

哈尔滨工业大学

<<模式识别与深度学习>>

实验6 实验报告:Double DIP论文的复

现与思考

(2020春季学期)

成员1:	1170500913 熊健羽
成员2:	1171000520 鲍克勤
成员3:	1171000622 章金凯

实验目标

在图像处理的任务中，往往需要我们从图像背景中提取出我们需要的重要信息，如我们希望能够从图片上提取出来马这个动物，因为对于这张图片来说马是主要元素，草原只是他的一个背景，在实际生活中往往有很多这样的应用，我们对此十分感兴趣，在查阅过资料以后，我们了解到使用进行前后景分离 DOUBLE DIP是一种简单高效的方法，所以我们希望能够自己动手实现double dip并进行前后景分离。

论文地址：<https://arxiv.org/pdf/1812.00467>

成员分工

1170500913熊健羽：

- 调研前后景分离知识，学习DOUBLE DIP，实现DOUBLE DIP核心网络结构代码，以及处理数据
- 撰写实验报告

1171000520鲍克勤：

- 调研前后景分离知识，学习DOUBLE DIP，训练DOUBLE DIP并调整参数，对网络结构微调
- 撰写实验报告

1171000622章金凯：

- 数据的后期处理与展示部分的代码编写，结果分析
- 撰写实验报告

实验环境

实验环境为免费的Baidu AI studio。

操作系统：Ubuntu 16.04

CPU：8核

RAM：32G

GPU：Tesla V100-SXM2

实验准备

- DIP

- DIP提出

DIP的提出源自于CVPR2018的一篇论文 Deep Image Prior。这篇论文主要的观点有如下几点：

- 之前的观点是认为卷积神经网络之所以在各类问题上能够取得好的结果是因为，其能够从大量数据集中学习到真实图片的先验分布，但是本文认为并不是这样的，本文作者说明神经网络这个结构本身就能够提取低层次统计分布，作者展示了一个随机初始化的神经网络就可以用来提取手工设计的先验分布特征，并在一些标准的逆问题上取得较好的效果
 - 另一方面作者认为网络的优秀表现并不只来自于其能够从数据中学习到真实的先验分布
 - 作者的工作显示了学习过程并不是建立图片数据先验分布所必须的。神经网络这样一种结构能够捕获大量的图片数据的统计量，且这是独立于训练过程的。在图片恢复问题中这种捕获统计量的能力表现的尤为重要，因为图片恢复需要图片数据的先验分布以恢复在图片下采样过程中损失掉的信息。
1. 深度神经网络可以被用于图片生成，将一个随机噪声映射到图片这就相当于从一个随即图片分布中进行采样，而作者在这里将注意力集中于受到一个“损坏”图片限制的生成问题，及图片降噪和超分辨率问题。作者，作者使用Unet结构研究未经训练的网络能捕获先验分布的能力
 2. 以上的问题都可以表示为如下能量函数，其中E取决于具体的应用场景，x0是对应的noisy，R是一个正则项用来捕捉自然图像的先验分布，本文中将R使用神经网络来捕捉先验分布，即R这项消失，然后替换为另一个神经网络映射的表述形式。

$$\begin{aligned} x^* &= \min_x E(x; x_0) + R(x) < 1 > \\ x^* &= f_{\theta^*}(x_0) \end{aligned}$$

其中

$$\theta^* = \arg \min_{\theta} E(f_{\theta}(z); x_0)$$

最后作者通过随机初始化参数然后进行梯度下降得到局部最优点，作者发现在迭代过程中每过一定循环次数扰动随机向量z会在某些实验中得到较好的效果。

因为上面的神经网络并没有经过一个数据集的训练，仅仅是针对需要生成的单个图片进行优化，所以这个神经网络的功能更像是一个hand-crafted prior。简单来说，DIP实现了用一个神经网络结构捕获图像先验知识的能力，这个能力可以应用在多的领域中。

