for name, layer in layers\_to\_monitor.items():  
 layer.register\_forward\_hook(get\_activation(name))  
  
 def get\_intermediate\_outputs(self):  
 """获取中间层输出信息"""  
 return self.intermediate\_outputs  
  
 def enhance\_weights\_init(self, m):  
 if isinstance(m, nn.Conv2d):  
 m.weight.data.normal\_(0.0, 0.02)  
 if m.bias is not None: # 修正为 bias  
 m.bias.data.zero\_()  
 if isinstance(m, nn.BatchNorm2d):  
 m.weight.data.normal\_(1, 0.02)  
  
 def denoise\_weights\_init(self, m):  
 if isinstance(m, nn.Conv2d):  
 m.weight.data.normal\_(0, 0.02)  
 if m.bias is not None: # 修正为 bias  
 m.bias.data.zero\_()  
 if isinstance(m, nn.BatchNorm2d):  
 m.weight.data.normal\_(1, 0.02)  
  
 def \_compute\_brightness\_histogram(self, x, bins=100):  
 """计算输入图像的亮度直方图（转为灰度后计算）"""  
 gray = 0.299 \* x[:, 0] + 0.587 \* x[:, 1] + 0.114 \* x[:, 2] # [B, H, W]  
 hist\_list = []  
 for i in range(gray.shape[0]):  
 hist = torch.histc(gray[i], bins=bins, min=0, max=1)  
 hist = hist / (gray.shape[1] \* gray.shape[2])  
 hist\_list.append(hist)  
 return torch.stack(hist\_list, dim=0) # [B, 100]  
  
 def \_estimate\_noise\_level(self, x):  
 """简单估计噪声水平（输入图像与模糊版本的差异）"""  
 x\_blur = blur(x)  
 noise = x - x\_blur  
 return torch.mean(torch.abs(noise), dim=[1, 2, 3]) # [B]  
  
 def \_debug\_print(self, name, tensor):  
 """调试打印函数"""  
 if self.debug:  
 print(f"{name}: shape={tensor.shape}, min={tensor.min().item():.4f}, "  
 f"max={tensor.max().item():.4f}, mean={tensor.mean().item():.4f}, "  
 f"has\_nan={torch.isnan(tensor).any().item()}, "  
 f"has\_inf={torch.isinf(tensor).any().item()}")  
  
 def forward(self, input):  
 # 清空中间层输出记录  
 self.intermediate\_outputs = {}  
  
 outputs = {} # 初始化字典  
 eps = 1e-4  
 input = input + eps  
 input.requires\_grad\_(True)  
  
 # 调试输入  
 self.\_debug\_print("Input", input)  
  
 # 计算亮度直方图和噪声水平  
 brightness\_hist = self.\_compute\_brightness\_histogram(input)  
 noise\_level = self.\_estimate\_noise\_level(input)  
 alpha\_pred, beta = self.param\_predictor(brightness\_hist, noise\_level)  
  
 # 调试参数预测器  
 if self.debug:  
 print(f"Alpha\_pred: {alpha\_pred.mean().item():.4f}, Beta: {beta.mean().item():.4f}")  
  
 # 第一级去噪  
 noise\_residual = checkpoint(self.denoise\_1, input, use\_reentrant=False)  
 self.\_debug\_print("Noise\_residual", noise\_residual)  
  
 noise\_prob = self.noise\_classifier(noise\_residual)  
 if self.debug:  
 print(f"Noise\_prob: {noise\_prob.mean(dim=0)}")  
  
 # 下采样输入图像（构建多尺度）  
 L11, L12 = pair\_downsampler(input)  
 L\_pred1 = L11 - checkpoint(self.denoise\_1, L11, use\_reentrant=False)  
 L\_pred2 = L12 - checkpoint(self.denoise\_1, L12, use\_reentrant=False)  
 L2 = input - noise\_residual  
 L2 = torch.clamp(L2, eps, 1)  
 self.\_debug\_print("L2", L2)  
  
 # 增强模块生成光照图  
 s2 = checkpoint(self.enhance, L2, alpha\_pred, beta, use\_reentrant=False) # 传入两个参数  
 # CRF校正  
 s2 = self.crf\_correction(s2)  
 s2 = torch.clamp(s2, 0.01, 1)  
 self.\_debug\_print("s2", s2)  
  
 s21, s22 = pair\_downsampler(s2)  
  
 # +++ 关键修改：确保H2反射图被钳位在[0,1]范围内 +++  
 s2\_clamped = torch.clamp(s2, min=0.01) # 使用一个更大、更安全的最小值  
 H2 = input / (s2\_clamped + 1e-6)  
 H2 = torch.clamp(H2, 0, 1.0) # 同时将输出也钳位到有效范围内  
 self.\_debug\_print("H2", H2)  
  
 # 对增强后的反射图H2进行颜色校正（输入原图input和增强图H2）  
 H2\_color = self.color\_correct(input, H2)  
 H2\_color = torch.clamp(H2\_color, 0, 1.0)  
 outputs['H2\_color'] = H2\_color # 将校正结果加入输出  
 self.\_debug\_print("H2\_color", H2\_color)  
  
 # 多尺度增强的反射图  
 s21\_clamped = torch.clamp(s21, min=0.01)  
 H11 = L11 / s21\_clamped  
 H11 = torch.clamp(H11, 0, 1.0)  
  
 s22\_clamped = torch.clamp(s22, min=0.01)  
 H12 = L12 / s22\_clamped  
 H12 = torch.clamp(H12, 0, 1.0)  
  
 # 第二级去噪（多尺度输入）  
 H3\_pred = self.denoise\_2(H11, s21, noise\_residual)  
 H3\_pred = torch.clamp(H3\_pred, eps, 1)  
 self.\_debug\_print("H3\_pred", H3\_pred)  
  
 H13 = H3\_pred[:, :3, :, :]  
 s13 = H3\_pred[:, 3:, :, :]  
 H4\_pred = self.denoise\_2(H12, s22, noise\_residual)  
 H4\_pred = torch.clamp(H4\_pred, eps, 1)  
 self.\_debug\_print("H4\_pred", H4\_pred)  
  
 H14 = H4\_pred[:, :3, :, :]  
 s14 = H4\_pred[:, 3:, :, :]  
 H5\_pred = self.denoise\_2(H2, s2, noise\_residual)  
 H5\_pred = torch.clamp(H5\_pred, eps, 1)  
 self.\_debug\_print("H5\_pred", H5\_pred)  
  
 H3 = H5\_pred[:, :3, :, :]  
 enhanced\_final = H3 \* s2 # 这是最终的增强结果  
 enhanced\_final = torch.clamp(enhanced\_final, 1e-4, 1.0)  
 s3 = H5\_pred[:, 3:, :, :]  
 # 应用可学习亮度校正（新增代码）  
 enhanced\_final = self.brightness\_correction(enhanced\_final)  
 # 更新H3为校正后的值（如果H3作为最终输出）  
 H3 = enhanced\_final  
  
 self.\_debug\_print("H3 (final output)", H3) # 确认H3来自Denoise2  
  
 # 纹理差异计算（用于损失）  
 L\_pred1\_L\_pred2\_diff = self.TextureDifference(L\_pred1, L\_pred2)  
 H3\_denoised1, H3\_denoised2 = pair\_downsampler(H3)  
 H3\_denoised1\_H3\_denoised2\_diff = self.TextureDifference(H3\_denoised1, H3\_denoised2)  
  
 # 计算模糊版本（用于颜色一致性损失）  
 H1 = L2 / (s2 + 1e-8)  
 H1 = torch.clamp(H1, 0, 1)  
 H2\_blur = blur(H1)  
 H3\_blur = blur(H3)  
  
 # 明确主输出和辅助输出  
 return {  
 'enhanced': enhanced\_final, # 作为主输出  
 'illumination': s2, # 光照图作为辅助输出  
 'denoised': H3, # 去噪结果  
 'L\_pred1': L\_pred1,  
 'L\_pred2': L\_pred2,  
 'L2': L2,  
 's2': s2,  
 's21': s21,  
 's22': s22,  
 'H2': H2,  
 'H2\_color': H2\_color, # 颜色校正后的增强图  
 'H11': H11,  
 'H12': H12,  
 'H13': H13,  
 's13': s13,  
 'H14': H14,  
 's14': s14,  
 'H3\_denoised1': H3\_denoised1,  
 'H3\_denoised2': H3\_denoised2,  
 'H3': H3, # 最终输出，来自Denoise2  
 's3': s3,  
 'H3\_pred': H3\_pred,  
 'H4\_pred': H4\_pred,  
 'L\_pred1\_L\_pred2\_diff': L\_pred1\_L\_pred2\_diff,  
 'H3\_denoised1\_H3\_denoised2\_diff': H3\_denoised1\_H3\_denoised2\_diff,  
 'H2\_blur': H2\_blur,  
 'H3\_blur': H3\_blur,  
 'alpha\_pred': alpha\_pred,  
 'beta\_pred': beta,  
 'noise\_prob': noise\_prob,  
 'noise\_residual': noise\_residual  
 }  
  
 def \_loss(self, input, target, epoch=0, \*\*outputs):  
 # 计算总损失（组合多种损失项）  
 return self.\_criterion(input, target, epoch=epoch, \*\*outputs)  
  
