Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder

Dezeming Family

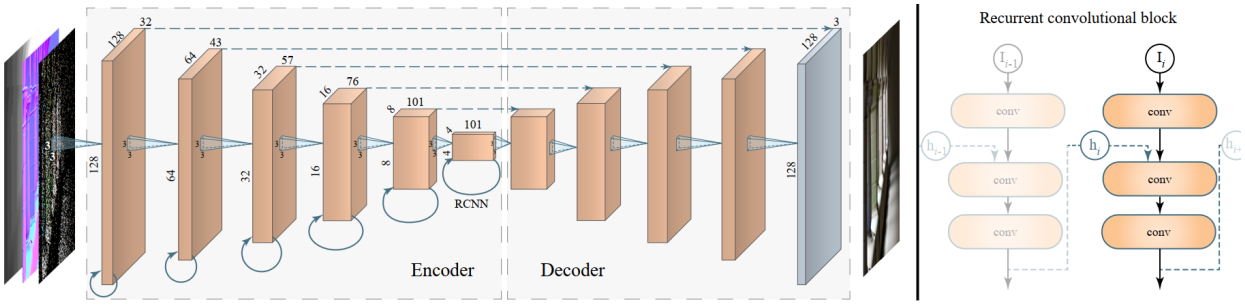
2021 年 11 月 22 日

目录

一　基本介绍

参考文献见 [1]，该论文的复现代码见 [2]。

神经网络结构如下：



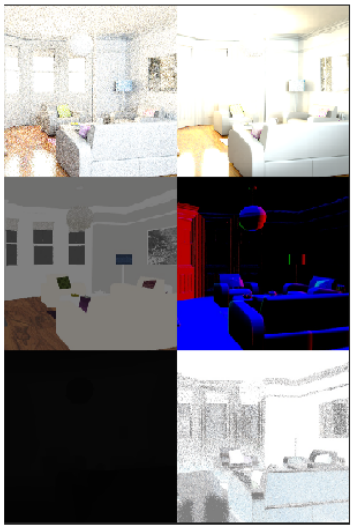
具体细节我们对照着源码再进行讲解。

二　数据集的准备

数据准备方面的代码见 [2] 的 data.py 文件。

为了方便起见，[2] 中把输入数据全都搞到了一张大图上，排列如下：

渲染图 shading 10spp（当然论文中要求1spp）	reference ray_shading
albedo 10spp（当然论文中要求1spp）	normal
depth	roughness



同时需要注意的是，对于循环神经网络，输入是一个序列，这个序列包含 7 张图像，这 7 张图像被放在同一个文件夹里；所有的序列包含在一个总的文件夹。因此，self.images 里存储的是大文件夹，里面有一系列的小文件夹，每个小文件夹里有 7 张大图。___getitem___ 函数会取出相应的图像序列，seq_images 表示存储了这个序列的文件夹，里面有这个序列的 7 张大图。

[2] 中，在 RAEData 类中设定了一个 width 和 height，表示我们在训练时使用的一张图像的大小，所以要把图像进行 resize，resize 为 (2*width,3*height)。

对于一个序列中的一个图像数据，是分为多层的，其中 RGB 三通道为三层、法向量有 (x, y, z) 三分量，因此也是三个通道，图像中，坐标深度设为一个通道，图像每个采样点采样到的物体粗糙度也是一个通道。因此，输入图像一共有 8 个通道。

首先注意论文中描述 shading normal 要乘以投影矩阵到视觉空间，并存储其 x 和 y 值，但是 [2] 中并没有这样做。注意深度的计算，[2] 把三通道取了个平均；roughness 也是如此。

我们的输入 shading 是把 shading 值与 albedo 解耦过的，网络的输出乘以 albedo 之后就可以得到用于成像的渲染结果。ray_shading 也是与 albedo 解耦过的。

之后 [2] 调用.permute 函数来交换几个维度。原来 A 的维度表示为 [序列图像数 =7, 图像高, 图像宽, 图像通道数 =8]，调用 permute 以后得到 A 的维度为 [序列图像数 =7, 图像通道数 =8, 图像高, 图像宽]。

三　损失函数

见 [2] 的 losses.py 文件。

我们需要注意的是在论文 [1] 中我们只会输出序列中最后一张图像，并计算最后一张图像与 reference 的损失值，而在 [2] 里相当于把序列中的 7 张图像都输出了，并把这七张图全都用于计算总体损失值（我们会在下一节关于网络结构这方面来验证这一点）。

损失函数分为三个部分：L_s，L_g 和 L_t。

l1_norm 就是计算 L_s，即两张图像之间的 L1 梯度，求和以后除以像素数（torch.numel(output)）。

HFEN 是计算 L_g，即计算输出的图像边缘信息和 reference 图像的边缘信息之间的差值。[2] 的 HFEN 函数调用了 LoG 函数，注意这里的 weight 相当于是高斯模糊了以后的边缘信息。

temporal_norm 函数是用来计算 L_t 的。根据论文中的叙述，在序列的后面为帧的损失函数分配更高的权重，以放大时序梯度，从而激励 RNN 块的时序训练，也就是说，对于一个序列的七张图，每个图的时序权重设定为 (0.011,0.044,0.135,0.325,0.607,0.882,1)。但是 [2] 的源码实现并不同，它把 output 序列里每张图与前一张相减，然后将绝对值求和并除以像素数来得到时序 loss。

