5 月 21 日-5 月 29 日工作汇报

Ku Jui

May 2023

Contents

1 文献阅读

1.1 Pre-knowledge

模型中的损失函数是用来衡量模型的预测值和真实值 (Ground Truth) 之间的差异程度的函数，它可以反映模型的优化方向和性能指标[1]。它是一种衡量模型预测结果与真实结

[1]https://zhuanlan.zhihu.com/p/375968083

果之间差异的方法，用于指导模型参数的更新。在训练过程中，通过不断最小化损失函数来优化模型参数，使模型能够更好地拟合数据[2]。因此，需要使用合适的损失函数，当模型在数据集上进行训练时，该函数可以适当地惩罚模型 [3]。

不同的损失函数适用于不同的任务和数据分布，例如回归问题常用的有均方误差损失函数 (MSE，也叫做 $\mathcal{L}_2$ 损失函数) 和 $\mathcal{L}_1$ 损失函数，分类问题常用的有交叉熵损失函数 (Cross Entropy Loss) 等[4]。损失函数的选择会影响模型的收敛速度和精度，因此需要根据具体情况选择合适的损失函数[5]。

目前常用的损失函数是从相关视觉任务中借用的，但这些损失函数可能并不完全适用于低照度图像增强 LLIE。因此，需要设计更适合 LLIE 的损失函数，以更好地驱动深度网络的优化。这可以通过研究人类对图像质量的视觉感知来实现，使用深度神经网络来近似人类视觉感知，并将这些理论应用于损失函数的设计。损失函数可以分为两个大类：回归问题和分类问题。

1.1.1 loss function for CV

$\mathcal{L}_1$-loss

平均绝对误差 (MAE) 损失，也称 $\mathcal{L}_1$ 范数损失，计算实际值和预测值之间绝对差之和的平均值。

$$\mathcal{L}_1 = \frac{1}{N}\sum_{i=1}^{N} \|\hat{y}_i - y_i\|_1 \tag{1}$$

适用于回归问题，MAE loss 对异常值更具鲁棒性，尤其是当目标变量的分布有离群值时 (小值或大值与平均值相差很大)。

函数：*torch.nn.L1Loss*

$\mathcal{L}_2$-loss

均方误差 (MSE) 损失，也称为 $\mathcal{L}_2$ 范数损失，计算实际值和预测值之间平方差的平均值[6]。

[2]https://zhuanlan.zhihu.com/p/436809988

[3]https://zhuanlan.zhihu.com/p/473113939

[4]https://blog.csdn.net/weixin_57643648/article/details/122704657

[5]https://www.zhihu.com/tardis/zm/art/136047113?source_id=1005

[6]https://blog.csdn.net/yanyuxiangtoday/article/details/119788949

$$\mathcal{L}_1 = \frac{1}{N}\sum_{i=1}^{N} \|\hat{y}_i - y_i\|_2^2 \tag{2}$$

平方意味着较大的误差比较小的误差会产生更大的惩罚，所以 $\mathcal{L}_2 - loss$ 的收敛速度要比 $\mathcal{L}_1 - loss$ 要快得多。但是，$\mathcal{L}_2 - loss$ 对异常点更敏感，鲁棒性差于 $\mathcal{L}_1 - loss$。

$\mathcal{L}_1 - loss$ 损失函数相比于 $\mathcal{L}_2 - loss$ 损失函数的鲁棒性更好。因为以 $\mathcal{L}_2 - loss$ 范数将误差平方化 (如果误差大于 1，则误差会放大很多)，模型的误差会比以 $\mathcal{L}_1 - loss$ 范数大的多，因此模型会对这种类型的样本更加敏感，这就需要调整模型来最小化误差。但是很大可能这种类型的样本是一个异常值，模型就需要调整以适应这种异常值，那么就导致训练模型的方向偏离目标了[7]。

对于大多数回归问题，一般是使用 $\mathcal{L}_2 - loss$ 而不是 $\mathcal{L}_1 - loss$，$\mathcal{L}_2 - loss$ 在 LLIE 中，具有的相当很有效的恢复效果，但是由于平方项，它通常会产生模糊的恢复结果，如 [1] 所述，平方项为相对较小的误差提供了较低的梯度。。

函数: *torch.nn.MSELoss*

Regularization of the $\mathcal{L}_1$ and $\mathcal{L}_2$ loss functions

正则化的基本思想是通过在损失函数中加入额外信息，以便防止过拟合和提高模型泛化性能。无论哪一种正则化方式，基本思想都是希望通过限制权重的大小，使得模型不能任意拟合训练数据中的随机噪声，正则化实际是在损失函数中加入刻画模型复杂程度的指标[8]。

对应的 L1 正则损失函数:

$$\mathcal{L}_{norm1} = \mathcal{L}_1(\hat{y}, y) + \lambda \sum_{\omega} \|\omega\|_1 \tag{3}$$

对应的 L2 正则损失函数：

$$\mathcal{L}_{norm2} = \mathcal{L}_2(\hat{y}, y) + \lambda \sum_{\omega} \|\omega\|_2^2 \tag{4}$$

假设 $\mathcal{L}_1(\hat{y}, y)$ 和 $\mathcal{L}_2(\hat{y}, y)$ 是未加正则项的损失，λ 是一个超参，用于控制正则化项的大小，惩罚项 ω 用于惩罚大的权重，隐式地减少自由参数的数量。

正则化是如何降低过拟合现象的？

正则化之所以能够降低过拟合的原因在于，正则化是结构风险最小化的一种策略实现。给损失函数加上正则化项，能使得新得到的优化目标函数 $h = f + normal$ ，需要在 f 和

[7] https://zhuanlan.zhihu.com/p/137073968

[8] https://blog.csdn.net/weixin_41960890/article/details/104891561

normal 中做一个权衡 (trade-off)，如果还像原来只优化 f 的情况下，那可能得到一组解比较复杂，使得正则项 *normal* 比较大，那么 h 就不是最优的，因此可以看出加正则项能让解更加简单，符合奥卡姆剃刀理论，同时也比较符合在偏差和方差 (方差表示模型的复杂度) 分析中，通过降低模型复杂度，得到更小的泛化误差，降低过拟合程度[9]。

PyTorch 实现：$\mathcal{L}_2$ 正则项是通过 optimizer 优化器的参数 weight_decay(float, optional) 添加的，用于设置权值衰减率，即正则化中的超参 λ，默认值为 0。

```
optimizer = torch.optim.SGD(model.parameters(),lr=0.01,weight_decay=0.01)
```

Smooth $\mathcal{L}_1$ loss function

Smooth $\mathcal{L}_1$ 损失函数是由 Girshick R 在 Fast R-CNN 中提出的，主要用在目标检测中防止梯度爆炸。它是一个分段函数，在 $[-1,1]$ 之间是 $\mathcal{L}_2$ 损失，其他区间就是 $\mathcal{L}_1$ 损失。这样即解决了 $\mathcal{L}_1$ 损失在 0 处不可导的问题，也解决了 $\mathcal{L}_2$ 损失在异常点处梯度爆炸的问题[10]。

$$\text{smooth } \mathcal{L}_1 \text{ loss} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\dfrac{\|\hat{y}_i - y_i\|_2^2}{2\beta}, \|\hat{y}_i - y_i\| < \beta, \\ \|\hat{y}_i - y_i\|_1 - \dfrac{1}{2}\beta, \|\hat{y}_i - y_i\| \geq \beta.\end{cases} \tag{5}$$

一般取 $\beta = 1$。smooth $\mathcal{L}_1$ 和 $\mathcal{L}_1$-loss 函数的区别在于，smooth $\mathcal{L}_1$ 在 0 点附近使用 $\mathcal{L}_2$ 使得它更加平滑, 它同时拥有 $\mathcal{L}_2$-loss 和 $\mathcal{L}_1$-loss 的部分优点。

函数：*torch.nn.SmoothL1Loss*

Huber loss function

$\mathcal{L}_2$-loss 但容易受离群点的影响，$\mathcal{L}_1$-loss 对离群点更加健壮但是收敛慢，Huber Loss 则是一种将 MSE 与 MAE 结合起来，取两者优点的损失函数，也被称作 Smooth Mean Absolute Error Loss。其原理很简单，就是在误差接近 0 时使用 $\mathcal{L}_2$-loss，误差较大时使用 $\mathcal{L}_1$-loss

$$J_{Huber}(\delta) = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\dfrac{1}{2}\|\hat{y}_i - y_i\|_2^2, \|\hat{y}_i - y_i\| < \delta, \\ \delta\left(\|\hat{y}_i - y_i\|_1 - \dfrac{1}{2}\delta\right), \|\hat{y}_i - y_i\| \geq \delta.\end{cases} \tag{6}$$