- DIP结构

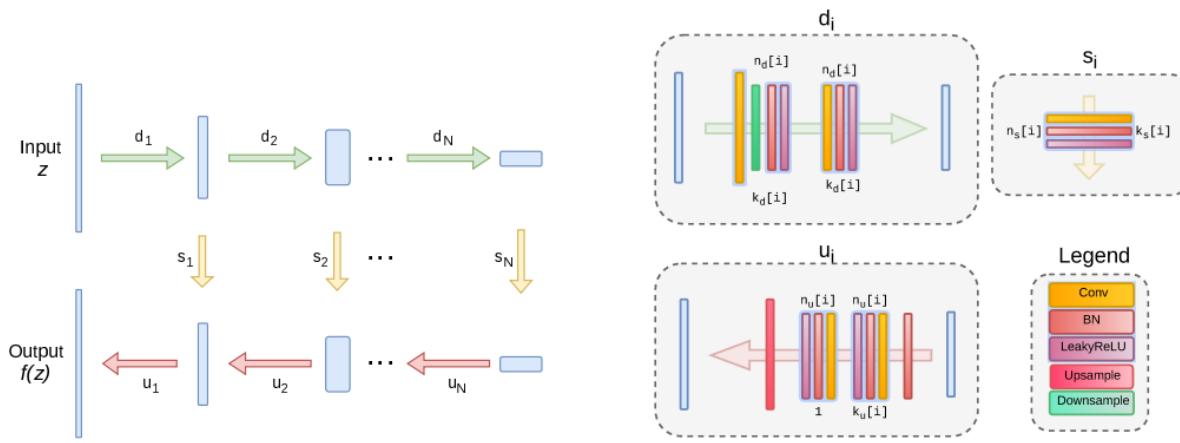


Figure 1: **The architecture used in the experiments.** We use “hourglass” (also known as “decoder-encoder”) architecture. We sometimes add skip connections (yellow arrows). $n_u[i]$, $n_d[i]$, $n_s[i]$ correspond to the number of filters at depth i for the upsampling, downsampling and skip-connections respectively. The values $k_u[i]$, $k_d[i]$, $k_s[i]$ correspond to the respective kernel sizes.

如图是DIP的结构，作者使用了Encode-Decoder模型，同时在编码解码的对应层中加入了跨层连接，非线性层使用了LReLU结构，上采样的时候使用了双线性的方式，padding不使用补零，最后优化器选择了ADAM优化器

作者首先通过卷积下采样压缩图像信息，使得图像保留较为低级的统计信息，然后再通过卷积上采样重构图像，重构过程中使用跨层连接将丢失掉的图像信息传递到解卷积层，以此来弥补信息的丢失。

• DOUBLE DIP

- DOUBLE DIP 原理

Double DIP主要介绍了一种图像分割技术，从字面意思来看其实用了多个DIP结构来捕获图像的先验信息，然后对图像进行分割。

文章的主要贡献在于提供了一种无监督学习进行图像分层的框架，使得每个图层内的图像元素是相互独立的并且简单的，Double DIP使用了DIP生成器网络的结构，上文中已经提到单个DIP网络结构足以捕获单个图像的低级统计数据，DIP网络的输入是随机噪声，训练DIP网络以重建单个图像，这种网络用语解决无监督去噪，超分辨率等问题非常有效果，作者观察到使用多个DIP网络来组合重建图像时，这些DIP倾向于分割图像，使得每个DIP输出相对于其他输出是独立而简单的，因此，作者使用多个DIP组合来进行图像分割，作者证明了这种方法适用于各种计算机视觉任务，如：图像去噪、图像和视频分割，水印去除等等。

作者在论文中提到，在一张自然图像中，小的 patch 是有很高的重复性的，这个也是很好理解的，纹理或者结构的尺度越小，重复的概率越大，都是一些很小的边界或者平滑区域，同时，根据数据统计发现，自然图像的纹理具有一定的规律和自相似性，同一区域的小 patch 的 empirical entropy 会小于不同区域的 empirical cross-entropy，基于自然图像纹理结构的内在相关性，作者提出了一种无监督的图层分离方法，将多个 DIP 应用于重建一幅融合图像时，这些 DIP 倾向于将这张复合图像进行分离，将一张融合图像，分成两张相对 simple 的图层。

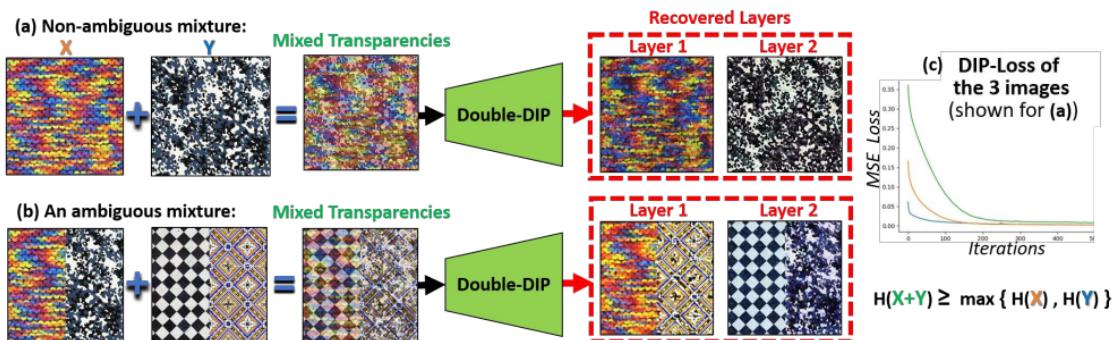


Figure 3: **The complexity of mixtures of layers vs. the simplicity of the individual components.** (See text for explanation).