  
class Finetunemodel(nn.Module):  
 # 微调模型（测试时使用）  
 def \_\_init\_\_(self, weights, debug=False):  
 super(Finetunemodel, self).\_\_init\_\_()  
 self.debug = debug # 调试模式标志  
 self.enhance = Enhancer(layers=8, channels=64)  
 self.denoise\_1 = Denoise\_1(chan\_embed=32)  
 self.denoise\_2 = Denoise2(channels=64)  
 self.param\_predictor = DynamicParamPredictor()  
 self.noise\_classifier = NoiseClassifier()  
 # +++ 添加增强版颜色校正模块 +++  
 self.color\_correct = EnhancedColorCorrection()  
 # 添加CRF校正模块  
 self.crf\_correction = CRFCorrection(learnable=True)  
 # 加载预训练权重  
 base\_weights = torch.load(weights, map\_location='cpu')  
 model\_dict = self.state\_dict()  
 pretrained\_dict = {k: v for k, v in base\_weights.items() if k in model\_dict}  
 model\_dict.update(pretrained\_dict)  
 self.load\_state\_dict(model\_dict)  
 # 添加TTA标志  
 self.use\_tta = False # 默认启用测试时增强  
  
 def \_debug\_print(self, name, tensor):  
 """调试打印函数"""  
 if self.debug:  
 print(f"{name}: shape={tensor.shape}, min={tensor.min().item():.4f}, "  
 f"max={tensor.max().item():.4f}, mean={tensor.mean().item():.4f}, "  
 f"has\_nan={torch.isnan(tensor).any().item()}, "  
 f"has\_inf={torch.isinf(tensor).any().item()}")  
  
 def \_compute\_brightness\_histogram(self, x, bins=100):  
 """计算输入图像的亮度直方图（转为灰度后计算）"""  
 gray = 0.299 \* x[:, 0] + 0.587 \* x[:, 1] + 0.114 \* x[:, 2] # [B, H, W]  
 hist\_list = []  
 for i in range(gray.shape[0]):  
 # 先将数据移动到CPU计算直方图，然后再移回原设备  
 hist = torch.histc(gray[i].cpu(), bins=bins, min=0, max=1)  
 hist = hist.to(x.device) # 移回GPU  
 hist = hist / (gray.shape[1] \* gray.shape[2])  
 hist\_list.append(hist)  
 return torch.stack(hist\_list, dim=0) # [B, 100]  
  
 def \_estimate\_noise\_level(self, x):  
 """估计噪声水平（输入与模糊图之差）"""  
 x\_blur = blur(x)  
 noise = x - x\_blur  
 return torch.mean(torch.abs(noise), dim=[1, 2, 3])  
  
 def \_forward\_impl(self, input):  
 """单次前向传播实现"""  
 eps = 1e-4  
 input = input + eps  
 # 调试输入  
 self.\_debug\_print("Input", input)  
  
 # 使用梯度检查点包装计算密集型操作  
 def compute\_features(x):  
 brightness\_hist = self.\_compute\_brightness\_histogram(x)  
 noise\_level = self.\_estimate\_noise\_level(x)  
 return self.param\_predictor(brightness\_hist, noise\_level)  
  
 alpha\_pred, beta = checkpoint(compute\_features, input)  
  
 # 调试参数预测器  
 if self.debug:  
 print(f"Alpha\_pred: {alpha\_pred.mean().item():.4f}, Beta: {beta.mean().item():.4f}")  
  
 # 第一级去噪与噪声分类  
 noise\_residual = self.denoise\_1(input)  
 self.\_debug\_print("Noise\_residual", noise\_residual)  
  
 noise\_prob = self.noise\_classifier(noise\_residual)  
 if self.debug:  
 print(f"Noise\_prob: {noise\_prob.mean(dim=0)}")  
  
 # 计算去噪后图像  
 L2 = input - noise\_residual  
 L2 = torch.clamp(L2, eps, 1)  
 self.\_debug\_print("L2", L2)  
  
 # 增强模块生成光照图  
 s2 = checkpoint(self.enhance, L2, alpha\_pred, beta)  
 # CRF校正  
 s2 = self.crf\_correction(s2)  
 s2 = torch.clamp(s2, eps, 1)  
 self.\_debug\_print("s2", s2)  
  
 # 计算增强后的反射图  
 H2 = input / (s2 + 1e-8)  
 H2 = torch.clamp(H2, 0, 1.2) # 修改：从[0,2]改为[0,1]  
 self.\_debug\_print("H2", H2)  
  
 # 第二级去噪（RD-Net）  
 H5\_pred = checkpoint(self.denoise\_2, H2, s2, noise\_residual)  
 H5\_pred = torch.clamp(H5\_pred, eps, 1)  
 self.\_debug\_print("H5\_pred", H5\_pred)  
  
 H3 = H5\_pred[:, :3, :, :] # 最终去噪结果  
 self.\_debug\_print("H3 (final output)", H3) # 确认H3来自Denoise2  
  
 # +++ 添加后处理：通道调整 +++  
 # 蓝色通道增强（索引2为蓝色通道）  
 H3[:, 2] = H3[:, 2] \* 1.05  
 # 红色通道减弱（索引0为红色通道）  
 H3[:, 0] = H3[:, 0] \* 0.97  
 # 确保数值仍在[0, 1]范围内  
 H3 = torch.clamp(H3, eps, 1)  
 self.\_debug\_print("H3 (after color adjustment)", H3)  
  
 # 返回与训练阶段对应的输出  
 return {  
 'enhanced': H3, # 主输出  
 'illumination': s2, # 辅助输出  
 'H2': H2, # 增强图  
 'H3': H3, # 处理后的去噪图  
 'alpha\_pred': alpha\_pred,  
 'beta\_pred': beta,  
 'noise\_prob': noise\_prob,  
 'noise\_residual': noise\_residual  
 }  
  
 def forward(self, input):  
 if not self.use\_tta:  
 # 不使用TTA，直接返回单次前向传播结果  
 return self.\_forward\_impl(input)  
  
 # 测试时增强（TTA） - 多尺度融合  
 # 原始尺度  
 output1 = self.\_forward\_impl(input)  
  
 # 水平翻转  
 output2 = self.\_forward\_impl(torch.flip(input, [3]))  
 output2['H2'] = torch.flip(output2['H2'], [3])  
 output2['H3'] = torch.flip(output2['H3'], [3])  
 output2['enhanced'] = torch.flip(output2['enhanced'], [3])  
  
 # 垂直翻转  
 output3 = self.\_forward\_impl(torch.flip(input, [2]))  
 output3['H2'] = torch.flip(output3['H2'], [2])  
 output3['H3'] = torch.flip(output3['H3'], [2])  
 output3['enhanced'] = torch.flip(output3['enhanced'], [2])  
  
 # 多尺度平均  
 final\_output = {}  
 for key in output1.keys():  
 if isinstance(output1[key], torch.Tensor) and output1[key].dim() == 4:  
 # 对图像输出进行平均  
 final\_output[key] = (output1[key] + output2[key] + output3[key]) / 3  
 else:  
 # 对其他输出保持原始值  
 final\_output[key] = output1[key]  
  
 return final\_output

# loss.py

import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import numpy as np  
import scipy.stats as st  
import timm  
from torchvision.models import vgg19, VGG19\_Weights  
from utils import pair\_downsampler, calculate\_local\_variance, LocalMean, gauss\_kernel # 导入工具函数  
from torch.nn.utils import spectral\_norm  
# 尝试导入LPIPS，如果不可用则回退到VGG  
try:  
 import lpips  
  
 LPIPS\_AVAILABLE = True  
except ImportError:  
 LPIPS\_AVAILABLE = False  
 print("LPIPS not available, using VGG-based perceptual loss")  
  
EPS = 1e-9 # 防止除零  
PI = 22.0 / 7.0 # 圆周率近似值  
  
  
# 多尺度SSIM损失  
class MultiScaleSSIMLoss(nn.Module):  
 def \_\_init\_\_(self, weights=None):  
 super().\_\_init\_\_()  
 self.weights = weights or [0.5, 0.3, 0.2] # 多尺度权重  
  
 def forward(self, pred, target):  
 loss = 0  
 for i, scale in enumerate([1.0, 0.5, 0.25]): # 全尺度、半尺度、1/4尺度  
 if scale != 1.0:  
 pred\_scale = F.interpolate(pred, scale\_factor=scale, mode='bilinear')  
 target\_scale = F.interpolate(target, scale\_factor=scale, mode='bilinear')  
 else:  
 pred\_scale, target\_scale = pred, target  
  
 ssim\_loss = 1 - self.ssim(pred\_scale, target\_scale)  
 loss += ssim\_loss \* self.weights[i]  
 return loss  
  
 def ssim(self, pred, target, window\_size=11, size\_average=True):  
 # 简化版SSIM实现  
 C1 = 0.01 \*\* 2  
 C2 = 0.03 \*\* 2  
  
 mu1 = F.avg\_pool2d(pred, window\_size, 1, 0)  
 mu2 = F.avg\_pool2d(target, window\_size, 1, 0)  
  
 mu1\_sq = mu1.pow(2)  
 mu2\_sq = mu2.pow(2)  
 mu1\_mu2 = mu1 \* mu2  
  
 sigma1\_sq = F.avg\_pool2d(pred \* pred, window\_size, 1, 0) - mu1\_sq  
 sigma2\_sq = F.avg\_pool2d(target \* target, window\_size, 1, 0) - mu2\_sq  
 sigma12 = F.avg\_pool2d(pred \* target, window\_size, 1, 0) - mu1\_mu2  
  
 ssim\_map = ((2 \* mu1\_mu2 + C1) \* (2 \* sigma12 + C2)) / ((mu1\_sq + mu2\_sq + C1) \* (sigma1\_sq + sigma2\_sq + C2))  
  
 if size\_average:  
 return ssim\_map.mean()  
 else:  
 return ssim\_map.mean(1).mean(1).mean(1)  
  