[9]https://zhuanlan.zhihu.com/p/35356992

[10]https://zhuanlan.zhihu.com/p/261059231

残差比较小时，Huber Loss 是二次函数；残差比较大时，Huber Loss 是线性函数 (残差，即观测值和预测值之间的差值)。与 $\mathcal{L}_2$-loss 相比，Huber 损失对数据中的异常值不那么敏感。使函数二次化的小误差值是多少取决于“超参数”δ，它可以调整。当 $\delta = 1$ 时，退化成 smooth $\mathcal{L}_1$ Loss。

函数：*torch.nn.HuberLoss*

log-MSE

$$J_{log-MSE} = 10 \log_{10} \left(\frac{1}{N} \sum_{i=1}^{N} \|\hat{y}_i - y_i\|_2^2 \right) \tag{7}$$

Perceptual loss function

感知损失 (Perceptual Loss) 是一种用于比较两个看起来相似的图像的损失函数，这一损失函数由 Johnson et al. [2] 提出。它用于比较图像之间的高层次差异，如内容和风格差异[11]。它已被广泛用作图像合成任务 (包括图像超分辨率和风格转换) 中的有效损失项。

感知损失函数用于比较两个看起来相似的不同图像，例如同一张照片，但偏移了一个像素。该函数用于比较图像之间的高级差异，例如内容和样式差异。感知损失函数与每像素损失函数非常相似，因为两者都用于训练前馈神经网络以进行图像转换任务。感知损失函数是一个更常用的组件，因为它通常提供有关风格迁移的更准确的结果。

简而言之，感知损失函数的工作原理是将所有像素之间的所有平方误差相加并取平均值。这与每像素损失函数形成对比，后者对像素之间的所有绝对误差求和 [2]。

作者认为感知损失函数不仅在生成高质量图像方面更准确，而且在优化后也快了三倍。神经网络模型在图像上进行训练，其中感知损失函数基于从已训练网络中提取的高级特征进行优化。

$$\ell_{feat}^{\phi,j}(\hat{y}, y) = \frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y)\|_2^2 \tag{8}$$

其中 $\hat{y}$ 为输出图像，y 为目标图像，ϕ 为损失网络。$\phi_j(x)$ 为处理图像 x 时损失网络 ϕ 的第 j 层的激活情况，如果 j 是一个卷积层，那么 $\phi_j(x)$ 将是形状 $C_j \times H_j \times W_j$ 的特征映射，特征重建损失是特征表示之间的欧式距离，如 eq 8。

Fig.1表示经过训练以将输入图像转换为输出图像的神经网络。用于图像分类的预训练损失网络有助于通知损失函数。预先训练的网络有助于定义测量图像之间内容和风格的感知差异所需的感知损失函数。

[11]https://deepai.org/machine-learning-glossary-and-terms/perceptual-loss-function

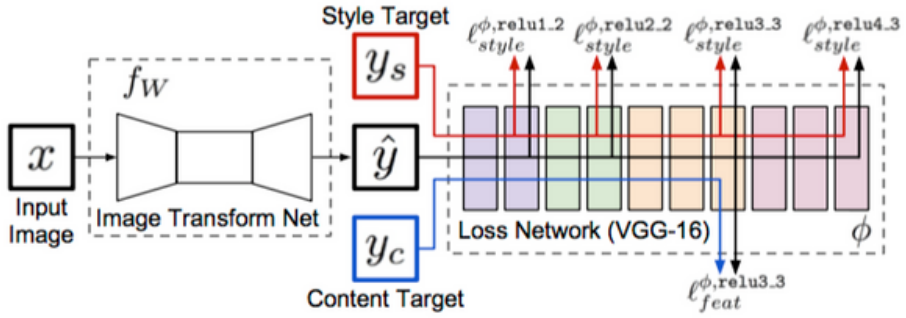


Figure 1: System overview. We train an image transformation network to transform input images into output images. We use a loss network pretrained for image classification to define perceptual loss functions that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

对于图像数据来说，网络在提取特征的过程中，较浅层通常提取边缘、颜色、亮度等低频信息，而网络较深层则提取一些细节纹理等高频信息，再深一点的网络层则提取一些具有辨别性的关键特征，也就是说，网络层越深提取的特征越抽象越高级。

感知损失就是通过一个固定的网络 (通常使用预训练的 VGG16 或者 VGG19)，分别以真实图像 (Ground Truth)、网络生成结果 (Prediciton) 作为其输入，得到对应的输出特征：feature_gt、feature_pre，然后使用 feature_gt 与 feature_pre 构造损失 (通常为 $\mathcal{L}_2$-loss)，逼近真实图像与网络生成结果之间的深层信息，也就是感知信息，相比普通的 $\mathcal{L}_2$-loss 而言，可以增强输出特征的细节信息[12]。

SSIM loss function

SSIM 损失函数是一种用于衡量两幅图像之间差距的损失函数。它考虑了亮度、对比度和结构指标，这就考虑了人类视觉感知，一般而言，SSIM 得到的结果会比 $\mathcal{L}_1$-loss，$\mathcal{L}_2$-loss 的结果更有细节[13]。

每个像素 p 的SSIM被定义为

$$\begin{aligned}\text{SSIM}(p) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \cdot \frac{2\sigma_{xy} + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \\ &= l(p) \cdot cs(p)\end{aligned} \tag{9}$$

其中省略了均值和标准偏差对像素 p 的依赖性，均值和标准差是用标准偏差为 σ_G, G_{σ_G}

$$\begin{aligned}&\varepsilon(p) = 1 - \text{SSIM}(p) : \\ &\mathcal{L}^{\text{SSIM}}(P) = \frac{1}{N}\sum_{p\in P} 1 - \text{SSIM}(p).\end{aligned} \tag{10}$$

[12]https://blog.csdn.net/qq_43665602/article/details/127077484

[13]https://blog.csdn.net/u013289254/article/details/99694412

eq. 9表明 SSIM(p) 需要关注像素 p 的邻域，这个领域的大小取决于 G_{σ_G}，网络的卷积性质允许我们将 SSIM 损失写为

$$\mathcal{L}^{\text{SSIM}}(P) = 1 - \text{SSIM}(\tilde{p}). \tag{11}$$

其中 $\tilde{p}$ 是 P 的中心像素。

MS-SSIM loss function

多尺度结构相似性 (MS-SSIM) 损失函数是基于多层 (图片按照一定规则，由大到小缩放) 的 SSIM 损失函数，相当于考虑了分辨率[14]。它是一种更为复杂的 SSIM 损失函数，可以更好地衡量图像之间的相似性。

$$\text{MS-SSIM}(p) = l_M^{\alpha}(p) \cdot \prod_{j=1}^{M} cs_j^{\beta_j}(p) \tag{12}$$

其中 M, j 描述的是比例，设 $\alpha = \beta_j = 1$，对于 $j = 1, \cdots, M$ 类似 eq. 11，利用中心像素 $\tilde{p}$ 处计算的损失来近似贴片 P 的损失：

$$\mathcal{L}^{\text{MS-SSIM}}(P) = 1 - \text{MS-SSIM}(\tilde{p}) \tag{13}$$

Cross-entropy loss function

交叉熵损失函数是一种常用的分类问题损失函数。在二分类问题中，它的定义为 Eq. 14

$$\mathcal{L}(\hat{y}, y) = -\left(y \log_{\hat{y}} + (1-y)\log(1-\hat{y})\right) \tag{14}$$

其中，$\hat{y}$ 表示模型预测的概率值 (即分类器输出)，y 表示样本真实的类别标签。对于正例样本 ($y = 1$)，交叉熵损失函数的值等于 $log\hat{y}$；对于反例样本 ($y = 0$)，交叉熵损失函数的值等于 $\log(1-\hat{y})$。因此，交叉熵损失函数的目标是最小化模型预测与实际标签之间的差距，从而让模型能够更准确地进行分类。

交叉熵损失函数可以推广到多分类问题中，此时它的表达式略有不同。在多分类问题中，交叉熵损失函数可以写成以下形式：

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{K} y_i \log \hat{y}_i \tag{15}$$

[14]https://blog.csdn.net/u013289254/article/details/99694412

其中，K 表示类别的数量，y_i 表示第 i 个类别的真实标签，$\hat{y}_i$ 表示模型对于第 i 个类别的预测概率值。交叉熵损失函数的目标仍然是最小化预测与实际标签之间的差距，从而让模型能够更准确地进行分类。