作者在文中运用上图作为示例，上图中橙色和蓝色曲线表示单个图层的拟合 loss，绿色曲线表示混合图像的拟合 loss，从中可以看出，单个图层的拟合比混合图像的拟合要容易，这也间接说明了混合图像的熵比单个图层的熵要大，单个图像的纹理结构有更多的自相似性，混合图像破坏了这种内在的相似结构，所以拟合会更困难。所以为了对混合图像进行分离，我们可以假设图像Z是由图像X和Y组成的，我们可以假设X和Y与Z满足简单的线性组合关系，即 $Z = a * X + (1 - a) * Y$

另一方面，我们希望每个图层内部的相似性要尽可能的高，不同图层之间的相关性要尽可能地低，对此作者提出了如下地损失函数：

$$Loss = L_{rec} + \alpha \dot{L}_{exl} + \beta L_{reg} \quad (1)$$

其中第一项是重构损失，第二项是互斥损失，让分离出来的图层尽可能地不相关，第三项是一个与任务相关的正则项，用来约束融合地mask，不同的任务设置的正则项也不相同，比如图像分割中，mask 希望尽可能二值化，而在图像去雾中，mask 希望是平滑而且连续的。训练的时候位每个图层设置一个DIP 网络，输入都是随机噪声，期望通过DIP学习到图层先验信息，最后将两个图层通过一个mask进行融合，这个mask本身也是通过一个DIP网络得到的信息。

- DOUBLE DIP 网络结构

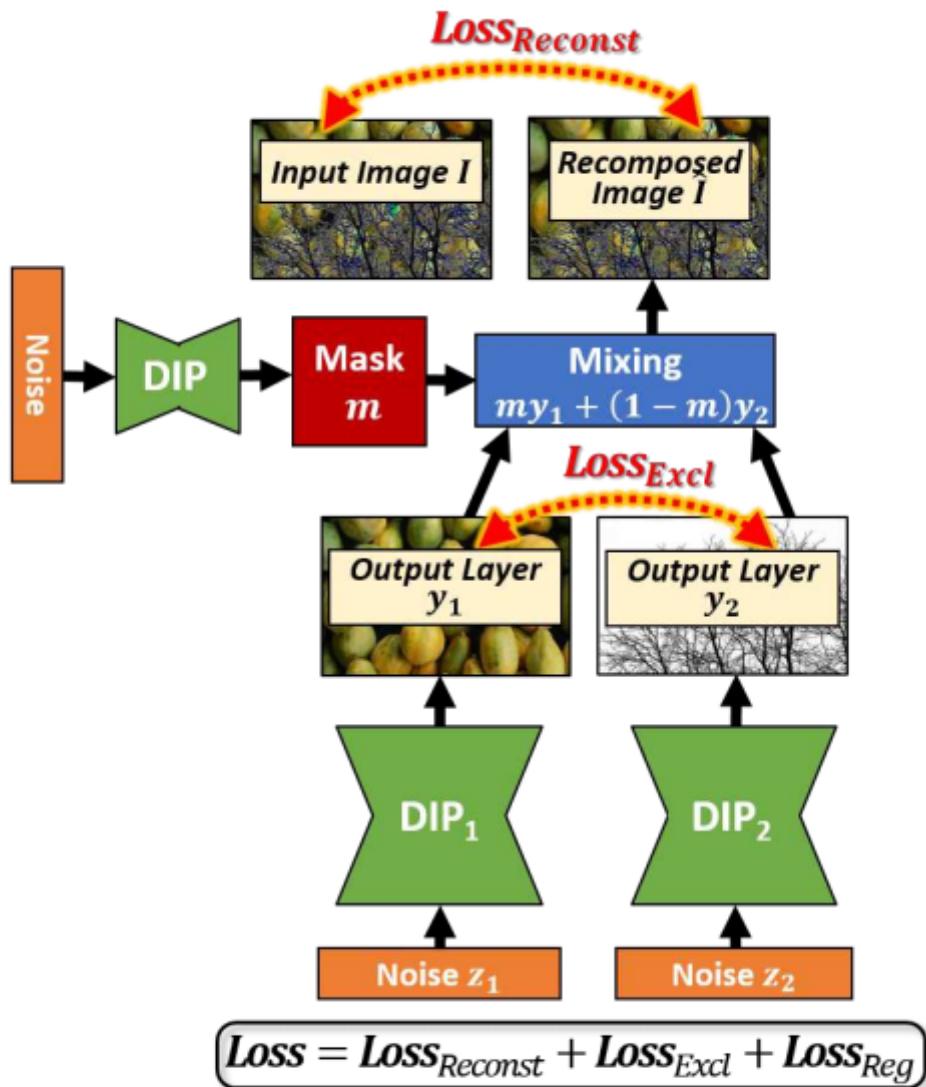


Figure 2: **Double-DIP Framework.** Two Deep-Image-Prior networks (DIP_1 & DIP_2) jointly decompose an input image I into its layers (y_1 & y_2). Mixing those layers back according to a learned mask m , reconstructs an image $\hat{I} \approx I$.