  
# loss.py 中 ImprovedPerceptualLoss 类修改  
class ImprovedPerceptualLoss(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.lpips\_available = LPIPS\_AVAILABLE  
 self.adaptive\_pool = nn.AdaptiveAvgPool2d(1) # 提前定义池化层  
 if self.lpips\_available:  
 # LPIPS可用时，仅初始化LPIPS，延迟VGG19  
 self.lpips = lpips.LPIPS(net='vgg')  
 self.vgg = None  
 self.slice1 = None  
 self.slice2 = None  
 else:  
 # LPIPS不可用时，也延迟VGG19初始化  
 self.lpips = None  
 self.vgg = None  
 self.slice1 = None  
 self.slice2 = None  
  
 def forward(self, pred, target):  
 if self.lpips\_available:  
 # 仅在计算时将LPIPS移到GPU，用完移回CPU  
 self.lpips.to(pred.device)  
 # +++ 归一化输入到[-1,1] +++  
 pred\_lpips = 2 \* pred - 1 # [0,1] → [-1,1]  
 target\_lpips = 2 \* target - 1  
 loss = self.lpips(pred\_lpips, target\_lpips).mean()  
 return loss  
 else:  
 if self.vgg is None:  
 # 延迟初始化并仅在需要时加载  
 self.vgg = vgg19(weights=VGG19\_Weights.IMAGENET1K\_V1).features.to(pred.device)  
 for param in self.vgg.parameters():  
 param.requires\_grad = False  
 self.slice1 = nn.Sequential(\*list(self.vgg[:2])).to(pred.device)  
 # self.slice2 = nn.Sequential(\*list(self.vgg[2:7])).to(pred.device) # 示例：取2-6层  
  
 # 标准化处理（与原逻辑一致）  
 mean = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1).to(pred.device)  
 std = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1).to(pred.device)  
 pred = (pred - mean) / std  
 target = (target - mean) / std  
  
 # 提取特征并计算损失（与原逻辑一致）  
 features = []  
 pred\_feat = self.slice1(pred)  
 target\_feat = self.slice1(target)  
 features.append((self.adaptive\_pool(pred\_feat), self.adaptive\_pool(target\_feat)))  
  
 # pred\_feat = self.slice2(pred\_feat)  
 # target\_feat = self.slice2(target\_feat)  
 # features.append((self.adaptive\_pool(pred\_feat), self.adaptive\_pool(target\_feat)))  
  
 loss = 0  
 for (p, t) in features:  
 loss += F.mse\_loss(p, t)  
  
 return loss  
  
  
# 频率域损失  
class FrequencyLoss(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def forward(self, pred, target):  
 # 计算DCT变换后的差异  
 pred\_dct = torch.fft.fft2(pred, dim=(-2, -1))  
 target\_dct = torch.fft.fft2(target, dim=(-2, -1))  
  
 # 计算幅度谱  
 pred\_mag = torch.abs(pred\_dct)  
 target\_mag = torch.abs(target\_dct)  
  
 # 低频和高频分别计算损失  
 h, w = pred.shape[-2], pred.shape[-1]  
 low\_freq\_mask = torch.zeros((h, w), device=pred.device)  
 center\_h, center\_w = h // 2, w // 2  
 low\_freq\_range = min(h, w) // 4 # 低频区域大小  
 low\_freq\_mask[center\_h - low\_freq\_range:center\_h + low\_freq\_range,  
 center\_w - low\_freq\_range:center\_w + low\_freq\_range] = 1  
  
 high\_freq\_mask = 1 - low\_freq\_mask  
  
 low\_freq\_loss = F.l1\_loss(pred\_mag \* low\_freq\_mask, target\_mag \* low\_freq\_mask)  
 high\_freq\_loss = F.l1\_loss(pred\_mag \* high\_freq\_mask, target\_mag \* high\_freq\_mask)  
  
 # 调整权重，更注重高频细节（对PSNR和SSIM更有利）  
 return low\_freq\_loss \* 0.2 + high\_freq\_loss \* 0.8  
  
  
# 噪声感知损失  
class NoiseAwareLoss(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def forward(self, pred, target, noise\_residual):  
 # 基础重建损失  
 base\_loss = F.l1\_loss(pred, target)  
  
 # 噪声一致性损失：预测图像与目标图像的噪声特性应该相似  
 # 使用 avg\_pool2d 的近似值，并添加 clamp 和 epsilon 防止除零和极端值  
 pred\_blur = F.avg\_pool2d(pred, kernel\_size=3, stride=1, padding=1)  
 pred\_blur = torch.clamp(pred\_blur, min=1e-4, max=1-1e-4) # 限制模糊后的值在合理范围内  
 pred\_noise = pred - pred\_blur  
  
 target\_blur = F.avg\_pool2d(target, kernel\_size=3, stride=1, padding=1)  
 target\_blur = torch.clamp(target\_blur, min=1e-4, max=1-1e-4)  
 target\_noise = target - target\_blur  
  
 noise\_loss = F.l1\_loss(pred\_noise, target\_noise)  
  
 # 噪声分布损失  
 pred\_noise\_std = torch.std(pred\_noise.view(pred\_noise.shape[0], -1), dim=1)  
 target\_noise\_std = torch.std(target\_noise.view(target\_noise.shape[0], -1), dim=1)  
 # 防止 std 为 0  
 pred\_noise\_std = torch.clamp(pred\_noise\_std, min=1e-6)  
 target\_noise\_std = torch.clamp(target\_noise\_std, min=1e-6)  
 std\_loss = F.l1\_loss(pred\_noise\_std, target\_noise\_std)  
  
 return base\_loss + 0.1 \* noise\_loss + 0.05 \* std\_loss  
  
  
# +++ 修改：在Discriminator的CNN结构中应用谱归一化 +++  
class Discriminator(nn.Module):  
 def \_\_init\_\_(self, in\_channels=3, feat\_channels=64):  
 super().\_\_init\_\_()  
 self.cnn = nn.Sequential(  
 # 将每个Conv2d层用spectral\_norm包装  
 spectral\_norm(nn.Conv2d(in\_channels, feat\_channels, 4, stride=2, padding=1)),  
 nn.LeakyReLU(0.2, inplace=True),  
  
 spectral\_norm(nn.Conv2d(feat\_channels, feat\_channels \* 2, 4, stride=2, padding=1)),  
 nn.InstanceNorm2d(feat\_channels \* 2),  
 nn.LeakyReLU(0.2, inplace=True),  
  
 spectral\_norm(nn.Conv2d(feat\_channels \* 2, feat\_channels \* 4, 4, stride=2, padding=1)),  
 nn.InstanceNorm2d(feat\_channels \* 4),  
 nn.LeakyReLU(0.2, inplace=True),  
  
 spectral\_norm(nn.Conv2d(feat\_channels \* 4, feat\_channels \* 8, 4, stride=2, padding=1)),  
 nn.InstanceNorm2d(feat\_channels \* 8),  
 nn.LeakyReLU(0.2, inplace=True),  
  
 spectral\_norm(nn.Conv2d(feat\_channels \* 8, 1, 4, stride=1, padding=1))  
 )  
  
 def forward(self, x):  
 # 添加输入值范围检查  
 if torch.isnan(x).any() or torch.isinf(x).any():  
 print("警告：判别器输入包含NaN或Inf值！")  
 x = torch.clamp(x, -1.0, 1.0) # 强制裁剪到合理范围  
  