Adversarial loss function

Adversarial loss function 是生成对抗网络 (GAN) 的标准损失函数。GAN 是由生成器 (Generator) 和判别器 (Discriminator) 组成的两个神经网络模型。生成器的目标是生成与真实数据相似的假数据，而判别器的目标是将真实数据与生成的假数据区分开来。Adversarial loss 使得 GAN 能够不断的改进生成器的性能，使其能够生成更加逼真的数据。

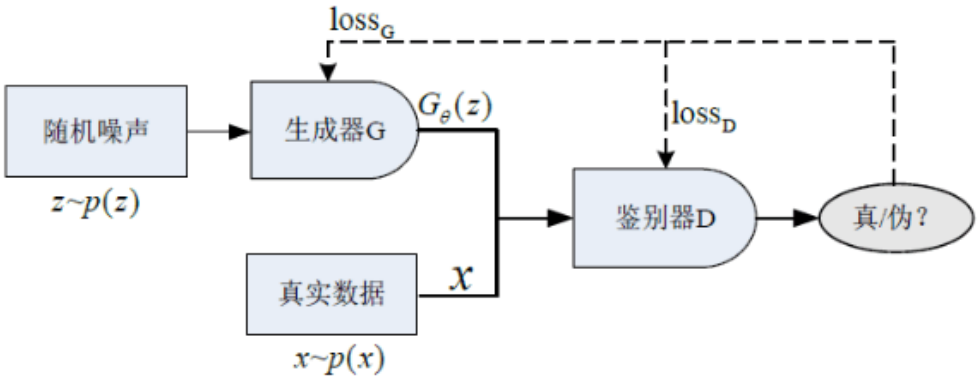


Figure 2: Computational flow and structure of the GAN.

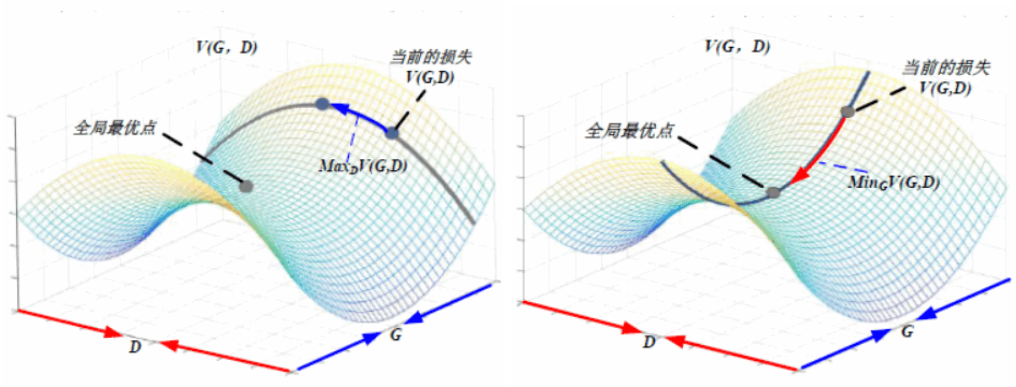
如 Fig. 2所示，在训练过程中，生成器和判别器相互竞争和对抗。生成器试图生成逼真的假数据 $G_\theta(z)$ 以欺骗判别器，而判别器则试图准确地判断真实数据和生成的假数据。Adversarial loss 通过衡量生成器生成的假数据被判别器识别为真实数据的程度，来指导生成器的训练。它的目标是最小化生成器生成的假数据与真实数据之间的差异，使得判别器难以区分它们。

一般而言，GAN 的目标函数为 $V(D,G)$Eq. 17

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_{noise}(z)}[\log(1-D(G(z)))] \tag{16}$$

其中，$E(*)$ 表示分布函数的期望值，$P_{data}(x)$ 代表着真实样本的分布，$P_{noise}(z)$ 是定义在低维的噪声分布，通过参数为 θ_g 的 G 映射到高维的数据空间得到 $P_g = G(z,\theta_g)$, 这些参数通过交替迭代的方法来进行优化，见 Fig. 3。Fig. 3a固定 G 参数不变，优化 D 的参数，即最大化 $\max V(D,G)$ 等价于 $\min[-V(D,G)]$。因此，D 的损失函数等价于 Eq. 16

$$J^D(\theta^D,\theta^G) = -\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] - \mathbb{E}_{\tilde{x}\sim p_g(x)}[\log(1-D(\tilde{x}))] \tag{17}$$



(a) D 的优化过程　　(b) G 的优化过程

Figure 3: The optimization of the GAN parameters

Adversarial loss 的具体形式可以根据具体的 GAN 架构和任务而定，常见的形式包括最小二乘损失 (Least Squares Loss)、二进制交叉熵损失 (Binary Cross-Entropy Loss) 等。这些损失函数的选择取决于具体的生成器和判别器结构以及任务的特点[15]。

Region loss function

Region loss function 是一种用于生物医学图像分割的损失函数。它是一种多功能的损失函数，可以同时考虑类别不平衡和像素重要性，并且可以很容易地实现为 softmax 输出和 RW(Region-wise) 地图之间的像素级乘法[16]。一般在一些涉及到异常检测的图像处理中，可以通过设计 region loss function 来使网络能够学习异常区域的位置，但是一般在这种情况下，需要设计一种可训练的异常区域引导框架。

在暗光增强领域，由于难以同时处理包括亮度、对比度、伪影和噪声在内的各种因素，该问题具有挑战性，在损失函数部分，结合使用 Region loss function 能获得不错的效果。一种基于深度学习的微光图像增强方法 (MBLLEN) 的损失函数的详细信息如 Fig. 4所示，Structural loss 损失旨在改善输出图像的视觉质量，Context loss 则通过关注图像中更加高层的信息来提高图像的视觉质量。

上述损失函数将图像作为一个整体。然而，对于弱光增强任务，我们需要更多地关注那些弱光区域。因此，作者提出了 Region loss 损失，它平衡了弱光和图像中其他区域的增强程度。为了做到这一点，作者首先提出了一个简单的策略来分离图像中的弱光区域。通过初步的实验，作者发现在所有像素中选择最暗的 40% 的像素可以很好地近似于弱光区域。人们也可以提出更复杂的方法来选择暗区，事实上在文献中有很多。最后，区域损失定义

[15]https://blog.csdn.net/qikaihuting/article/details/84950947

[16]https://arxiv.org/abs/2108.01405

如 Eq. 18

$$L_{Region} = w_L \cdot \frac{1}{m_L n_L} \sum_{i=1}^{n_L} \sum_{j=1}^{m_L} (\|E_L(i,j) - G_L(i,j)\|) + w_H \cdot \frac{1}{m_H n_H} \sum_{i=1}^{n_H} \sum_{j=1}^{m_H} (\|E_H(i,j) - G_H(i,j)\|) \tag{18}$$

其中，E_L 和 G_L 是增强图像和地面实况的暗光区域，E_H 和 G_H 是图像的剩余部分。

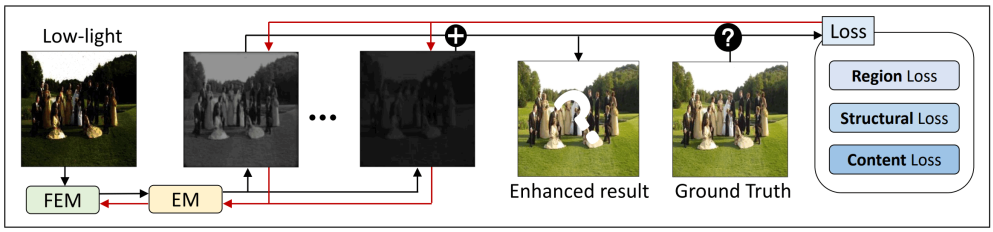


Figure 4: Data flow for training. The proposed loss function consists of three parts.