网络结构如图，如上文所介绍的，首先生成三个随机噪声一次输入到DIP网络，通过这个DIP网络将噪声一次转化成Mask信息和前景、后景信息，在前后景图像中使用互斥损失希望二者尽可能不同，将三者通过线性组合得到最终重构的图像，重构图像和原图像放在一起计算得到重构损失，最后损失函数还需要根据任务的不同加上一个正则项作为损失。

实验过程

数据处理

DIP的思想就是，从网络结构本身就是图像的先验，不需要额外的数据集，只需要图像本身，因此不存在训练数据集。

而对图像的读取则是直接使用了PIL库，相关代码如下：

```
from PIL import Image

seg = Segmentation()
downsample = (64, 48)
input_img = Image.open('./data/zebra.bmp')      # 读取图像
input_img = input_img.resize(downsample)          # 下采样到标准大小
input_img = np.array(input_img).astype(np.float32) / 255    # 归一化, [0, 255] -> [0,
1]
input_img = torch.from_numpy(input_img.transpose(2, 0, 1))
```

网络结构实现

单个DIP网络

如前面所述，double DIP的主要网络结构是DIP中的自编码器网络：

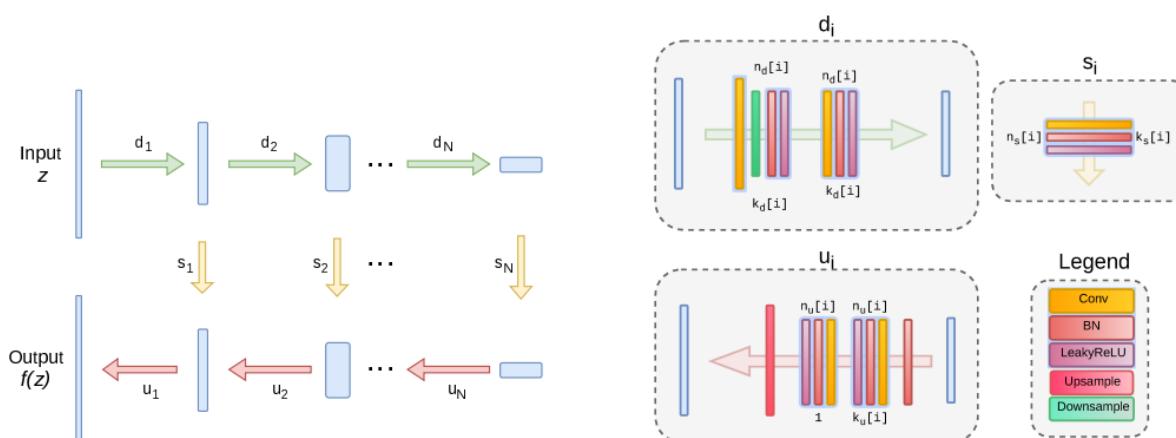


Figure 1: **The architecture used in the experiments.** We use “hourglass” (also known as “decoder-encoder”) architecture. We sometimes add skip connections (yellow arrows). $n_u[i]$, $n_d[i]$, $n_s[i]$ correspond to the number of filters at depth i for the upsampling, downsampling and skip-connections respectively. The values $k_u[i]$, $k_d[i]$, $k_s[i]$ correspond to the respective kernel sizes.

具体实现如下：

```
class DIP(nn.Module):
    def __init__(self,
                 down_channels=[8, 16, 32],
```

```

        up_channels=[8, 16, 32],
        skip_channels=[0, 0, 0],
        in_channels=2,
        out_channels=3
    ):
        super(DIP, self).__init__()
        assert len(down_channels) == len(up_channels)
        self.model = nn.Sequential()
        self.add_module('model', self.model)
        model_temp = self.model
        for i in range(len(down_channels)):

            model_temp.add_module('down_conv1', nn.Conv2d(in_channels=in_channels,
out_channels=down_channels[i],
                           kernel_size=3, stride=2, padding=1)) # 使用stride下采样
            model_temp.add_module('down_bn1',
nn.BatchNorm2d(num_features=down_channels[i]))
            model_temp.add_module('down_relu1', nn.LeakyReLU(0.2, inplace=True))
            model_temp.add_module('down_conv2',
nn.Conv2d(in_channels=down_channels[i], out_channels=down_channels[i],
                           kernel_size=3, stride=1, padding=1))
            model_temp.add_module('down_bn2',
nn.BatchNorm2d(num_features=down_channels[i]))
            model_temp.add_module('down_relu2', nn.LeakyReLU(0.2, inplace=True))

        deeper = nn.Sequential() # deeper layers

        skip = nn.Sequential() # skip connection
        if skip_channels[i] != 0:
            # 该层的skip
            skip.add_module('conv', nn.Conv2d(in_channels=down_channels[i],
out_channels=skip_channels[i],
                           kernel_size=3, stride=1, padding=1))
            skip.add_module('bn', nn.BatchNorm2d(num_features=skip_channels[i]))
            skip.add_module('relu', nn.LeakyReLU(0.2, inplace=True))
            model_temp.add_module('skip+deeper', Concat(skip, deeper)) # 把该层的deeper和skip连接
        else:
            model_temp.add_module('deeper', deeper)
        last_channel = up_channels[i + 1] if i != len(down_channels) - 1 else
down_channels[i]

        model_temp.add_module('up_bn1', nn.BatchNorm2d(num_features=last_channel +
skip_channels[i]))
        model_temp.add_module('up_conv1', nn.Conv2d(in_channels=last_channel +
skip_channels[i],
                           out_channels=up_channels[i],
                           kernel_size=3,
                           padding=1))
        model_temp.add_module('up_bn2',
nn.BatchNorm2d(num_features=up_channels[i]))
        model_temp.add_module('up_relu1', nn.LeakyReLU(0.2, inplace=True))
        model_temp.add_module('up_conv2', nn.Conv2d(in_channels=up_channels[i],
                           out_channels=up_channels[i],
                           kernel_size=3,
                           padding=1))
        model_temp.add_module('up_bn3',
nn.BatchNorm2d(num_features=up_channels[i]))
        model_temp.add_module('up_relu2', nn.LeakyReLU(0.2, inplace=True))