 # 添加梯度监控  
 with torch.autocast('cuda', enabled=False): # 禁用混合精度以确保数值稳定性  
 x = self.cnn(x)  
  
 # 更严格的输出限制  
 return torch.clamp(x, -5.0, 5.0) # 减少输出范围  
class LossFunction(nn.Module):  
 # 总损失函数：组合像素损失、平滑损失、纹理损失、亮度约束等  
 def \_\_init\_\_(self):  
 super(LossFunction, self).\_\_init\_\_()  
 self.\_l2\_loss = nn.MSELoss() # 均方误差损失  
 self.\_l1\_loss = nn.L1Loss() # 平均绝对误差损失  
 self.smooth\_loss = SmoothLoss() # 光照平滑损失  
 self.texture\_difference = TextureDifference() # 纹理差异损失  
 self.local\_mean = LocalMean(patch\_size=5) # 局部均值计算  
 self.L\_TV\_loss = L\_TV() # 总变分(TV)损失  
 self.perceptual\_loss = ImprovedPerceptualLoss() # 改进的感知损失  
 self.ms\_ssim\_loss = MultiScaleSSIMLoss() # 多尺度SSIM损失  
 self.frequency\_loss = FrequencyLoss() # 频率域损失  
  
 self.noise\_aware\_loss = NoiseAwareLoss() # 噪声感知损失  
  
 # 添加颜色一致性损失权重  
 self.color\_constancy\_weight = 0.3  
 self.histogram\_match\_weight = 0.25  
  
 # 预生成直方图平滑用的高斯核  
 self.hist\_bins = 64  
 self.hist\_kernel\_size = 5  
 self.hist\_bandwidth = 0.1  
 kernel = torch.exp(-0.5 \* (torch.linspace(-2, 2, self.hist\_kernel\_size) \*\* 2) / (self.hist\_bandwidth \*\* 2))  
 kernel = kernel / kernel.sum()  
 self.register\_buffer('hist\_kernel', kernel.view(1, 1, -1))  
  
 self.texture\_preserve = TexturePreservationLoss() # 纹理保留损失实例  
 # 添加亮度监控参数  
 self.brightness\_threshold = 0.9 # 降低亮度阈值从0.92到0.9  
 self.overexposure\_weight = 0.1 # 增加过曝惩罚权重从0.3到0.5  
 self.underexposure\_threshold = 0.15  
 # 添加详细的损失记录  
 self.loss\_components\_detail = {}  
 self.loss\_components = {}  
 # 动态权重参数 - 调整以提高PSNR和SSIM  
 self.dynamic\_weights = {  
 'pixel\_reconstruction': {'initial': 1.5, 'final': 0.8, 'transition\_epoch': 2000},  
 'perceptual': {'initial': 0.1, 'final': 0.8, 'transition\_epoch': 2000},  
 'texture\_preserve': {'initial': 0.2, 'final': 0.5, 'transition\_epoch': 2000},  
 'color\_constancy': {'initial': 0.05, 'final': 0.1, 'transition\_epoch': 2000},  
 'histogram\_match': {'initial': 0.05, 'final': 0.2, 'transition\_epoch': 2000},  
 'ms\_ssim': {'initial': 0.3, 'final': 1.0, 'transition\_epoch': 2000},  
 'frequency': {'initial': 0.1, 'final': 0.2, 'transition\_epoch': 2000},  
 'noise\_aware': {'initial': 0.3, 'final': 0.5, 'transition\_epoch': 2000}  
}  
  
 # 当前权重值  
 self.current\_weights = {key: config['initial'] for key, config in self.dynamic\_weights.items()}  
  
 # 添加损失记录字典  
 self.loss\_components = {}  
  
 def ssim(self, x, y, window\_size=11, size\_average=True):  
 """SSIM计算，与MultiScaleSSIMLoss中的实现一致"""  
 C1 = 0.01 \*\* 2  
 C2 = 0.03 \*\* 2  
  
 mu\_x = F.avg\_pool2d(x, window\_size, 1, 0)  
 mu\_y = F.avg\_pool2d(y, window\_size, 1, 0)  
  
 mu\_x\_sq = mu\_x.pow(2)  
 mu\_y\_sq = mu\_y.pow(2)  
 mu\_x\_mu\_y = mu\_x \* mu\_y  
  
 sigma\_x\_sq = F.avg\_pool2d(x \* x, window\_size, 1, 0) - mu\_x\_sq  
 sigma\_y\_sq = F.avg\_pool2d(y \* y, window\_size, 1, 0) - mu\_y\_sq  
 sigma\_xy = F.avg\_pool2d(x \* y, window\_size, 1, 0) - mu\_x\_mu\_y  
  
 ssim\_map = ((2 \* mu\_x\_mu\_y + C1) \* (2 \* sigma\_xy + C2)) / (  
 (mu\_x\_sq + mu\_y\_sq + C1) \* (sigma\_x\_sq + sigma\_y\_sq + C2)  
 )  
  
 if size\_average:  
 return ssim\_map.mean()  
 else:  
 return ssim\_map.mean(1).mean(1).mean(1)  
 # 添加权重更新方法  
 def update\_weights(self, epoch):  
 # 更精细的权重调度  
 transition\_epoch = 2000  
  
 if epoch < 500:  
 # 初期：注重基础重建  
 self.current\_weights = {  
 'pixel\_reconstruction': 1.5,  
 'perceptual': 0.1,  
 'texture\_preserve': 0.2,  
 'color\_constancy': 0.05,  
 'histogram\_match': 0.05,  
 'ms\_ssim': 0.3,  
 'frequency': 0.1,  
 'noise\_aware': 0.3  
 }  
 elif epoch < transition\_epoch:  
 # 过渡期：线性调整  
 alpha = (epoch - 500) / (transition\_epoch - 500)  
 self.current\_weights = {  
 'pixel\_reconstruction': 1.5 - 0.7 \* alpha,  
 'perceptual': 0.1 + 0.7 \* alpha,  
 'texture\_preserve': 0.2 + 0.3 \* alpha,  
 'color\_constancy': 0.05 + 0.05 \* alpha,  
 'histogram\_match': 0.05 + 0.15 \* alpha,  
 'ms\_ssim': 0.3 + 0.7 \* alpha,  
 'frequency': 0.1 + 0.1 \* alpha,  
 'noise\_aware': 0.3 + 0.2 \* alpha  
 }  
 else:  
 # 后期：注重感知质量  
 self.current\_weights = {  
 'pixel\_reconstruction': 0.8,  
 'perceptual': 0.8,  
 'texture\_preserve': 0.5,  
 'color\_constancy': 0.1,  
 'histogram\_match': 0.2,  
 'ms\_ssim': 1.0,  
 'frequency': 0.2,  
 'noise\_aware': 0.5  
 }  
  
 def forward(self, input, target, epoch=0, \*\*kwargs):  
  
 # 数据范围检查  
 assert torch.all(input >= -0.1) and torch.all(  
 input <= 1.1), f"输入数据超出范围: {input.min().item():.4f} - {input.max().item():.4f}"  
 assert torch.all(target >= -0.1) and torch.all(  
 target <= 1.1), f"目标数据超出范围: {target.min().item():.4f} - {target.max().item():.4f}"  
 self.smooth\_factor = min(1.0, epoch / 1000) # 逐渐增加平滑因子  
 # 更新权重  
 self.avg\_brightness = 0  
 self.overexposure\_ratio = 0  
 input = input.float()  
 target = target.float()  
 self.update\_weights(epoch)  
 eps = 1e-9  
  
 # 重置详细记录  
 self.loss\_components\_detail = {}  
  
 # 检查所有输入是否有效  
 for key, value in kwargs.items():  
 if torch.is\_tensor(value):  
 kwargs[key] = value.float() # 确保所有输入都是float32  
 if torch.isnan(value).any() or torch.isinf(value).any():  
 print(f"输入 {key} 包含无效值，使用零替代")  
 kwargs[key] = torch.where(  
 torch.isnan(value) | torch.isinf(value),  
 torch.zeros\_like(value),  
 value  
 )  
  
 # 确保输入在合理范围内  
 input = torch.clamp(input + eps, 0, 1)  
 target = torch.clamp(target, 0, 1) # 确保target也在[0,1]范围内  
 self.update\_weights(epoch)  
  
 # 从 kwargs 中提取所需参数，使用get方法提供默认值  
 L\_pred1 = kwargs.get('L\_pred1', torch.zeros\_like(input))  
 L\_pred2 = kwargs.get('L\_pred2', torch.zeros\_like(input))  
 L2 = kwargs.get('L2', torch.zeros\_like(input))  
 s2 = kwargs.get('s2', torch.zeros\_like(input))  
 s21 = kwargs.get('s21', torch.zeros\_like(input))  
 s22 = kwargs.get('s22', torch.zeros\_like(input))  
 H2 = kwargs.get('H2', torch.zeros\_like(input))  
 H11 = kwargs.get('H11', torch.zeros\_like(input))  
 H12 = kwargs.get('H12', torch.zeros\_like(input))  
 H13 = kwargs.get('H13', torch.zeros\_like(input))  
 s13 = kwargs.get('s13', torch.zeros\_like(input))  
 H14 = kwargs.get('H14', torch.zeros\_like(input))  
 s14 = kwargs.get('s14', torch.zeros\_like(input))  
 H3 = kwargs.get('H3', torch.zeros\_like(input))  
 s3 = kwargs.get('s3', torch.zeros\_like(input))  
 H3\_pred = kwargs.get('H3\_pred', torch.zeros\_like(input))  
 H4\_pred = kwargs.get('H4\_pred', torch.zeros\_like(input))  
 L\_pred1\_L\_pred2\_diff = kwargs.get('L\_pred1\_L\_pred2\_diff', torch.zeros\_like(input))  
 H3\_denoised1\_H3\_denoised2\_diff = kwargs.get('H3\_denoised1\_H3\_denoised2\_diff', torch.zeros\_like(input))  
 H2\_blur = kwargs.get('H2\_blur', torch.zeros\_like(input))  
 H3\_blur = kwargs.get('H3\_blur', torch.zeros\_like(input))  
 H3\_denoised1 = kwargs.get('H3\_denoised1', torch.zeros\_like(input))  
 H3\_denoised2 = kwargs.get('H3\_denoised2', torch.zeros\_like(input))  
 alpha\_pred = kwargs.get('alpha\_pred', torch.zeros(input.size(0), device=input.device))  
 beta\_pred = kwargs.get('beta\_pred', torch.zeros(input.size(0), device=input.device))  
 noise\_residual = kwargs.get('noise\_residual', torch.zeros\_like(input))  
 noise\_prob = kwargs.get('noise\_prob', torch.zeros((input.size(0), 3), device=input.device))  
  