Reflectance loss function

Reflectance loss function 是计算机视觉领域中的一种损失函数，常用于图像去雾 (image dehazing) 任务中。(图像去雾是指通过算法去除图像中由雾霾引起的能见度降低的效果。Reflectance loss function 用于衡量生成的去雾图像与真实清晰图像之间的差异，以指导去雾模型的训练过程。)

Reflectance loss function 的核心思想是基于图像的反射率 (reflectance) 属性。反射率是指物体表面对入射光线的反射程度，与雾霾相关的信息主要存在于反射率中。通过计算生成的去雾图像的反射率与真实清晰图像的反射率之间的差异，可以量化生成图像的质量。

具体来说，Reflectance loss function 通常使用像素级别的差异度量函数，比如均方误差 (Mean Squared Error，MSE) 或结构相似性 (Structural Similarity，SSIM) 等来计算生成图像与真实图像之间的差异。优化过程的目标是最小化 Reflectance loss，使得生成的去雾图像能够与真实清晰图像更加接近。

通过引入 Reflectance loss 作为训练目标，去雾模型可以学习到更准确的雾霾分布和反射率信息，从而生成更清晰的去雾图像。这有助于提高图像去雾算法的性能和质量。

Consistency loss function

Consistency loss function 是一种在机器学习中使用的损失函数，用于提高模型的一致性。它主要用于半监督学习或自监督学习任务中，其中存在一些带有标签的数据（有目标变量的输入-输出对）和一些没有标签的数据（只有输入）。区别于全监督学习，半监督学习针对训练集标记不完整的情况：仅仅部分数据具有标签，然而大量数据是没有标签的。因

此，目前半监督学习的关键问题在于如何充分地挖掘没有标签数据的价值。主流的半监督学习方法有下面几种[17]:

(1) **自训练方法 (Self-Training)**。这是一种很直观的思路：既然大量数据是没有标签的，那么能否对这些数据生成一些伪标签 (Pseuo Labels)，对这些伪标签数据的训练从而利用原始的无标签数据。

(2) **基于对抗学习的方法 (Adversarial-Learning-based)**。这类方法基于一种假设，无标签数据 (unlabeled data) 通常具有和有标签数据 (labeled data) 在某种程度上类似的潜在标签。所以很自然地，可以采用 GAN 的图像模拟思路来进行对有标签数据进行类似于无标签数据的数据增强，进而利用无标签数据的潜在知识 (latent knowledge)。

(3) **基于一致性的方法 (Consistency-based)**。这类方法的核心思路在于一致性损失函数 Consistency loss function，对于进过扰动的无标签数据，模型应该对其做出一致性的预测——可以理解成一种利用无标签数据进行网络正则化的方法。

一致性损失函数的基本思想是，在模型对相似输入的处理应该具有一致性。它通过对输入进行一些变换或扰动，然后要求模型在这些变换后仍然能够产生相似的输出。如果模型能够在不同的变换下产生一致的输出，那么我们可以认为模型具有一定的鲁棒性和泛化能力。

具体来说，一致性损失函数可以定义为模型在原始输入和变换后输入上的输出之间的差异。常见的一致性损失函数包括均方差损失、平均绝对误差损失、KL 散度等。通过最小化一致性损失函数，模型被迫保持对输入的一致性响应，从而提高模型的泛化能力。

需要注意的是，一致性损失函数的具体形式和应用取决于具体的任务和模型架构。它通常与其他损失函数（如分类损失函数或重构损失函数）结合使用，以共同训练模型。

Color loss function

$$\mathcal{L}_c^i = \sum_p \angle\left((\mathcal{F}(I_i))_p, (\tilde{I}_i)_p\right) \tag{19}$$

Lim [3] 基于余弦相似性基于 [4] 中提出的损失函数 color-loss 进行了改进，针对于颜色恢复设计的损失函数，将 RGB 三通道看做三维向量，计算增强图像与 GT 之间的余弦距离 (见 Eq. 19), 因为在 reconstruction loss 中使用了 $\mathcal{L}_2$-loss, 但是 $\mathcal{L}_2$-loss 只是数值上测量色差，不能保证颜色向量的方向是相同的，可能导致颜色 mismatch, 所以使用了夹角的 loss,Lim [3] 修改之后, 见 Eq. 20。

$$\mathcal{L}_c = \sum_{k=1}^{3}\left(1 - \frac{1}{H_k \times W_k}\sum_{i\in(H_k,W_k)}\frac{\tilde{I}_k(i)\cdot I_k^*(i)}{\|\tilde{I}_k(i)\|^2\|I_k^*(i)\|^2}\right) \tag{20}$$

[17]https://blog.csdn.net/JYZhang_CVML/article/details/106817709

其中 H_k 和 W_k 分别是第 k 个拉普拉斯金字塔级别 (Laplacian pyramid level) 中的增强结果的高度和宽度。$\tilde{I}_k(i)$ 表示像素位置，$\cdot$ 表示内积。颜色越接近，颜色损失 L_c 就越接近于零。通过在恢复过程中考虑颜色向量的方向，能够成功地将增强结果的颜色属性恢复为与地面实况相似。

文章描述该方法是通过模糊输入图像与 GT 的纹理、内容，仅仅保存图像的颜色信息实现图像颜色的校正。实现过程比较简单，首先构建一个高斯模糊核，然后利用高斯模糊核作为卷积核对图像进行卷积运算，得到模糊后的图像；然后计算输入图像与 GT 的 MSE 作为损失函数。(pytorch 实现[18])

Laplacian loss function

拉普拉斯损失 (Laplacian loss) 是一种在计算机视觉任务中常用的损失函数，用于促进图像的平滑性。它通常用于图像生成、图像修复或图像去噪等任务中。其形式上一般定义为 Eq. 21。从形式上看，Laplacian loss function 主要工作是在 $\mathcal{L}_1$-loss 的基础上乘了一个 2 的幂次项，并且将多层的结果相加[19]。

Laplacian loss 的基本思想是鼓励生成的图像具有平滑的变化，以减少图像中的噪点或锐利边缘。它利用图像的拉普拉斯金字塔来比较生成图像和真实图像之间的结构差异。拉普拉斯金字塔 [5] 最初是为紧凑的图像表示而设计的，具有简单的操作，即缩小和扩展。许多研究人员试图将拉普拉斯金字塔应用于广泛的计算机视觉任务，例如纹理分析和合成。拉普拉斯金字塔具有线性可逆过程来重建原始输入而不损失任何信息以精确地细化局部细节的性质。Ghiasi 等人 [6] 提出了基于拉普拉斯金字塔的细化过程，以增强语义分割的低分辨率估计结果的局部细节。

$$\mathcal{L}_l = \sum 2^{i-1} \|\mathcal{L}^i(\hat{\alpha}) - \mathcal{L}^i(\alpha)\|_1 \tag{21}$$

具体来说，Laplacian loss 首先通过将生成图像和真实图像分别构建成拉普拉斯金字塔。拉普拉斯金字塔是一种多尺度表示，其中每个级别表示了原始图像与其上一级别平滑版本之间的细节差异。然后，Laplacian loss 计算两个图像金字塔之间的差异，通过比较它们在每个级别上的像素差异来量化结构上的差异。

通过最小化 Laplacian loss，生成的图像被鼓励具有更平滑的结构，从而减少图像中的噪点和锐利边缘。

需要注意的是，具体的 Laplacian loss 的实现方式可能因任务而异。在某些情况下，可以使用像素级别的差异或特征级别的差异作为损失度量。此外，Laplacian loss 通常与其他损失函数（如像素级别的差异、感知损失或对抗损失）结合使用，以共同训练生成模型。

[18] https://blog.csdn.net/ztzi321/article/details/101424448

[19] https://blog.csdn.net/lanceloter/article/details/121925904

1.1.2 细节

Table 1给出了最近几年主流的基于深度学习的 LLIE 方案，并从不同角度对其进行了划分。

1.2 Paper reading

1.2.1 Switching gaussian mixture variational rnn for anomaly detection of diverse cdn websites

2 个人工作进展

2.1 梳理损失函数

这一周的工作主要是接着梳理上一周未梳理完成的损失函数

1) **Consistency loss function** 一般用于半监督学习中，用于对 unlabeled data 进行变换和扰动，如果模型在这些不同的变换下，结果相似或者一致，就可以说明模型具有一定的鲁棒性和泛化性。同样，其没有具体的表现形式，取决于具体的任务和模型架构，一般情况下不单独使用，而是与其他损失函数结合使用，共同训练模型。此外总结了三种主流的半监督学习方法。

2) **Color loss function** 是一种实现图像颜色的校正的损失函数。一般结合其他损失函数使用，在结合 $\mathcal{L}_2$-loss 时，因为 $\mathcal{L}_2$-loss 只是能够从数值上测量色差，无法保证颜色向量的方向是相同的，这可能会导致颜色 mismatch，Color loss 需要计算夹角余弦。

3) **Laplacian loss function** 则是衡量预测结果与 GT 之间对应像素上的颜色差异。使用这个损失能够得到更好的结果。它还能捕捉全局以及局部的差异。此外，额外去了解了一下拉普拉斯金字塔的详细原理。