```

```

        model_temp.add_module('up_upsample', nn.Upsample(scale_factor=2,
mode='bilinear', align_corners=True))
        model_temp = deeper
        in_channels = down_channels[i]

    self.model.add_module('last_conv', nn.Conv2d(in_channels=up_channels[0],
                                              out_channels=out_channels,
                                              kernel_size=3,
                                              padding=1))
    self.model.add_module('sigmoid', nn.Sigmoid())

def forward(self, input):
    return self.model(input)

```

如代码中所示，每一层的通道数，按前向传播顺序设置为[8, 16, 32, 32, 16, 8]。

在自编码器中，涉及到Skip connection，在两个支路的连接处使用的是通道concat（即把两个支路的通道合在一起，得到的通道数是两者之和），因此还需要引入一个Concat结构：

```

class Concat(nn.Module):
    def __init__(self, model_1, model_2):
        super(Concat, self).__init__()
        self.model_1 = model_1
        self.model_2 = model_2

    def forward(self, input):
        out_1 = self.model_1(input)
        out_2 = self.model_2(input)

        if out_1.shape[2] != out_2.shape[2] or out_1.shape[3] != out_2.shape[3]: # 如果图像大小不相等
            min_shape_2 = min([out_1.shape[2], out_2.shape[2]])
            min_shape_3 = min([out_1.shape[3], out_2.shape[3]])

            diff2 = (out_1.size(2) - min_shape_2) // 2
            diff3 = (out_1.size(3) - min_shape_3) // 2
            out_1 = out_1[:, :, diff2 : min_shape_2 + diff2, diff3 : min_shape_3 + diff3]

            diff2 = (out_2.size(2) - min_shape_2) // 2
            diff3 = (out_2.size(3) - min_shape_3) // 2
            out_2 = out_2[:, :, diff2 : min_shape_2 + diff2, diff3 : min_shape_3 + diff3]

        return torch.cat((out_1, out_2), dim=1) # 通道连接

```

- Double DIP

虽然论文名叫Double DIP，实际上使用了三个DIP，一个用于生成前景背景分离的mask；一个用于生成前景；一个用于生成背景：

```

self.left_net = DIP(out_channels=3).to(device)
self.right_net = DIP(out_channels=3).to(device)
self.mask_net = DIP(out_channels=1).to(device)

```

• 训练过程

在本任务中，基于DIP的思想，训练过程即为对前后景分离问题的求解过程。

- 损失函数

Double DIP的可解释性的重点就在于它的损失函数。分为重构损失、正则损失、互斥损失。最后优化的方式是把三者按一定的系数累和，对三个DIP网络的参数进行联合优化。

重构损失

重构损失的直观理解是，三个网络生成的mask、fg、bg图片，能够合成接近于原始图像的图像。在这里直接使用合成图像与原始图像的L1 Loss：

```
l1_loss = nn.L1Loss().to(device)

def reconst_loss(input_img, recomp_img):
    ...
    重构损失
    :param recomp_img:
    :param self:
    :return:
    ...
    return l1_loss(input_img, recomp_img)
```

正则损失

正则损失是对mask的约束，也就是mask的先验。在不同的图像分离任务中，有着不同的先验。而在本实验的前后景分离任务中，mask应该是更加接近于二值化的，于是对mask有以下损失函数：

$$Loss_{Reg}(m) = \left(\sum_x |m(x) - 0.5| \right)^{-1}$$

代码实现如下：

```
def reg_loss(mask):
    ...
    正则损失(作为限制mask的先验)
    :param mask:
    :return:
    ...
    return 1 / l1_loss(mask, torch.ones_like(mask) / 2)
```

互斥损失

参与互斥损失计算的是前/背景图片，意在使这两个图片尽可能不相关，在这里Double DIP的作者参考了*Single Image Reflection Separation with Perceptual Losses* 这篇论文，使用以下公式计算其互斥损失：

$$L_{\text{excl}}(\theta) = \sum_{I \in \mathcal{D}} \sum_{n=1}^N \|\Psi(f_T^{\downarrow n}(I; \theta), f_R^{\downarrow n}(I; \theta))\|_F,$$