按理来说，我认为就算给时序赋不同的权重，一个序列的七张图也应该只需要 6 个权重而已，因为 7 张图只有 6 个间隔，即只能计算出 6 个时序梯度。不知道这里论文中描述为什么会这样，不过无伤大雅。

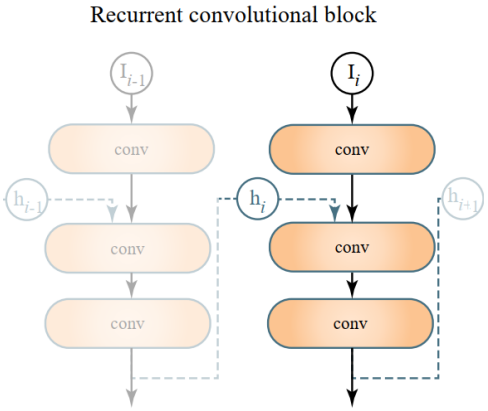
四　网络模块层和整体

见 model.py 的 RecurrentBlock：

我们可以把整个大的网络分成多个小模块，就像前面的图一样。根据位置和功能，分为 Encoder 块和 Decoder 块。在 [2] 中增加了一个瓶颈层 (bottleneck)，不过它跟 downsampling 并没有任何区别。

在 [2] 中，Encoder 部分就是由一堆 downsampling 模块构成的（包括最后边的一个 bottleneck）；Decoder 部分就是由一堆 upsampling 模块构成的。

一个 downsampling 模块正如下图所示。



在一个模块中，第一层卷积的输出与 hidden 通过.cat 在维度 1 上合并到一起，注意合并的方式，我们用下图为例：

```
tensor([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]], dtype=torch.int32)
tensor([[4, 4, 4],
        [5, 5, 5],
        [6, 6, 6]], dtype=torch.int32)
tensor([[1, 1, 1, 4, 4, 4],
        [2, 2, 2, 5, 5, 5],
        [3, 3, 3, 6, 6, 6]], dtype=torch.int32)
```

A

B

torch.cat((A, B), dim=1)

由于我们输入给卷积层的 dim1 是“一个序列里的图像数（经过多层卷积核以后会变多）”，因此相当于使输入“变厚”了。

下采样是由最大池化来完成的，定义在 RecurrentAE 里。

上采样的过程就不再包含循环模块了，就是首先上采样，然后通过两次卷积即可。

网络整体见 model.py 的 RecurrentAE 类：

该类的结构可以参考 [1] 的网络结构示意图，基本上没有多大的改动。有些过程例如 reset_hidden 在训练的时候才会用到，这里就先不说了。

五　训练

见 train.py 文件。

我们先看 main 函数。注意 data_loader 加载的图像数据长宽为 (256,256)。

RecurrentAE(8) 传入的 input_nc 值为 8，即正如我们前面描述的那样，输入的一个序列中有 8 张图。

在每个 epoch 里都会调用一次 train 函数，我们下面简单介绍一下这个函数。train 函数同样比较简单，主要就是调用了 train_sequence 函数，然后计算损失值，并启动反向传播。

train_sequence 函数接收的参数中，item 就是一个序列的 8 张图。

输入名为 inp，reference 名为 target。output_final target_final

5 1　第一个 for 循环

在第一个 for 循环中，为 inpi 和 gti 赋值，并生成 final_inp。注意 inpi 的维度和 gti 的维度是 [batch_size, 序列中的序列号（序列号从 0 到 6）, 该序列的第几张图像（张数从 0 到 7）, 图像高，图像宽]。

当当前的 j 为 0 时，也就是在输入第一个样本到网络之前，需要先将循环模块中的隐含层进行重置，即调用 model.reset_hidden。

对于第 j 次的输入，输出保存到 output_final[:, j, :, :, :] 中。

5 2　第二个 for 循环

该循环计算所有的 loss 项。

六　测试

test 与 train 的步骤基本相似，但是有几个地方需要注意。

这句代码 albedo = albedo[:, j, :, :, :] 会将序列中（序列号 0 到 6）第 j 个序列取出，然后通过 detach 取消梯度，并使用 squeeze 去除为 0 的梯度，此时 albedo 就变成了三维数据：[图像通道数，图像高，图像宽]，对于 albedo 来说一共有三个通道。之后再调用 permute 调整为 [图像高，图像宽，图像通道数]。