从数学角度看，图像的高斯模糊过程就是图像与正态分布做卷积，在二维卷积里面，一个正方形高斯核的值中间高、四周边缘低，大致呈一个“圆形”，拉普拉斯金字塔与高斯金字塔相反，是个倒着的金字塔，也就是图片尺度逐级变大，但是这个并不是单纯把图像放大，它要计算 [原图像] 和 [下采样、上采样后图像] 两者差值，得到的结果尺度逐渐变大，形成一个金字塔。高斯金字塔下采样过程中丢失的信息在上采样后并不能完全恢复，丢失信息的情况依然存在，如果直观把原图和下采样、上采样后的图片进行对比，会发现后者仍然模糊，此时用前者减去后者得到的新图像，形成的就是拉普拉斯金字塔。这个相减的过程，能够更突出显示图像的像素信息，因为高斯核属于低通滤波，图像经过这个操作后得到的结果更加平滑，边界信息相比原图没那么明显（因为图像中高频的边缘部分都被过滤），而原图这些边界信息较为明显，至于非边界部分信号变化不大，因此相减后得到的图

	Method	Learning	Network Structure	Loss Function	Training Data	Testing Data	Evaluation Metric	Format	Platform	Retinex
2017	LLNet	SL	SSDA	SRR loss	simulated by Gamma Correction & Gaussian Noise	simulated self-selected	PSNR SSIM	RGB	Theano	
2018	LightenNet	SL	four layers	L_2 loss	simulated by random illumination values	simulated self-selected	PSNR MAE SSIM User Study	RGB	Caffe MATLAB	✓
	Retinex-Net	SL	multi-scale network	L_1 loss invariable reflectance loss smoothness loss	LOL simulated by adjusting histogram	self-selected	-	RGB	TensorFlow	✓
	MBLLEN	SL	multi-branch fusion	SSIM loss perceptual loss region loss	simulated by Gamma Correction & Poisson Noise	simulated self-selected	PSNR SSIM AB VIF LOE TOMI	RGB	TensorFlow	
	SCIE	SL	frequency decomposition	L_2 loss L_1 loss SSIM loss	SCIE	SCIE	PSNR FSIM Runtime FLOPs	RGB	Caffe MATLAB	
	Chen et al.	SL	U-Net	L_1 loss	SID	SID	PSNR SSIM	Raw	TensorFlow	
	Deepexposure	RL	policy network GAN	deterministic policy gradient adversarial loss	MIT-Adobe FiveK	MIT-Adobe FiveK	PSNR SSIM	Raw	TensorFlow	
2019	Chen et al.	SL	siamese network	L_1 loss self-consistency loss	DRV	DRV	PSNR SSIM MAE	Raw	TensorFlow	
	Jiang and Zheng	SL	3D U-Net	L_1 loss	SMOID	SMOID	PSNR SSIM MSE	Raw	TensorFlow	
	DeepUPE	SL	illumination map	L_1 loss smoothness loss color loss	retouched image pairs	MIT-Adobe FiveK	PSNR SSIM User Study	RGB	TensorFlow	✓
	KinD	SL	three subnetworks U-Net	reflectance similarity loss illumination smoothness loss mutual consistency loss L_1 loss L_2 loss SSIM loss texture similarity loss illumination adjustment loss	LOL	LOL LIME NPE MEF	PSNR SSIM LOE NIQE	RGB	TensorFlow	✓
	Wang et al.	SL	two subnetworks pointwise Conv	L_1 loss	simulated by camera imaging model	IP100 FNF38 MPI LOL NPE	PSNR SSIM NIQE	RGB	Caffe	✓
	Ren et al.	SL	U-Net like network RNN dilated Conv	L_2 loss perceptual loss adversarial loss	MIT-Adobe FiveK with Gamma correction & Gaussion noise	simulated self-selected DPED	PSNR SSIM Runtime	RGB	Caffe	
	EnlightenGAN	UL	U-Net like network	adversarial loss self feature preserving loss	unpaired real images	NPE LIME MEF DICM VV BBD-100K ExDARK	User Study NIQE Classification	RGB	PyTorch	
	ExCNet.	ZSL	fully connected layers	energy minimization loss	real images	$IE_{ps}D$	User Study CDIQA LOD	RGB	PyTorch	
2020	Zero-DCE	ZSL	U-Net like network	spatial consistency loss exposure control loss color constancy loss illumination smoothness loss	SICE	SICE NPE LIME MEF DICM VV DARK FACE	User Study PI PNSR SSIM MAE Runtime Face detection	RGB	PyTorch	
	DRBN	SSL	recursive network	SSIM loss perceptual loss adversarial loss	LOL images selected by MOS	LOL	PSNR SSIM SSIM-GC	RGB	PyTorch	
	Lv et al.	SL	U-Net like network	Huber loss SSIM loss perceptual loss illumination smoothness loss	simulated by a retouching module	LOL SICE DeepUPE	User Study PSNR SSIM VIF LOE NIQE #P Runtime Face detection	RGB	TensorFlow	✓
	Fan et al.	SL	four subnetworks U-Net like network feature modulation	mutual smoothness loss reconstruction loss illumination smoothness loss cross entropy loss consistency loss SSIM loss gradient loss ratio learning loss	simulated by illumination adjustment, slight color distortion, and noise simulation	simulated self-selected	PSNR SSIM NIQE	RGB	-	✓
	Xu et al.	SL	frequency decomposition U-Net like network	L_2 loss perceptual loss	SID in RGB	SID in RGB self-selected	PSNR SSIM	RGB	PyTorch	
	EEMEFN	SL	U-Net like network edge detection network	L_1 loss weighted cross-entropy loss	SID	SID	PSNR SSIM	Raw	TensorFlow PaddlePaddle	
	DLN	SL	residual learning interactive factor back projection network	SSIM loss total variation loss	simulated by illumination adjustment, slight color distortion, and noise simulation	simulated LOL	User Study PSNR SSIM NIQE	RGB	PyTorch	
	LPNet	SL	pyramid network	L_1 loss perceptual loss luminance loss	LOL SID in RGB MIT-Adobe FiveK	LOL SID in RGB MIT-Adobe FiveK MEF NPE DICM VV	PSNR SSIM NIQE #P FLOPs Runtime	RGB	PyTorch	
	SIDGAN	SL	U-Net	CycleGAN loss	SIDGAN	SIDGAN	PSNR SSIM TPSNR TSSIM ATWE	Raw	TensorFlow	
	RRDNet	ZSL	three subnetworks	retinex reconstruction loss texture enhancement loss noise estimation loss	-	NPE LIME MEF DICM	NIQE CPCQI	RGB	PyTorch	✓
	TBEFN	SL	three stages U-Net like network	SSIM loss perceptual loss smoothness loss	SCIE LOL	SCIE LOL DICM MEF NPE VV	PSNR SSIM NIQE Runtime #P FLOPs	RGB	TensorFlow	✓
	DSLR	SL	Laplacian pyramid U-Net like network	L_2 loss Laplacian loss color loss	MIT-Adobe FiveK	MIT-Adobe FiveK self-selected	PSNR SSIM NIQMC NIQE BTMQI CaHDC	RGB	PyTorch	

Table 1: Summary of essential characteristics of representative deep learning-based methods, including learning strategies, network structures, loss functions, training datasets, testing datasets, evaluation metrics, data formats of input, and whether the models are Retinex-based or not. "simulated" means the testing data are simulated by the same approach as the synthetic training data. "self-selected" stands for the real-world images selected by the authors. "#P" represents the number of trainable parameters. "-" means this item is not available or not indicated in the paper.

像更关注边界的信息，或者说对高频信号更为敏感。

2.2 复现 KinD 代码

KinD[20]采用的是类 U-Net 网络，且对比使用了多种损失函数，采用的数据集相对较少，且环境配置相对简单，具体见 Tab. 1。同时，KinD [7] 目前引用数量为 543，采用的网络架构较为经典，即 Retinex-based 的思想。

2.2.1 Requirement

```
Python
Tensorflow >=1.10.0
numpy, PIL

```

2.2.2 Train

KinD Network 分为三部分 (Fig. 5)：(1) 图像分解网络：Layer Decomposition Net;(2) 反射分量纠正网络：Reflectance Restoration Net;(3) 光照分量纠正网络：Illumination Adjustment Net。

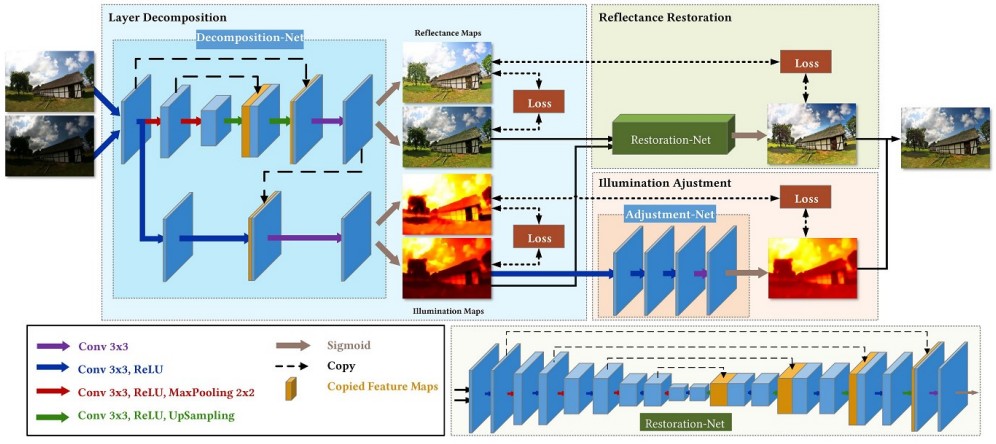


Figure 5: The architecture of our KinD network. Two branches correspond to the reflectance and illumination, respectively. From the perspective of functionality, it also can be divided into three modules, including layer decomposition, reflectance restoration, and illumination adjustment.