$$\Psi(T, R) = \tanh(\lambda_T |\nabla T|) \odot \tanh(\lambda_R |\nabla R|),$$

公式的大意是，分别计算两个图片的梯度图，各取tanh，进而规范到(-1, 1)的区间上。其乘积越小，则表明其在梯度域的差异越大。代码实现如下：

```

class ExclusionLoss(nn.Module):

    def __init__(self, level=3):
        """
        两个梯度的差别
        参考了以下论文：
        http://openaccess.thecvf.com/content\_cvpr\_2018/papers/Zhang\_Single\_Image\_Reflection\_CVPR\_2018\_paper.pdf
        """

        super(ExclusionLoss, self).__init__()
        self.level = level
        self.avg_pool = torch.nn.AvgPool2d(2, stride=2).type(torch.cuda.FloatTensor)
        self.sigmoid = nn.Sigmoid().type(torch.cuda.FloatTensor)

    def get_gradients(self, img1, img2):
        gradx_loss = []
        grady_loss = []

        for l in range(self.level):
            gradx1, grady1 = self.compute_gradient(img1)
            gradx2, grady2 = self.compute_gradient(img2)
            # alphax = 2.0 * torch.mean(torch.abs(gradx1)) /
            torch.mean(torch.abs(gradx2))
            # alphay = 2.0 * torch.mean(torch.abs(grady1)) /
            torch.mean(torch.abs(grady2))
            alphay = 1
            alphax = 1
            gradx1_s = (self.sigmoid(gradx1) * 2) - 1
            grady1_s = (self.sigmoid(grady1) * 2) - 1
            gradx2_s = (self.sigmoid(gradx2 * alphax) * 2) - 1
            grady2_s = (self.sigmoid(grady2 * alphay) * 2) - 1

            # gradx_loss.append(torch.mean(((gradx1_s ** 2) * (gradx2_s ** 2))) **
            0.25)
            # grady_loss.append(torch.mean(((grady1_s ** 2) * (grady2_s ** 2))) **
            0.25)
            gradx_loss += self._all_comb(gradx1_s, gradx2_s)
            grady_loss += self._all_comb(grady1_s, grady2_s)
            img1 = self.avg_pool(img1)
            img2 = self.avg_pool(img2)
        return gradx_loss, grady_loss

    def _all_comb(self, grad1_s, grad2_s):
        v = []
        for i in range(3):
            for j in range(3):
                v.append(torch.mean(((grad1_s[:, j, :, :] ** 2) * (grad2_s[:, i, :, :] ** 2))) ** 0.25)
        return v

```

```
def forward(self, img1, img2):
    gradx_loss, grady_loss = self.get_gradients(img1, img2)
    loss_gradxy = sum(gradx_loss) / (self.level * 9) + sum(grady_loss) /
(self.level * 9)
    return loss_gradxy / 2.0

def compute_gradient(self, img):
    gradx = img[:, :, 1:, :] - img[:, :, :-1, :]
    grady = img[:, :, :, 1:] - img[:, :, :, :-1]
    return gradx, grady
```

- 其他参数

经过不断尝试，选择了Adam优化方法，学习率设置为0.001。

实验结果与分析

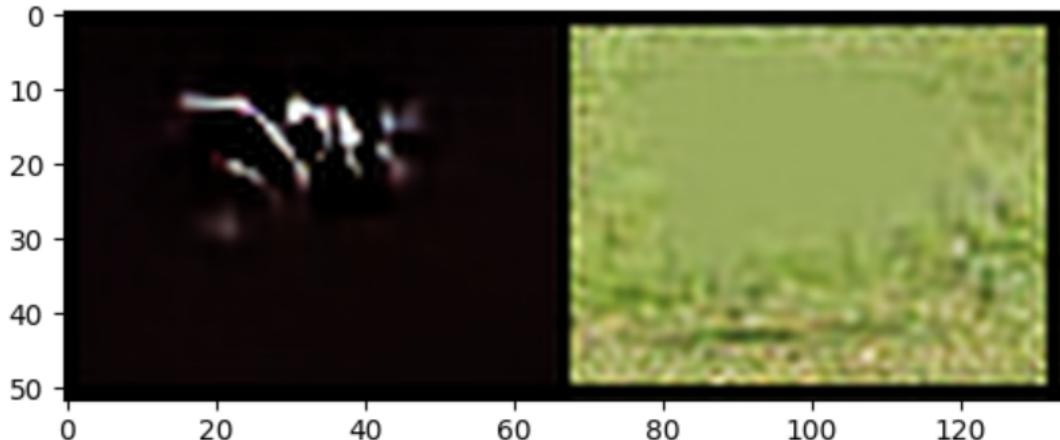
为了展现实验结果，在训练过程中，我设置每500个epoch输出一次以下三种图像：