 input = input + eps # 避免除以零  
  
 # 1. 亮度增强约束与归一化约束  
 # 标准 RGB 转灰度公式: R\*0.299 + G\*0.587 + B\*0.114  
 input\_Y = L2.detach()[:, 0] \* 0.299 + L2.detach()[:, 1] \* 0.587 + L2.detach()[:, 2] \* 0.114  
 input\_Y\_mean = torch.mean(input\_Y, dim=(1, 2))  
 enhancement\_factor = 0.5 / (input\_Y\_mean + eps)  
 enhancement\_factor = enhancement\_factor.unsqueeze(1).unsqueeze(2).unsqueeze(3)  
 enhancement\_factor = torch.clamp(enhancement\_factor, 1, 10)  
 adjustment\_ratio = torch.pow(0.7, -enhancement\_factor) / enhancement\_factor  
 adjustment\_ratio = torch.clamp(adjustment\_ratio, 0.1, 10) # 添加钳位  
 adjustment\_ratio = adjustment\_ratio.repeat(1, 3, 1, 1)  
  
 normalized\_low\_light = L2.detach() / (s2 + eps)  
 normalized\_low\_light = torch.clamp(normalized\_low\_light, eps, 1-eps)  
 enhanced\_brightness = torch.pow(L2.detach() \* enhancement\_factor, enhancement\_factor)  
 clamped\_enhanced = torch.clamp(enhanced\_brightness \* adjustment\_ratio, eps, 1)  
 clamped\_adjusted\_low = torch.clamp(L2.detach() \* enhancement\_factor, eps, 1)  
  
 loss = 0.0  
  
 # 亮度整体约束损失（基于动态α与β预测）  
 pix\_loss, smooth\_loss, total\_ie\_loss = ie\_loss(s2, L2, alpha\_pred, beta\_pred)  
 loss += total\_ie\_loss \* 1  
 self.loss\_components['ie\_loss'] = total\_ie\_loss.item()  
 # 添加详细记录  
 self.loss\_components\_detail['ie\_loss'] = {  
 'value': total\_ie\_loss.item(),  
 'components': {  
 'pix\_loss': pix\_loss.item(),  
 'smooth\_loss': smooth\_loss.item()  
 }  
 }  
  
 # 归一化低光层与增强亮度目标的约束  
 norm\_loss = self.\_l2\_loss(normalized\_low\_light, clamped\_adjusted\_low) \* 100  
 loss += norm\_loss  
 self.loss\_components['norm\_loss'] = norm\_loss.item()  
 self.loss\_components\_detail['norm\_loss'] = {  
 'value': norm\_loss.item(),  
 'components': None  
 }  
  
 # 2. 多尺度去噪一致性损失  
 L11\_small, L12\_small = pair\_downsampler(input)  
 loss1 = self.\_l2\_loss(L11\_small, L\_pred2) \* 10  
 loss2 = self.\_l2\_loss(L12\_small, L\_pred1) \* 10  
 loss += loss1 + loss2  
 self.loss\_components['downsample\_loss1'] = loss1.item()  
 self.loss\_components['downsample\_loss2'] = loss2.item()  
 self.loss\_components\_detail['downsample\_loss1'] = {  
 'value': loss1.item(),  
 'components': None  
 }  
 self.loss\_components\_detail['downsample\_loss2'] = {  
 'value': loss2.item(),  
 'components': None  
 }  
  
 denoised1, denoised2 = pair\_downsampler(L2)  
 loss3 = self.\_l2\_loss(L\_pred1, denoised1) \* 50  
 loss4 = self.\_l2\_loss(L\_pred2, denoised2) \* 50  
 loss += loss3 + loss4  
 self.loss\_components['denoise\_loss1'] = loss3.item()  
 self.loss\_components['denoise\_loss2'] = loss4.item()  
 self.loss\_components\_detail['denoise\_loss1'] = {  
 'value': loss3.item(),  
 'components': None  
 }  
 self.loss\_components\_detail['denoise\_loss2'] = {  
 'value': loss4.item(),  
 'components': None  
 }  
  
 # 3. 残差尺寸对齐一致性损失  
 target\_H3 = torch.cat([H12.detach(), s22.detach()], dim=1)  
 if H3\_pred.shape[2:] != target\_H3.shape[2:]:  
 H3\_pred = F.interpolate(H3\_pred, size=target\_H3.shape[2:], mode='bilinear', align\_corners=True)  
 align\_loss1 = self.\_l2\_loss(H3\_pred, target\_H3) \* 50  
 loss += align\_loss1  
 self.loss\_components['align\_loss1'] = align\_loss1.item()  
 self.loss\_components\_detail['align\_loss1'] = {  
 'value': align\_loss1.item(),  
 'components': None  
 }  
  
 target\_H4 = torch.cat([H11.detach(), s21.detach()], dim=1)  
 if H4\_pred.shape[2:] != target\_H4.shape[2:]:  
 H4\_pred = F.interpolate(H4\_pred, size=target\_H4.shape[2:], mode='bilinear', align\_corners=True)  
 align\_loss2 = self.\_l2\_loss(H4\_pred, target\_H4) \* 50  
 loss += align\_loss2  
 self.loss\_components['align\_loss2'] = align\_loss2.item()  
 self.loss\_components\_detail['align\_loss2'] = {  
 'value': align\_loss2.item(),  
 'components': None  
 }  
  
 # 4. 颜色一致性损失（模糊后保证颜色分布一致）  
 color\_loss = self.\_l2\_loss(H2\_blur.detach(), H3\_blur) \* 100  
 loss += color\_loss  
 self.loss\_components['color\_loss'] = color\_loss.item()  
 self.loss\_components\_detail['color\_loss'] = {  
 'value': color\_loss.item(),  
 'components': None  
 }  
  
 # 5. 光照一致性损失  
 illumination\_loss = self.\_l2\_loss(s2.detach(), s3) \* 10  
 loss += illumination\_loss  
 self.loss\_components['illumination\_loss'] = illumination\_loss.item()  
 self.loss\_components\_detail['illumination\_loss'] = {  
 'value': illumination\_loss.item(),  
 'components': None  
 }  
  
 # 6. 内容一致性损失（局部均值约束）  
 local\_mean1 = self.local\_mean(H3\_denoised1)  
 local\_mean2 = self.local\_mean(H3\_denoised2)  
 weighted\_diff1 = (  
 1 - H3\_denoised1\_H3\_denoised2\_diff) \* local\_mean1 + H3\_denoised1 \* H3\_denoised1\_H3\_denoised2\_diff  
 weighted\_diff2 = (  
 1 - H3\_denoised1\_H3\_denoised2\_diff) \* local\_mean2 + H3\_denoised2 \* H3\_denoised1\_H3\_denoised2\_diff  
 content\_loss1 = self.\_l2\_loss(H3\_denoised1, weighted\_diff1) \* 50  
 content\_loss2 = self.\_l2\_loss(H3\_denoised2, weighted\_diff2) \* 50  
 loss += content\_loss1 + content\_loss2  
 self.loss\_components['content\_loss1'] = content\_loss1.item()  
 self.loss\_components['content\_loss2'] = content\_loss2.item()  
 self.loss\_components\_detail['content\_loss1'] = {  
 'value': content\_loss1.item(),  
 'components': None  
 }  
 self.loss\_components\_detail['content\_loss2'] = {  
 'value': content\_loss2.item(),  
 'components': None  
 }  
  
 # 7. 噪声方差约束损失  
 noise\_std = calculate\_local\_variance(H3 - H2)  
 H2\_var = calculate\_local\_variance(H2)  
 noise\_var\_loss = self.\_l2\_loss(H2\_var, noise\_std) \* 50  
 loss += noise\_var\_loss  
 self.loss\_components['noise\_var\_loss'] = noise\_var\_loss.item()  
 self.loss\_components\_detail['noise\_var\_loss'] = {  
 'value': noise\_var\_loss.item(),  
 'components': None  
 }  
  
  
 # 8. 基础像素重建损失（使用动态权重）  
 pred\_img = H3 # 最终的去噪输出图像  
  
 # 添加范围检查和处理  
 pred\_img = torch.clamp(pred\_img, 0, 1)  
 target = torch.clamp(target, 0, 1)  
  
 # 使用更稳定的MSE计算  
 rd\_loss = F.mse\_loss(pred\_img, target)  
 # 添加SSIM损失作为辅助  
 ssim\_loss\_val = 1 - self.ssim(pred\_img, target)  
 # 组合损失  
 reconstruction\_loss = rd\_loss + 0.3 \* ssim\_loss\_val  
  
 loss += self.current\_weights['pixel\_reconstruction'] \* reconstruction\_loss  
 self.loss\_components['pixel\_reconstruction'] = reconstruction\_loss.item()  
 self.loss\_components\_detail['pixel\_reconstruction'] = {  
 'value': reconstruction\_loss.item(),  
 'components': {  
 'mse\_loss': rd\_loss.item(),  
 'ssim\_loss': ssim\_loss\_val.item()  
 }  
 }  
  