以暗光/正常光照图像 (I_{low}/I_{high}) 对作为训练样本,Layer Decomposition Net 对 (I_{low}/I_{high}) 依次进行分解，得到光照分量 L_{low}、L_{high} 和反射分量 R_{low}、R_{high}。再通过 Reflectance

[20]https://github.com/zhangyhuaee/KinD

Restoration Net 和 Illumination Adjustment Net 得到 $\tilde{R}_{low}$ 和 $\tilde{L}_{low}$ 。

Layer Decomposition Net

Layer Decomposition Net 有两个分支，一个分支用于预测反射分量，另一个分支用于预测光照分量，反射分量分支以五层 Unet 网络为主要网络结构，后接一个卷积层和 Sigmoid 层。光照分量分支由三个卷积层构成，其中还利用了反射分量分支中的特征图，具体细节可参考论文。

Layer Decomposition Net 对 (I_l/I_h) 依次进行分解，得到光照分量 L_l、L_h 和反射分量 R_l、R_h。

从 Fig. 5可以看出，KinD 的损失函数主要由三部分损失构成，它们分别是层分解部分损失、反射重建部分损失以及亮度调整部分损失。

层分解 (Layer Decomposition Net) 部分损失定义如下：

$$\mathcal{L}^{LD} = \mathcal{L}^{LD}_{rec} + 0.01\mathcal{L}^{LD}_{rs} + 0.15\mathcal{L}^{LD}_{is} + 0.2\mathcal{L}^{LD}_{mc} \tag{22}$$

其中，$\mathcal{L}^{LD}_{rs} = \|R_l - R_h\| 1$ 表示反射相似性损失 (Reflectance Similarity)，即短曝光与长曝光图形的反射图应该是相同的；

$$\mathcal{L}^{LD}_{is} = \left\| \frac{\nabla L_l}{\max(|\nabla I_l|, \varepsilon)} \right\|_1 + \left\| \frac{\nabla L_h}{\max(|\nabla I_h|, \varepsilon)} \right\|_1$$

表示亮度平滑损失约束 (Illumination Smoothness)，它度量了亮度图与输入图像之间的相对结构，边缘区域惩罚较小，平滑区域惩罚较大；$\mathcal{L}^{LD}_{mc} = \|M \circ \exp(-c \cdot M)\|_1, M = \nabla\mathcal{L}_l + \nabla\mathcal{L}_h$ 表示相互一致性约束 (Mutual Consistency)，它意味着强边缘得以保留，弱边缘被抑制；$\mathcal{L}^{LD}_{rec} = \|I_l - R_t \circ \mathcal{L}_l\|_1 + \|I_h - R_h \circ \mathcal{L}_h\|_1$ 表示重建损失 (Reconstruction Error)。

Layer Decomposition Net 复现代码如下

```
    # 设置一些训练所需的参数
    batch_size =10
    patch_size =48

    # 创建一个TensorFlow会话
    sess =tf.Session()

    # 定义输入的占位符，这些占位符用于接收低分辨率和高分辨率的输入图像。
    input_low =tf.placeholder(tf.float32, [None, None, None, 3], name='input_low
                                          ')
    input_high =tf.placeholder(tf.float32, [None, None, None, 3], name='
                                          input_high')
```

```
11
12      # 使用定义的模型构建计算图,DecomNet_simple是一个分解网络模型，它接受输入图像并
                                            输出反射和亮度组件
13      [R_low, I_low] =DecomNet_simple(input_low)
14      [R_high, I_high] =DecomNet_simple(input_high)
15
16      # 将反射和亮度组件拼接起来形成输出图像
17      # 这一步操作将反射和亮度组件进行通道拼接，以生成输出图像。
18      I_low_3 =tf.concat([I_low, I_low, I_low], axis=3)
19      I_high_3 =tf.concat([I_high, I_high, I_high], axis=3)
20      output_R_low =R_low
21      output_R_high =R_high
22      output_I_low =I_low_3
23      output_I_high =I_high_3
24
25      # 定义损失函数
26
27      def mutual_i_loss(input_I_low, input_I_high):
28
29      # 互信息损失函数的定义
30        ...
31
32      def mutual_i_input_loss(input_I_low, input_im):
33      # 输入互信息损失函数的定义
34      ...
35
36      recon_loss_low =tf.reduce_mean(tf.abs(R_low *I_low_3 -input_low))
37      recon_loss_high =tf.reduce_mean(tf.abs(R_high *I_high_3 -input_high))
38      equal_R_loss =tf.reduce_mean(tf.abs(R_low -R_high))
39      i_mutual_loss =mutual_i_loss(I_low, I_high)
40      i_input_mutual_loss_high =mutual_i_input_loss(I_high, input_high)
41      i_input_mutual_loss_low =mutual_i_input_loss(I_low, input_low)
42
43      loss_Decom =1*recon_loss_high +1*recon_loss_low +0.01*equal_R_loss +0.2*
                                            i_mutual_loss +0.15*
                                            i_input_mutual_loss_high +0.15*
                                            i_input_mutual_loss_low
44
45      # recon_loss_low 和 recon_loss_high 是重构损失，用于衡量输出图像与输入图像之间
                                            的差异。
46      # equal_R_loss 是反射一致性损失，用于衡量两个不同尺度下的反射分量之间的一致性。
```

```
    # i_mutual_loss 是亮度互信息损失，用于鼓励亮度分量之间的一致性。
    # i_input_mutual_loss_high 和 i_input_mutual_loss_low 是输入亮度与反射之间的
                                        互信息损失，用于鼓励输入图像与反射分量
                                        之间的一致性。

```

最后定义优化器和训练操作

```
    # 我们使用 Adam 优化器来最小化损失函数，其中只更新 DecomNet 模型的可训练变量。
    lr =tf.placeholder(tf.float32, name='learning_rate')
    optimizer =tf.train.AdamOptimizer(learning_rate=lr, name='AdamOptimizer')
    var_Decom =[var for var in tf.trainable_variables() if 'DecomNet' in var.
                                        name]
    train_op_Decom =optimizer.minimize(loss_Decom, var_list=var_Decom)
    sess.run(tf.global_variables_initializer())
    saver_Decom =tf.train.Saver(var_list=var_Decom)

    # 加载数据集
    # 这里使用 glob 函数获取训练集的低分辨率和高分辨率图像的文件名，并进行排序。
    train_low_data =[]
    train_high_data =[]
    train_low_data_names =glob('./LOLdataset/our485/low/*.png')
    train_low_data_names.sort()
    train_high_data_names =glob('./LOLdataset/our485/high/*.png')
    train_high_data_names.sort()

    # 定义了一些辅助变量和文件夹路径
    # epoch 表示训练的总轮数，learning_rate 表示学习率，sample_dir 是保存样本图像
                                        的文件夹路径，checkpoint_dir 是保存模
                                        型检查点的文件夹路径。
    epoch =2000
    learning_rate =0.0001
    sample_dir ='./Decom_net_train/'
    checkpoint_dir ='./checkpoint/decom_net_train/'