我们把每个生成的部分通过组合来形成最终的输出图像（也是一张大图）。

至于 RAE 的效果，大家可以参考 Optix 的实时光追系统里的实现。我个人推测 Optix 使用的 RAE 就是类似于这个版本的，用于对每张图像去噪，并通过重投影来利用时序信息。因为我目前没有详细了解过其实现方案，所以暂时不做过多评论。

本人复现 RAE 是为了分析其稳定性与光照的模糊程度，个人认为，不考虑使用重投影的 RAE，其时序效果远不如应用类似 TAA 时序重投影方法的 SVGF。

七　整体功能结构

本节从输入数据开始，介绍整个计算流程。

train 中会调用 train_sequence 函数，该函数用一个图像序列（7 张连续序列）进行训练。数组结构是：A[7,8,256,256];B[7,3,256,256];Albedo[7,3,256,256]。在 train_sequence 函数里，inp 就是 [batch_size,7,8,256,256]，每次喂入网络的数据是其中一张，大小是：[batch_size,8,256,256]。

在 RecurrentBlock 类的 downsampling 函数中，分为 l1 和 l2 部分。我们以输入网络的第一层为例，输入是 [batch_size,8,256,256]，通过 l1 卷积以后得到 [batch_size,32, 卷积后的高 =256, 卷积后的宽 =256]。通过 l1 卷积以后，再通过 cat 函数进行合并，如果是 7 张图像序列的第一张，则 hidden 就是初始化的值，否则就是 l2 的输出。l2 输入的大小为 [batch_size,64,256,256]，输出为 [batch_size,32,256,256]。

再看 RecurrentBlock 类的 reset_hidden 函数，该函数使用的 self.inp 的 shape 是 [batch_size,8,256,256]。hidden 的 size[1] 为 output_nc，size[2] 为池化以后的图像高，size[3] 为池化以后的图像宽。

运算完以后，最终输出的数据大小是：[batch_size,3,256,256]。

get_temporal_data 会计算多种损失率，然后加权组合：0.8*ls+0.1*lg+0.1*lt，用该损失率来进行 backward()。

训练好以后，就调用 test 函数。对于 albedo，首先通过 detach 将它与网络隔离开，使其不参与更新。然后将维度 0 进行 squeeze（此时的 batch_size 为 1）, 得到 [3,256,256]，再 permute 以后得到 [256,256,3]。ray 同样也是如此。下面这行代码：

```
1    final[:, :width, :] = og[:, :, :3]
```

表示 inp 的前三个维度，也就是被去噪的光照值。

最终输出的是一个 [256, 4*256, 3] 大小的图像。

八　整体功能结构

网络功能设计：整个计算流程设计如下。为了防止 HDR 范围超出像素值，我将 HDR 值除以 2，然后 clamp 到 [0,1] 之间。这样可能会损失精度。我考虑过直接定义 float 数组，但是这样网络的文件输入就会过于浪费时间，可能训练一个 epoch 就得一周，所以算了。

train 中调用 train_sequence 函数，该函数用一个图像序列（7 张连续序列）进行训练。数组结构是：A[7,8,256,256];B[7,3,256,256];Albedo[7,3,256,256]。在 train_sequence 函数里，inp 就是 [batch_size,7,8,256,256]，每次喂入网络的数据是其中一张，大小是：[batch_size,8,256,256]。

在 RecurrentBlock 类的 downsampling 函数中，分为 l1 和 l2 部分。输入是 [batch_size,8,256,256]，通过 l1 卷积以后得到 [batch_size,32, 卷积后的高 =256, 卷积后的宽 =256]。通过 l1 卷积以后，再通过 cat 函数进行合并，如果是 7 张图像序列的第一张，则 hidden 就是初始化的值，否则就是 l2 的输出。l2 输入的大小为 [batch_size,64,256,256]，输出为 [batch_size,32,256,256]。

RecurrentBlock 类的 reset_hidden 函数使用的 self.inp 的 shape 是 [batch_size,8,256,256]。hidden 的 size[1] 为 output_nc，size[2] 为池化以后的图像高，size[3] 为池化以后的图像宽。

运算完以后，最终输出的数据大小是：[batch_size,3,256,256]。然后我通过 get_temporal_data 会计算多种损失率，然后加权组合：0.8*ls+0.1*lg+0.1*lt，用该损失率来进行 backward()。

训练好以后调用 test 函数。对于 albedo，首先通过 detach 将它与网络隔离开，使其不参与更新。然后将维度 0 进行 squeeze（此时的 batch_size 为 1）, 得到 [3,256,256]，再 permute 以后得到 [256,256,3]。ray 同样也是如此。og 表示 inp 的前三个维度，也就是被去噪的光照值。

最终输出的是一个 [256, 4*256, 3] 大小的图像。

参考文献

[1] Chaitanya C R A, Kaplanyan A S, Schied C, et al. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder[J]. ACM Transactions on Graphics (TOG), 2017, 36(4): 1-12.

[2] https://github.com/AakashKT/pytorch-recurrent-ae-siggraph17