- 前景图/背景图
- mask
- 合成图与原图对比

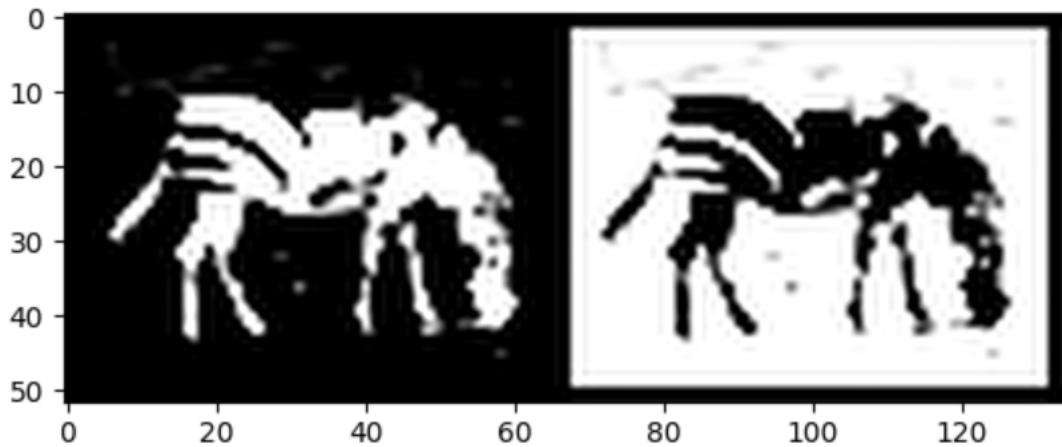
• 最终的效果

训练了5500个epoch，得到了如下结果。

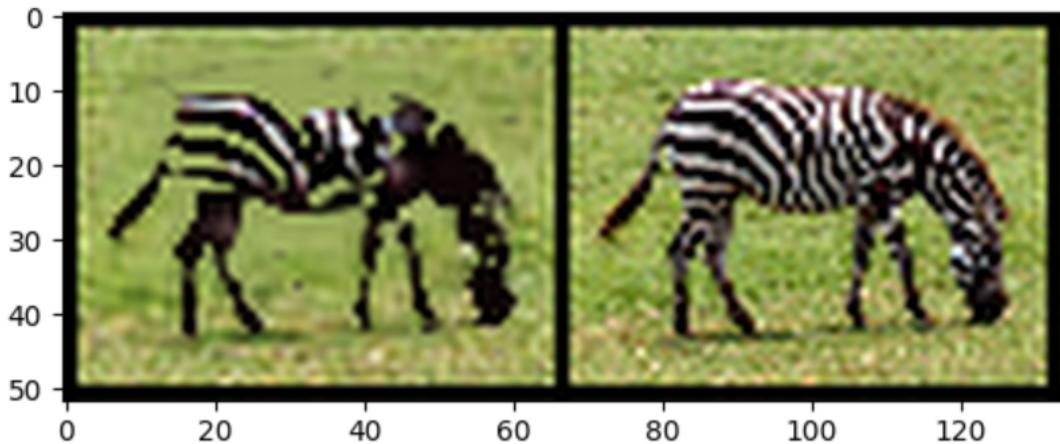
前景/背景图：



mask:



合成图与原图对比：

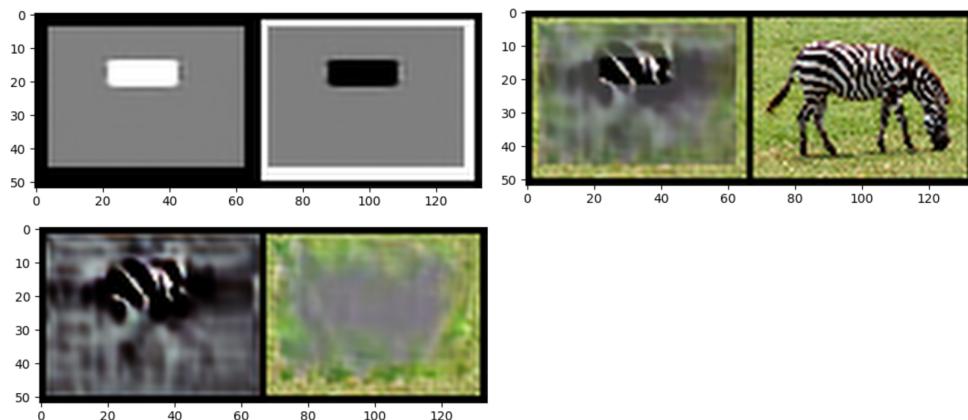


可以看到，基本实现了无任何先验下的前后景分离。

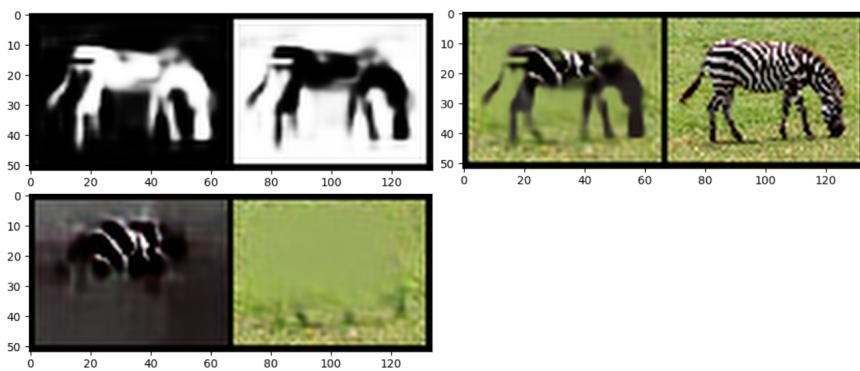
- 前景/背景图的对比主要体现了互斥损失作用的结果，可以看到前景和背景的相关性降到了很低；
- mask的结果基本实现了分离，但是由于时间原因，没有进一步去尝试更多的参数选择，所以斑马的某些白色条纹被误识别为背景部分。但整体轮廓分离良好。明显的二值化也证明了正则损失的作用；
- 合成图与原图对比体现了重构损失的作用。可以看到三个网络合成的图片在纹理和颜色上，的确接近于原图。