```

最后开始训练循环

```
      for epoch in range(start_epoch, epoch):
      for batch_id in range(start_step, numBatch):
      # 获取一个批次的训练数据
      ...
```

```
5
6         # 执行训练操作，计算损失
7         _, loss =sess.run([train_op, train_loss], feed_dict={input_low:
                                            batch_input_low, input_high:
                                            batch_input_high, lr: learning_rate})
8
9         # 打印训练进度和损失
10        print("%s Epoch: [%2d] [%4d/%4d] time: %4.4f, loss: %.6f" % (train_phase,
                                            epoch +1, batch_id +1, numBatch, time
                                            .time() -start_time, loss))
11
12        # 每隔100个批次保存模型和样本图像
13        if (epoch +1) % 100 ==0:
14        print('Saving sample images...')
15        sample_results =sess.run([output_R_low, output_R_high, output_I_low,
                                            output_I_high], feed_dict={input_low:
                                             sample_input_low, input_high:
                                            sample_input_high})
16        save_images(sample_results, [batch_size, 1], sample_dir +'train_%d.png' % (
                                            epoch +1))
17
18        print('Saving model...')
19        saver.save(sess, checkpoint_dir +'model.ckpt', global_step=epoch +1)
20
21        # 每隔500个批次降低学习率
22        if (epoch +1) % 500 ==0:
23        learning_rate /=10
24
```

在每个 epoch 的训练过程中，当遍历完一个批次后，计算并打印损失值。

当达到一定条件时，例如每隔 100 个批次，保存模型和样本图像。首先，我用当前模型对一部分样本进行推断，并将结果保存为图像文件。然后，保存模型的检查点，以便在需要时恢复模型。

另外，每隔 500 个批次，将学习率除以 10，以实现学习率的衰减。

Reflectance Restoration Net

作者认为来自低光图像的反射率图比来自明亮光图像的反射率图更容易受到退化的干扰。利用更清晰的反射率作为混乱反射率的参考 (非正式的地面实况)。对于寻找 Restora-

tion 函数，目标简单定义如 Eq. 23

$$\mathcal{L}^{RR} := \left\|\hat{R} - R_h\right\|_2^2 - SSIM(\hat{R}, R_h) + \left\|\nabla\hat{R} - \nabla R_h\right\|_2^2 \tag{23}$$

Reflectance Restoration Net 复现代码如下

```
    # 定义一些超参数
    batch_size =4
    patch_size =384

    # 创建 TensorFlow 会话并配置 GPU 使用
    config =tf.ConfigProto()
    config.gpu_options.allow_growth =True
    sess=tf.Session(config=config)

    # 定义输入占位符
    input_decom =tf.placeholder(tf.float32, [None, None, None, 3], name='
                                        input_decom')
    input_low_r =tf.placeholder(tf.float32, [None, None, None, 3], name='
                                        input_low_r')
    input_low_i =tf.placeholder(tf.float32, [None, None, None, 1], name='
                                        input_low_i')
    input_high_r =tf.placeholder(tf.float32, [None, None, None, 3], name='
                                        input_high_r')

    # 构建模型的图结构
    [R_decom, I_decom] =DecomNet_simple(input_decom)
    decom_output_R =R_decom
    decom_output_I =I_decom
    output_r =Restoration_net(input_low_r, input_low_i)

```

loss_restoration = loss_square + loss_ssim + loss_grad：这一行代码定义了总的损失函数 loss_restoration，它是由三个部分的损失函数相加而成。loss_square 是图像修复前后像素差的平方和的均值，用来衡量修复后图像与原始图像的像素差异。loss_ssim 是结构相似性损失，用来衡量修复后图像与原始图像的结构相似性。loss_grad 是梯度损失，用来衡量修复后图像与原始图像的梯度差异。通过将这三个损失函数相加，可以综合考虑图像修复的像素准确性、结构相似性和梯度一致性。

```
    # 定义损失函数
    def grad_loss(input_r_low, input_r_high):
      input_r_low_gray =tf.image.rgb_to_grayscale(input_r_low)
```

```
    input_r_high_gray =tf.image.rgb_to_grayscale(input_r_high)
    x_loss =tf.square(gradient(input_r_low_gray, 'x') -gradient(
                                        input_r_high_gray, 'x'))
    y_loss =tf.square(gradient(input_r_low_gray, 'y') -gradient(
                                        input_r_high_gray, 'y'))
    grad_loss_all =tf.reduce_mean(x_loss +y_loss)
    return grad_loss_all

def ssim_loss(output_r, input_high_r):
    output_r_1 =output_r[:,:,:,0:1]
    input_high_r_1 =input_high_r[:,:,:,0:1]
    ssim_r_1 =tf_ssim(output_r_1, input_high_r_1)
    output_r_2 =output_r[:,:,:,1:2]
    input_high_r_2 =input_high_r[:,:,:,1:2]          ssim_r_2 =tf_ssim(
                                        output_r_2, input_high_r_2)
    output_r_3 =output_r[:,:,:,2:3]
    input_high_r_3 =input_high_r[:,:,:,2:3]
    ssim_r_3 =tf_ssim(output_r_3, input_high_r_3)
    ssim_r =(ssim_r_1 +ssim_r_2 +ssim_r_3)/3.0
    loss_ssim1 =1-ssim_r
    return loss_ssim1

loss_square =tf.reduce_mean(tf.square(output_r -input_high_r))
loss_ssim =ssim_loss(output_r, input_high_r)
loss_grad =grad_loss(output_r, input_high_r)
loss_restoration =loss_square +loss_grad +loss_ssim

```

lr = tf.placeholder(tf.float32, name='learning_rate')：这一行代码创建了一个占位符 lr，用于传入学习率。占位符是在 TensorFlow 中用于表示在运行时提供数据的节点。在训练过程中，可以通过向 lr 传递不同的学习率值来控制模型的学习速度。

```
# 定义优化器和训练操作
lr =tf.placeholder(tf.float32, name='learning_rate')
optimizer =tf.train.AdamOptimizer(lr)
train_op =optimizer.minimize(loss_restoration)

# 初始化变量并创建模型保存器
sess.run(tf.global_variables_initializer())
saver=tf.train.Saver(max_to_keep=50)

```

```
10        # 定义一些训练过程中需要使用的函数和变量
11        train_dataset =glob('./data/train/*')
12        steps_per_epoch =len(train_dataset) //batch_size
13        global_step =0
14        learning_rate =0.0001
15
```

最后开始训练循环，for epoch in range(20): 这个循环用于控制训练的迭代次数，每个迭代称为一个 epoch。在每个 epoch 中，数据集被随机打乱，以增加数据的随机性。这样，整个训练过程会逐步优化模型参数，以达到更好的图像修复效果。

第 2 行代码使用 random.shuffle 函数随机打乱训练数据集的顺序，以增加数据的随机性。

第 3 行使用嵌套循环用于遍历每 epoch 中的小批量数据。steps_per_epoch 是每个 epoch 中的步骤数量，通过将训练数据集的大小除以批量大小得到。

第 5 行代码调用了 load_data 函数，从训练数据集中加载一批数据。train_dataset[] 表示取出当前批次的训练数据，patch_size 是图像块的大小。

第 7 行代码通过调用 sess.run 函数执行了一次训练操作和损失函数的计算，并将结果赋值给变量 loss_value、loss_square_value、loss_ssim_value 和 loss_grad_value。

```
1         for epoch in range(20):
2           random.shuffle(train_dataset)
3         for batch_id in range(steps_per_epoch):
4           start_time =time.time()
5           input_low_r_data, input_low_i_data, input_high_r_data =load_data(
                                              train_dataset[batch_id*batch_size:(
                                              batch_id+1)*batch_size], patch_size)
6
7           _, loss_value, loss_square_value, loss_ssim_value, loss_grad_value =sess.
                                              run([train_op, loss_restoration,
                                              loss_square, loss_ssim, loss_grad],
                                              feed_dict={input_low_r:
                                              input_low_r_data, input_low_i:
                                              input_low_i_data, input_high_r:
                                              input_high_r_data, lr: learning_rate}
                                              )
8
9           duration =time.time() -start_time
10
11          global_step +=1
12
```

```
13          if batch_id % 100 ==0:
14            print('Epoch {:d}/{:d} Batch {:d}/{:d} loss = {:.6f}, loss_square = {:.
                                                6f}, loss_ssim = {:.6f}, loss_grad =
                                                {:.6f}, time = {:.2f}s'.format(epoch
                                                +1, 20, batch_id +1, steps_per_epoch,
                                                 loss_value, loss_square_value,
                                                loss_ssim_value, loss_grad_value,
                                                duration))
15
```

通过这段代码不难看出，其实现了一个图像训练模型的过程，包括了数据预处理、模型构建、损失函数定义、优化器设置以及训练过程的迭代。它使用了 TensorFlow 框架和一些自定义的模块和函数来完成图像修复任务的训练。

Illumination Adjustment Net

一般而言，图像的不同场景和区域通常在透视和反射属性上不同，比如户外街景可以简单的分为三个部分：天空、地面和前景对象，这三个区域一般在透视和反射属性上是不相同的。天空通常是平滑的，通常与地面上的物体有不同的光源。与车行道相比，其他前景对象通常更明亮，包含更丰富的细节。