 # 9. 感知损失（使用动态权重）  
 perceptual\_loss\_val = self.perceptual\_loss(pred\_img, target)  
 # 应用平滑  
 perceptual\_loss\_val = perceptual\_loss\_val \* self.smooth\_factor + \  
 perceptual\_loss\_val.detach() \* (1 - self.smooth\_factor)  
 loss += self.current\_weights['perceptual'] \* perceptual\_loss\_val  
 self.loss\_components['perceptual'] = perceptual\_loss\_val.item()  
 self.loss\_components\_detail['perceptual'] = {  
 'value': perceptual\_loss\_val.item(),  
 'components': None  
 }  
  
 # 10. 纹理保留损失（使用动态权重）  
 texture\_loss = self.texture\_preserve(input, H3)  
 loss += self.current\_weights['texture\_preserve'] \* texture\_loss  
 self.loss\_components['texture\_preserve'] = texture\_loss.item()  
 self.loss\_components\_detail['texture\_preserve'] = {  
 'value': texture\_loss.item(),  
 'components': None  
 }  
  
 # 11. 颜色一致性损失（使用动态权重）  
 H2\_color = kwargs.get('H2\_color', None)  
 if H2\_color is not None:  
 color\_loss = self.color\_constancy\_loss(H2\_color)  
 loss += self.current\_weights['color\_constancy'] \* color\_loss  
 self.loss\_components['color\_constancy'] = color\_loss.item()  
 self.loss\_components\_detail['color\_constancy'] = {  
 'value': color\_loss.item(),  
 'components': None  
 }  
  
 # 12. 直方图匹配损失（使用动态权重）  
 H3\_for\_hist = kwargs.get('H3', None)  
 if H3\_for\_hist is not None:  
 hist\_loss = self.histogram\_match\_loss(H3\_for\_hist, target)  
 loss += self.current\_weights['histogram\_match'] \* hist\_loss  
 self.loss\_components['histogram\_match'] = hist\_loss.item()  
 self.loss\_components\_detail['histogram\_match'] = {  
 'value': hist\_loss.item(),  
 'components': None  
 }  
  
 # 13. 多尺度SSIM损失（增加权重以提高SSIM）  
 ms\_ssim\_loss\_val = self.ms\_ssim\_loss(pred\_img, target)  
 loss += self.current\_weights['ms\_ssim'] \* ms\_ssim\_loss\_val  
 self.loss\_components['ms\_ssim'] = ms\_ssim\_loss\_val.item()  
 self.loss\_components\_detail['ms\_ssim'] = {  
 'value': ms\_ssim\_loss\_val.item(),  
 'components': None  
 }  
  
 # 14. 频率域损失（调整权重分配）  
 freq\_loss\_val = self.frequency\_loss(pred\_img, target)  
 loss += self.current\_weights['frequency'] \* freq\_loss\_val  
 self.loss\_components['frequency'] = freq\_loss\_val.item()  
 self.loss\_components\_detail['frequency'] = {  
 'value': freq\_loss\_val.item(),  
 'components': None  
 }  
  
 # 16. 噪声感知损失（新增）  
 noise\_aware\_loss\_val = self.noise\_aware\_loss(pred\_img, target, noise\_residual)  
 loss += self.current\_weights['noise\_aware'] \* noise\_aware\_loss\_val  
 self.loss\_components['noise\_aware'] = noise\_aware\_loss\_val.item()  
 self.loss\_components\_detail['noise\_aware'] = {  
 'value': noise\_aware\_loss\_val.item(),  
 'components': None  
 }  
  
 # 添加噪声分类损失（如果提供了真实噪声标签）  
 noise\_type\_label = kwargs.get('noise\_type\_label', None)  
 if noise\_type\_label is not None:  
 noise\_cls\_loss = F.cross\_entropy(noise\_prob, noise\_type\_label)  
 loss += 0.1 \* noise\_cls\_loss  
 self.loss\_components['noise\_classification'] = noise\_cls\_loss.item()  
 self.loss\_components\_detail['noise\_classification'] = {  
 'value': noise\_cls\_loss.item(),  
 'components': None  
 }  
  
 # 5. 亮度约束与过曝控制（关键修改）  
 # 计算当前输出图像的亮度  
 brightness = 0.299 \* pred\_img[:, 0] + 0.587 \* pred\_img[:, 1] + 0.114 \* pred\_img[:, 2]  
 avg\_brightness = torch.mean(brightness)  
 # 记录到self，用于日志打印  
 self.avg\_brightness = avg\_brightness  
  
 # a. 欠曝光惩罚：如果平均亮度低于阈值，则施加惩罚  
 if avg\_brightness < self.underexposure\_threshold:  
 underexposure\_loss = (self.underexposure\_threshold - avg\_brightness) \* 2.0  
 loss += underexposure\_loss  
 self.loss\_components['underexposure\_loss'] = underexposure\_loss.item()  
 self.loss\_components\_detail['underexposure\_loss'] = {  
 'value': underexposure\_loss.item(),  
 'components': None  
 }  
  
 # b. 过曝光惩罚：惩罚过亮的像素  
 overexposure\_mask = (brightness > self.brightness\_threshold).float()  
 self.overexposure\_ratio = torch.mean(overexposure\_mask) # 记录过曝比例  
 overexposure\_loss = torch.mean(overexposure\_mask \* (brightness - self.brightness\_threshold) \*\* 2)  
 loss += self.overexposure\_weight \* overexposure\_loss  
 self.loss\_components['overexposure\_loss'] = overexposure\_loss.item()  
 self.loss\_components\_detail['overexposure\_loss'] = {  
 'value': overexposure\_loss.item(),  
 'components': None  
 }  
  
 # 记录亮度统计信息（用于日志）  
 self.avg\_brightness = torch.mean(pred\_img)  
  
 if not torch.is\_tensor(loss):  
 loss = torch.tensor(loss, device=input.device, dtype=torch.float32, requires\_grad=True)  
  
 # 记录总损失  
 self.loss\_components['total\_loss'] = loss.item()  
 self.loss\_components\_detail['total\_loss'] = {  
 'value': loss.item(),  
 'components': None  
 }  
  
 return loss  
  
 def get\_loss\_components(self):  
 """获取损失组件的字典"""  
 return self.loss\_components  
  
 def get\_detailed\_loss\_components(self):  
 """获取详细的损失组件信息"""  
 return self.loss\_components\_detail  
  
 # 优化后的颜色恒常性损失  
 def color\_constancy\_loss(self, x):  
 """颜色恒常性损失：减少色偏（确保批次内每个样本独立计算）"""  
 # x shape: (batch, 3, h, w)  
 mean\_r = torch.mean(x[:, 0, :, :], dim=(1, 2)) # shape: (batch,)  
 mean\_g = torch.mean(x[:, 1, :, :], dim=(1, 2))  
 mean\_b = torch.mean(x[:, 2, :, :], dim=(1, 2))  
  
 diff\_rg = torch.square(mean\_r - mean\_g)  
 diff\_rb = torch.square(mean\_r - mean\_b)  
 diff\_gb = torch.square(mean\_g - mean\_b)  
  
 return torch.mean(torch.sqrt(diff\_rg + diff\_rb + diff\_gb + 1e-8))  
  
 def adaptive\_brightness\_constraint(self, pred\_img):  
 # 计算亮度（RGB转灰度的加权和）  
 brightness = 0.299 \* pred\_img[:, 0] + 0.587 \* pred\_img[:, 1] + 0.114 \* pred\_img[:, 2]  
 # 计算每个样本的平均亮度（按空间维度求均值）  
 avg\_brightness = torch.mean(brightness, dim=(1, 2))  
  
 # 更温和的亮度调整：以目标亮度0.4为基准  
 target\_brightness = 0.45  
 # 计算调整比例，避免除零  
 brightness\_ratio = target\_brightness / (avg\_brightness + 1e-6)  
 brightness\_ratio = torch.where(avg\_brightness < target\_brightness,  
 torch.clamp(brightness\_ratio, 1.0, 1.5), # 欠曝最多提1.5倍  
 torch.clamp(brightness\_ratio, 0.8, 1.0)) # 过曝只降不升  
  
 # 应用亮度调整（广播到图像维度）  
 adjusted\_img = pred\_img \* brightness\_ratio.view(-1, 1, 1, 1)  
 # 4. 新增欠曝惩罚（对亮度<0.2的像素额外惩罚）  
 underexposed = (brightness < 0.2).float()  
 underexpose\_penalty = torch.mean(underexposed \* (0.2 - brightness) \*\* 2)  
 self.underexpose\_penalty = underexpose\_penalty # 用于后续损失叠加  
 # 确保像素值在有效范围[0,1]内  
 return torch.clamp(adjusted\_img, 0, 1)  
  
 # 修复后的直方图匹配损失  
 def histogram\_match\_loss(self, pred, target, bins=None):  
 bins = self.hist\_bins if bins is None else bins  
 loss = 0.0  
 pred\_clamped = torch.clamp(pred, 0.0, 1.0)  
 target\_clamped = torch.clamp(target, 0.0, 1.0)  
  
 # 确保直方图核在与输入相同的设备上  
 hist\_kernel = self.hist\_kernel.to(pred.device)  
  
 for c in range(3):  
 # 计算归一化直方图 - 确保在正确设备上  
 pred\_hist = torch.histc(pred\_clamped[:, c].flatten(), bins=bins, min=0.0, max=1.0)  
 pred\_hist = pred\_hist.to(pred.device) # 确保在相同设备  
 pred\_hist = pred\_hist / (pred\_hist.sum() + 1e-8)  
  
 target\_hist = torch.histc(target\_clamped[:, c].flatten(), bins=bins, min=0.0, max=1.0)  
 target\_hist = target\_hist.to(pred.device) # 确保在相同设备  
 target\_hist = target\_hist / (target\_hist.sum() + 1e-8)  
  