• 训练的中间结果

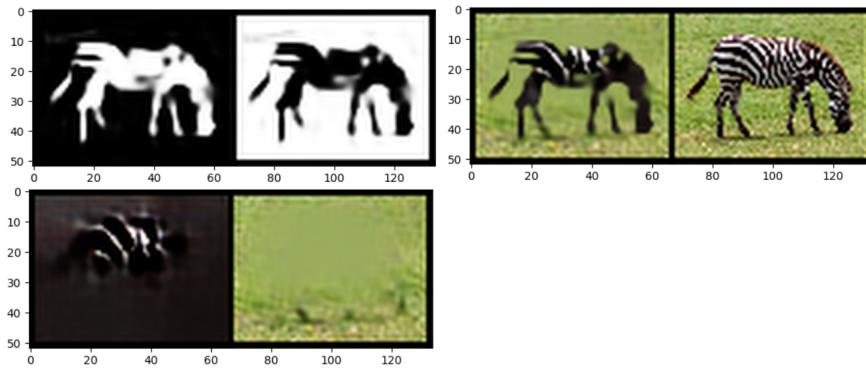
epoch=0:



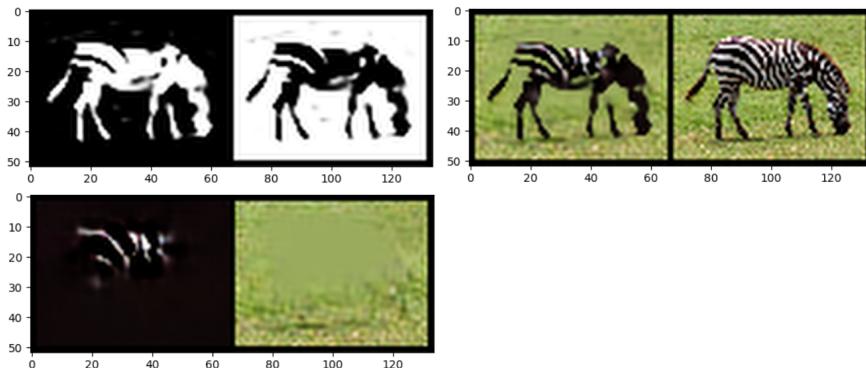
epoch=500:



epoch=1000:

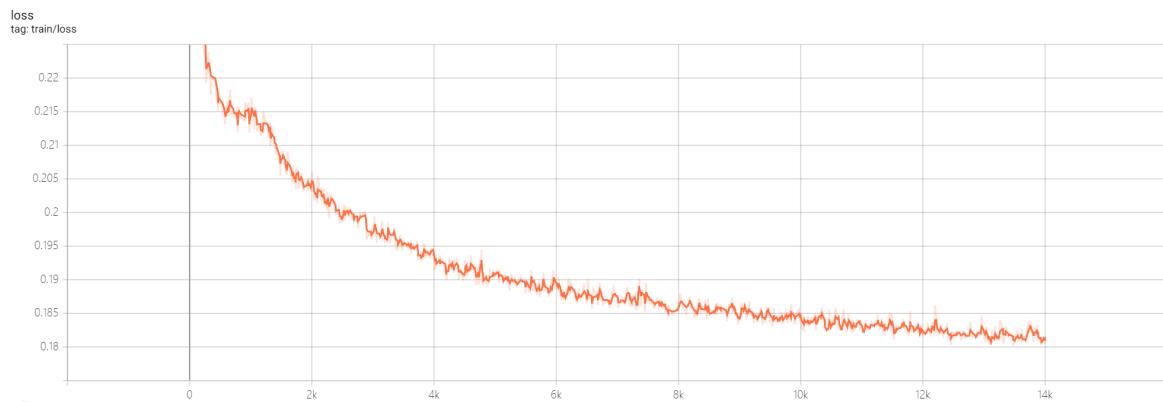


epoch = 3000:



从中间结果可以看出，在迭代次数很少的时候，Double DIP就已经大致实现了对前后景的分离任务，但是对细节边缘处的处理还不是很好。随着迭代次数的增多，mask逐渐精细，生成的fg/bg图像对斑马、草地的纹理拟合逐渐精确。但是对斑马的条纹的误识别逐渐出现。

以下是loss的变化曲线：



● 对结果的反思

对论文的复现结果基本达到了预期，实现了图片的前后景分离。但最终的结果还是有一点瑕疵，并没有达到论文中的完美结果。体现在测试图片中，是对斑马白色条纹的误识别。由于时间有限，我们没有对其进一步改进，但是我们尝试分析其原因，以及可能的改变方法：

- 可能是损失函数的各个部分的权重可能还需要调整，比如重构损失、正则损失、互斥损失之间的权重。或者是计算互斥损失时，两幅梯度图的权重等等。
- 从结果来看白色条纹被误认为是背景，也有可能是论文在应用过程中本身的缺陷。比如计算互斥损失时，论文中使用的方法只考虑了梯度图的特征。我们猜想，梯度图主要表现的是纹理特征，而如果能够加入一些对颜色的相关性的度量，也许会取得更好的效果。