由于不存在用于图像的地面实况光照级别。因此，为了满足不同的需求，需要一种机制来灵活地将一种光照条件转换为另一种光照条件。数据集中是成对的照明图。虽然不知道之间的确切关系，但是可以大致计算强度的元素级比率 $\alpha(L_t/L_s)$。这个比例可以作为指标来训练一个调整函数从一个光照条件 L_s 到另一个标准 L_t，如果调整光照的水平更高, 则 $\alpha > 1$, 否则 $\alpha \leq 1$。在测试阶段,α 可以由用户指定。该网络是轻量级的，包含 3 个 conv 层 (2 个 conv+ReLu 和 1 个 conv) 和 1 个 Sigmoid 层。我们注意到指标 α 是作为输入的一部分扩展到一个特性映射上。(代码中是把这个比值扩展成一个常数通道与输入图像合并变成 4 通道)。光照分量的纠正与反射分量纠正类似，这里同样使用高光照图光照分量 L_t，作为真值约束训练，网络预测得到的纠正后反射分量为 $\tilde{L}$，等式 Eq. 24是照明调节网的损失函数[21]。

$$\mathcal{L}^{IA} := \left\| \hat{L} - L_t \right\|_2^2 + \left\| \left| \nabla \hat{L} \right| - \nabla L_t \right\|_2^2 \tag{24}$$

```
1       # 设置一些训练相关的参数
2       batch_size =10
3       patch_size =48
4
```

[21]https://blog.csdn.net/xspyzm/article/details/106162581

```
    # 创建TensorFlow会话
    sess =tf.Session()

    # 定义输入占位符
    input_decom =tf.placeholder(tf.float32, [None, None, None, 3], name='
                                            input_decom')
    input_low_i =tf.placeholder(tf.float32, [None, None, None, 1], name='
                                            input_low_i')
    input_low_i_ratio =tf.placeholder(tf.float32, [None, None, None, 1], name='
                                            input_low_i_ratio')
    input_high_i =tf.placeholder(tf.float32, [None, None, None, 1], name='
                                            input_high_i')

```

下面的代码构建了图像处理的模型。首先通过 DecomNet_simple 函数构建了一个分解网络模型，将输入的图像分解为反射分量和光照分量，分别保存在 decom_output_R 和 decom_output_I 中。然后通过 Illumination_adjust_net 函数构建了一个光照调整网络模型，用于调整输入的光照分量，保存在 output_i 中。

```
    # 构建模型
    [R_decom, I_decom] =DecomNet_simple(input_decom)
    decom_output_R =R_decom
    decom_output_I =I_decom
    output_i =Illumination_adjust_net(input_low_i, input_low_i_ratio)

```

下列的代码定义了损失函数。grad_loss 函数计算输入的两个图像的梯度差异，并计算梯度差异的平均值。loss_grad 表示光照分量的梯度损失，loss_square 表示光照分量的均方差损失，而 loss_adjust 是将梯度损失和均方差损失相加得到的总体损失。

```
    # 定义损失函数
    def grad_loss(input_i_low, input_i_high):
      x_loss =tf.square(gradient(input_i_low, 'x') -gradient(input_i_high, 'x'))
      y_loss =tf.square(gradient(input_i_low, 'y') -gradient(input_i_high, 'y'))
      grad_loss_all =tf.reduce_mean(x_loss +y_loss)
      return grad_loss_all

    loss_grad =grad_loss(output_i, input_high_i)
    loss_square =tf.reduce_mean(tf.square(output_i -input_high_i))
    loss_adjust =loss_square +loss_grad

```

这些代码定义了一个 Adam 优化器，并使用它最小化总体损失 loss_adjust。train_vars 表示可训练的变量，这些变量将会在训练过程中更新。

```
    # 定义优化器和训练操作
    train_vars =tf.trainable_variables()
    train_op_adjust =tf.train.AdamOptimizer(learning_rate=0.0001).minimize(
                                            loss_adjust, var_list=train_vars)

```

这行代码使用 glob 函数加载训练数据的文件路径。'./data/train/*.png' 表示加载./data/train/路径下所有以.png 结尾的文件。

```
    # 初始化变量
    sess.run(tf.global_variables_initializer())

    # 加载训练数据
    train_data =glob('./data/train/*.png')

```

这段代码中的循环用于进行训练。外层循环是 30 个训练轮次 (epochs)，内层循环根据批大小将训练数据分成小批次进行训练。首先通过 get_image 函数加载批次的图像文件，并将它们转换为浮点数类型的数组。然后将图像数组归一化到 [0, 1] 的范围。接下来，将归一化后的低光照图像和差值图像分别作为输入的低光照图像和高光照图像。将它们扩展维度，以匹配占位符的形状。然后执行训练操作 train_op_adjust，并获取损失函数的值。

```
    # 开始训练过程
    for ep in range(30):
      for idx in range(len(train_data)//batch_size):
        batch_files =train_data[idx*batch_size:(idx+1)*batch_size]
        batch =[get_image(batch_file) for batch_file in batch_files]
        batch_images =np.array(batch).astype(np.float32)
        batch_low_images =batch_images /255.0
        batch_high_images =batch_images -batch_low_images
        batch_low_images =np.expand_dims(batch_low_images[:,:,:,0], axis=3)
        batch_high_images =np.expand_dims(batch_high_images[:,:,:,0], axis=3)
        batch_low_images_ratio =np.expand_dims(batch_low_images[:,:,:,0]/255.0,
                                            axis=3)
        _, loss_adjust_value, loss_grad_value, loss_square_value =sess.run([
                                            train_op_adjust, loss_adjust,
                                            loss_grad, loss_square], feed_dict={
                                            input_decom: batch_images,
```

```
                                        input_low_i: batch_low_images,
                                        input_low_i_ratio:
                                        batch_low_images_ratio, input_high_i:
                                         batch_high_images})

    # 打印训练过程中的损失值,%2d表示打印整数，占用两位字符的宽度；%.4f表示打印浮点
                                        数，保留四位小数
    print("Epoch: [%2d], step: [%2d], loss_adjust: [%.4f], loss_grad: [%.4f],
                                        loss_square: [%.4f]" % (ep+1, idx+1,
                                        loss_adjust_value, loss_grad_value,
                                        loss_square_value))

    # 保存模型
    saver.save(sess, "./model/model.ckpt")

```

Evaluation

3 下周工作计划

(1) 继续复现并详细分析 KinD 的项目代码，目前架构中的三个结构 (Fig. 5) 完成其中的训练和代码分析，因此，下周工作计划是尝试去评估模型，其中的 model.py 还没仔细去看，只是把训练的过程实施了一遍。

(2) 继续整理各类损失函数的区别。按照 Table1中整理的不同文献所采用的损失函数，弄清楚各类损失函数的作用。

(3) 试着更加在说明代码的时候让文档更加规范化，感觉目前的格式有些随意。

(4) 本周删去了关于“华为杯”的思考，这一章节，下一周关于第二个挑战可能会按照 Paper [8] 更多的提出自己的思考。

(1) 想要去了解一下“为什么高斯核用于图像模糊，拉普拉斯核用于图像锐化”的数学原理。

References

[1] Maxim Tatarchenko, Alexey Dosovitskiy, and Thomas Brox. Multi-view 3d models from single images with a convolutional network. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part VII 14*, pages 322–337. Springer, 2016.

[2] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*, pages 694–711. Springer, 2016.

[3] Seokjae Lim and Wonjun Kim. Dslr: Deep stacked laplacian restorer for low-light image enhancement. *IEEE Transactions on Multimedia*, 23:4272–4284, 2021.

[4] Ruixing Wang, Qing Zhang, Chi-Wing Fu, Xiaoyong Shen, Wei-Shi Zheng, and Jiaya Jia. Underexposed photo enhancement using deep illumination estimation. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6842–6850, 2019.

[5] Peter J Burt and Edward H Adelson. The laplacian pyramid as a compact image code. In *Readings in computer vision*, pages 671–679. Elsevier, 1987.

[6] Golnaz Ghiasi and Charless C Fowlkes. Laplacian pyramid reconstruction and refinement for semantic segmentation. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III 14*, pages 519–534. Springer, 2016.

[7] Yonghua Zhang, Jiawan Zhang, and Xiaojie Guo. Kindling the darkness: A practical low-light image enhancer. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 1632 – 1640, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Liang Dai, Wenchao Chen, Yanwei Liu, Antonios Argyriou, Chang Liu, Tao Lin, Penghui Wang, Zhen Xu, and Bo Chen. Switching gaussian mixture variational rnn for anomaly detection of diverse cdn websites. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 300–309, 2022.