 # 高斯平滑（设备一致）  
 pred\_smoothed = F.conv1d(  
 pred\_hist.view(1, 1, -1),  
 hist\_kernel,  
 padding=(self.hist\_kernel\_size - 1) // 2  
 ).squeeze()  
  
 target\_smoothed = F.conv1d(  
 target\_hist.view(1, 1, -1),  
 hist\_kernel,  
 padding=(self.hist\_kernel\_size - 1) // 2  
 ).squeeze()  
  
 loss += F.l1\_loss(pred\_smoothed, target\_smoothed)  
  
 return loss / 3  
  
  
def ie\_loss(s, i, alpha\_pred, beta\_pred):  
 # 使用预测的动态参数，而非固定计算  
  
 gamma = 0.7  
 eps = 1e-6  
 # 像素强度调整损失 - 使用预测的alpha和beta  
 # 将形状为 [B] 的 alpha\_pred 和 beta\_pred 扩展为 [B, 1, 1, 1] 以匹配图像张量 s 和 i 的形状 [B, C, H, W]  
 alpha\_expanded = alpha\_pred[:, None, None, None] # 等同于 .unsqueeze(1).unsqueeze(2).unsqueeze(3)  
 beta\_expanded = beta\_pred[:, None, None, None]  
  
 # 钳位输入值  
 i\_clamped = torch.clamp(i, eps, 1 - eps)  
 alpha\_i = torch.clamp(alpha\_expanded \* i\_clamped, min=eps)  
 # 计算像素损失  
 pix\_loss = F.mse\_loss(s, beta\_expanded \* (alpha\_i + eps) \*\* gamma)  
  
 # 平滑损失  
 grad\_h = torch.abs(s[:, :, 1:, :] - s[:, :, :-1, :])  
 grad\_w = torch.abs(s[:, :, :, 1:] - s[:, :, :, :-1])  
 smooth\_loss = grad\_h.mean() + grad\_w.mean()  
  
 total\_loss = pix\_loss + 0.01 \* smooth\_loss # 总损失  
 return pix\_loss, smooth\_loss, total\_loss # 返回子分量和总损失  
  
  
class TextureDifference(nn.Module):  
 # 计算两张图像的纹理差异  
 def \_\_init\_\_(self, patch\_size=5, constant\_C=1e-5, threshold=0.975):  
 super(TextureDifference, self).\_\_init\_\_()  
 self.patch\_size = patch\_size  
 self.constant\_C = constant\_C  
 self.threshold = threshold  
  
 def forward(self, image1, image2):  
 eps = 1e-8  
 # 转灰度  
 image1 = self.rgb\_to\_gray(image1)  
 image2 = self.rgb\_to\_gray(image2)  
 # 计算局部标准差（纹理变化程度）  
 stddev1 = self.local\_stddev(image1)  
 stddev2 = self.local\_stddev(image2)  
 numerator = 2 \* stddev1 \* stddev2  
 denominator = stddev1 \*\* 2 + stddev2 \*\* 2 + self.constant\_C + eps  
 diff = numerator / denominator # 范围[0,1]  
 # 超过阈值的视为纹理一致（记为1），否则为0  
 binary\_diff = torch.where(diff > self.threshold,  
 torch.tensor(1.0, device=diff.device),  
 torch.tensor(0.0, device=diff.device))  
 return binary\_diff  
  
 # 修复缩进：确保这两个方法在类内部  
 def local\_stddev(self, image):  
 padding = self.patch\_size // 2  
 image = F.pad(image, (padding, padding, padding, padding), mode='reflect')  
 patches = image.unfold(2, self.patch\_size, 1).unfold(3, self.patch\_size, 1)  
 mean = patches.mean(dim=(4, 5), keepdim=True)  
 squared\_diff = (patches - mean) \*\* 2  
 local\_var = squared\_diff.mean(dim=(4, 5))  
 local\_std = torch.sqrt(local\_var + 1e-9)  
 return local\_std  
  
 def rgb\_to\_gray(self, image):  
 gray\_image = 0.144 \* image[:, 0] + 0.587 \* image[:, 1] + 0.299 \* image[:, 2]  
 return gray\_image.unsqueeze(1)  
  
  
class TexturePreservationLoss(nn.Module):  
 def \_\_init\_\_(self, edge\_weight=0.8):  
 super().\_\_init\_\_()  
 self.edge\_weight = edge\_weight  
 # Sobel算子用于边缘检测  
 self.sobel\_x = nn.Conv2d(1, 1, kernel\_size=3, padding=1, bias=False)  
 self.sobel\_x.weight.data = torch.tensor([  
 [-1, 0, 1],  
 [-2, 0, 2],  
 [-1, 0, 1]  
 ], dtype=torch.float32).view(1, 1, 3, 3)  
  
 self.sobel\_y = nn.Conv2d(1, 1, kernel\_size=3, padding=1, bias=False)  
 self.sobel\_y.weight.data = torch.tensor([  
 [-1, -2, -1],  
 [0, 0, 0],  
 [1, 2, 1]  
 ], dtype=torch.float32).view(1, 1, 3, 3)  
  
 # 冻结参数  
 self.sobel\_x.weight.requires\_grad = True  
 self.sobel\_y.weight.requires\_grad = True  
  
 def forward(self, input, output):  
 # 转为灰度图  
 input\_gray = 0.299 \* input[:, 0] + 0.587 \* input[:, 1] + 0.114 \* input[:, 2]  
 output\_gray = 0.299 \* output[:, 0] + 0.587 \* output[:, 1] + 0.114 \* output[:, 2]  
  
 # 计算梯度幅度  
 input\_grad\_x = self.sobel\_x(input\_gray.unsqueeze(1))  
 input\_grad\_y = self.sobel\_y(input\_gray.unsqueeze(1))  
 input\_grad\_mag = torch.sqrt(input\_grad\_x \*\* 2 + input\_grad\_y \*\* 2 + 1e-6)  
  
 output\_grad\_x = self.sobel\_x(output\_gray.unsqueeze(1))  
 output\_grad\_y = self.sobel\_y(output\_gray.unsqueeze(1))  
 output\_grad\_mag = torch.sqrt(output\_grad\_x \*\* 2 + output\_grad\_y \*\* 2 + 1e-6)  
  
 # 梯度相似性损失  
 grad\_loss = F.l1\_loss(output\_grad\_mag, input\_grad\_mag)  
  
 # 结构相似性损失（SSIM）  
 ssim\_loss = 1 - self.ssim(output, input)  
  
 return self.edge\_weight \* grad\_loss + (1 - self.edge\_weight) \* ssim\_loss  
  
 def ssim(self, x, y, window\_size=11, size\_average=True):  
 # 简化SSIM实现  
 C1 = 0.01 \*\* 2  
 C2 = 0.03 \*\* 2  
  
 mu\_x = F.avg\_pool2d(x, window\_size, 1, 0)  
 mu\_y = F.avg\_pool2d(y, window\_size, 1, 0)  
  
 mu\_x\_sq = mu\_x.pow(2)  
 mu\_y\_sq = mu\_y.pow(2)  
 mu\_x\_mu\_y = mu\_x \* mu\_y  
  
 sigma\_x\_sq = F.avg\_pool2d(x \* x, window\_size, 1, 0) - mu\_x\_sq  
 sigma\_y\_sq = F.avg\_pool2d(y \* y, window\_size, 1, 0) - mu\_y\_sq  
 sigma\_xy = F.avg\_pool2d(x \* y, window\_size, 1, 0) - mu\_x\_mu\_y  
  
 ssim\_map = ((2 \* mu\_x\_mu\_y + C1) \* (2 \* sigma\_xy + C2)) / (  
 (mu\_x\_sq + mu\_y\_sq + C1) \* (sigma\_x\_sq + sigma\_y\_sq + C2))  
  
 if size\_average:  
 return ssim\_map.mean()  
 else:  
 return ssim\_map.mean(1).mean(1).mean(1)  
  
  
class L\_TV(nn.Module):  
 # 总变分损失，用于保持图像平滑  
 def \_\_init\_\_(self, TVLoss\_weight=1):  
 super(L\_TV, self).\_\_init\_\_()  
 self.TVLoss\_weight = TVLoss\_weight  
  
 def forward(self, x):  
 batch\_size = x.size(0)  
 h\_x = x.size(2)  
 w\_x = x.size(3)  
 count\_h = (h\_x - 1) \* w\_x  
 count\_w = h\_x \* (w\_x - 1)  
 h\_tv = ((x[:, :, 1:, :] - x[:, :, :h\_x - 1, :]) \*\* 2).sum()  
 w\_tv = ((x[:, :, :, 1:] - x[:, :, :, :w\_x - 1]) \*\* 2).sum()  
 return self.TVLoss\_weight \* 2 \* (h\_tv / count\_h + w\_tv / count\_w) / batch\_size  
  
  
class Blur(nn.Module):  
 def \_\_init\_\_(self, nc):  
 super().\_\_init\_\_()  
 self.nc = nc  
 kernel\_tensor = gauss\_kernel(kernlen=21, nsig=3, channels=self.nc)  
 weight = kernel\_tensor.float()  
 self.register\_buffer('weight', weight)  
  
 def forward(self, x):  
 if x.size(1) != self.nc:  
 raise RuntimeError(f"输入通道数[{x.size(1)}]与预设[{self.nc}]不匹配")  
 return F.conv2d(x, self.weight, stride=1, padding=10, groups=self.nc)  
  
  
class SmoothLoss(nn.Module):  
 # 平滑损失：约束光照图的空间平滑性（基于输入图像颜色相似性）  
 def \_\_init\_\_(self):  
 super(SmoothLoss, self).\_\_init\_\_()  
 self.sigma = 10  
  
 def rgb2yCbCr(self, input\_im):  
 im\_flat = input\_im.contiguous().view(-1, 3).float()  
 device = input\_im.device  
 mat = torch.tensor([[0.257, -0.148, 0.439],  
 [0.564, -0.291, -0.368],  
 [0.098, 0.439, -0.071]], device=device)  
 bias = torch.tensor([16 / 255., 128 / 255., 128 / 255.], device=device)  
 temp = im\_flat @ mat + bias  
 out = temp.view(input\_im.shape[0], 3, input\_im.shape[2], input\_im.shape[3])  
 return out  
  
 def forward(self, input, output):  
 # input: 原始图像; output: 光照图s2  
 self.output = output  
 self.input = self.rgb2yCbCr(input)  
 sigma\_color = -1.0 / (2 \* self.sigma \* self.sigma)  
 # 计算各方向的颜色相似性权重  
 w1 = torch.exp(  
 torch.sum((self.input[:, :, 1:, :] - self.input[:, :, :-1, :]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w2 = torch.exp(  
 torch.sum((self.input[:, :, :-1, :] - self.input[:, :, 1:, :]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w3 = torch.exp(  
 torch.sum((self.input[:, :, :, 1:] - self.input[:, :, :, :-1]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w4 = torch.exp(  
 torch.sum((self.input[:, :, :, :-1] - self.input[:, :, :, 1:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w5 = torch.exp(  
 torch.sum((self.input[:, :, :-1, :-1] - self.input[:, :, 1:, 1:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w6 = torch.exp(  
 torch.sum((self.input[:, :, 1:, 1:] - self.input[:, :, :-1, :-1]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w7 = torch.exp(  
 torch.sum((self.input[:, :, 1:, :-1] - self.input[:, :, :-1, 1:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w8 = torch.exp(  
 torch.sum((self.input[:, :, :-1, 1:] - self.input[:, :, 1:, :-1]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w9 = torch.exp(  
 torch.sum((self.input[:, :, 2:, :] - self.input[:, :, :-2, :]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w10 = torch.exp(  
 torch.sum((self.input[:, :, :-2, :] - self.input[:, :, 2:, :]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w11 = torch.exp(  
 torch.sum((self.input[:, :, :, 2:] - self.input[:, :, :, :-2]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w12 = torch.exp(  
 torch.sum((self.input[:, :, :, :-2] - self.input[:, :, :, 2:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w13 = torch.exp(  
 torch.sum((self.input[:, :, :-2, :-1] - self.input[:, :, 2:, 1:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w14 = torch.exp(  
 torch.sum((self.input[:, :, 2:, 1:] - self.input[:, :, :-2, :-1]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w15 = torch.exp(  
 torch.sum((self.input[:, :, 2:, :-1] - self.input[:, :, :-2, 1:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w16 = torch.exp(  
 torch.sum((self.input[:, :, :-2, 1:] - self.input[:, :, 2:, :-1]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w17 = torch.exp(torch.sum((self.input[:, :, :-1, :-2] - self.input[:, :, 1:, 2:], 2) \*\* 2, dim=1,  
 keepdim=True) \* sigma\_color)  
 w18 = torch.exp(  
 torch.sum((self.input[:, :, 1:, 2:] - self.input[:, :, :-1, :-2]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w19 = torch.exp(  
 torch.sum((self.input[:, :, 1:, :-2] - self.input[:, :, :-1, 2:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w20 = torch.exp(  
 torch.sum((self.input[:, :, :-1, 2:] - self.input[:, :, 1:, :-2]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w21 = torch.exp(  
 torch.sum((self.input[:, :, :-2, :-2] - self.input[:, :, 2:, 2:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
  
 w22 = torch.exp(  
 torch.sum((self.input[:, :, 2:, 2:] - self.input[:, :, :-2, :-2]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w23 = torch.exp(  
 torch.sum((self.input[:, :, 2:, :-2] - self.input[:, :, :-2, 2:]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 w24 = torch.exp(  
 torch.sum((self.input[:, :, :-2, 2:] - self.input[:, :, 2:, :-2]) \*\* 2, dim=1, keepdim=True) \* sigma\_color)  
 # 计算光照图在各方向的加权差异  
 pixel\_grad1 = w1 \* torch.norm(self.output[:, :, 1:, :] - self.output[:, :, :-1, :], p=1, dim=1, keepdim=True)  
 pixel\_grad2 = w2 \* torch.norm(self.output[:, :, :-1, :] - self.output[:, :, 1:, :], p=1, dim=1, keepdim=True)  
 pixel\_grad3 = w3 \* torch.norm(self.output[:, :, :, 1:] - self.output[:, :, :, :-1], p=1, dim=1, keepdim=True)  
 pixel\_grad4 = w4 \* torch.norm(self.output[:, :, :, :-1] - self.output[:, :, :, 1:], p=1, dim=1, keepdim=True)  
 pixel\_grad5 = w5 \* torch.norm(self.output[:, :, :-1, :-1] - self.output[:, :, 1:, 1:], p=1, dim=1, keepdim=True)  
 pixel\_grad6 = w6 \* torch.norm(self.output[:, :, 1:, 1:] - self.output[:, :, :-1, :-1], p=1, dim=1, keepdim=True)  
 pixel\_grad7 = w7 \* torch.norm(self.output[:, :, 1:, :-1] - self.output[:, :, :-1, 1:], p=1, dim=1, keepdim=True)  
 pixel\_grad8 = w8 \* torch.norm(self.output[:, :, :-1, 1:] - self.output[:, :, 1:, :-1], p=1, dim=1, keepdim=True)  
 pixel\_grad9 = w9 \* torch.norm(self.output[:, :, 2:, :] - self.output[:, :, :-2, :], p=1, dim=1, keepdim=True)  
 pixel\_grad10 = w10 \* torch.norm(self.output[:, :, :-2, :] - self.output[:, :, 2:, :], p=1, dim=1, keepdim=True)  
 pixel\_grad11 = w11 \* torch.norm(self.output[:, :, :, 2:] - self.output[:, :, :, :-2], p=1, dim=1, keepdim=True)  
 pixel\_grad12 = w12 \* torch.norm(self.output[:, :, :, :-2] - self.output[:, :, :, 2:], p=1, dim=1, keepdim=True)  
 pixel\_grad13 = w13 \* torch.norm(self.output[:, :, :-2, :-1] - self.output[:, :, 2:, 1:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad14 = w14 \* torch.norm(self.output[:, :, 2:, 1:] - self.output[:, :, :-2, :-1], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad15 = w15 \* torch.norm(self.output[:, :, 2:, :-1] - self.output[:, :, :-2, 1:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad16 = w16 \* torch.norm(self.output[:, :, :-2, 1:] - self.output[:, :, 2:, :-1], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad17 = w17 \* torch.norm(self.output[:, :, :-1, :-2] - self.output[:, :, 1:, 2:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad18 = w18 \* torch.norm(self.output[:, :, 1:, 2:] - self.output[:, :, :-1, :-2], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad19 = w19 \* torch.norm(self.output[:, :, 1:, :-2] - self.output[:, :, :-1, 2:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad20 = w20 \* torch.norm(self.output[:, :, :-1, 2:] - self.output[:, :, 1:, :-2], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad21 = w21 \* torch.norm(self.output[:, :, :-2, :-2] - self.output[:, :, 2:, 2:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad22 = w22 \* torch.norm(self.output[:, :, 2:, 2:] - self.output[:, :, :-2, :-2], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad23 = w23 \* torch.norm(self.output[:, :, 2:, :-2] - self.output[:, :, :-2, 2:], p=1, dim=1,  
 keepdim=True)  
 pixel\_grad24 = w24 \* torch.norm(self.output[:, :, :-2, 2:] - self.output[:, :, 2:, :-2], p=1, dim=1,  
 keepdim=True)  
 # 平滑损失：所有方向差异的平均和  
 reg\_term = (pixel\_grad1.mean() + pixel\_grad2.mean() + pixel\_grad3.mean() + pixel\_grad4.mean() +  
 pixel\_grad5.mean() + pixel\_grad6.mean() + pixel\_grad7.mean() + pixel\_grad8.mean() +  
 pixel\_grad9.mean() + pixel\_grad10.mean() + pixel\_grad11.mean() + pixel\_grad12.mean() +  
 pixel\_grad13.mean() + pixel\_grad14.mean() + pixel\_grad15.mean() + pixel\_grad16.mean() +  
 pixel\_grad17.mean() + pixel\_grad18.mean() + pixel\_grad19.mean() + pixel\_grad20.mean() +  
 pixel\_grad21.mean() + pixel\_grad22.mean() + pixel\_grad23.mean() + pixel\_grad24.mean())  
 return reg\_term