# Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings

Dezeming Family

2022 年 12 月 19 日

正常字体：表示论文的基本内容解释。
**粗体：表示需要特别注意的内容。**
红色字体：表示容易理解错误或者混淆的内容。
蓝色字体：表示额外增加的一些注释。
绿色字体：表示额外举的一些例子。

## 目录

## abstract

基于回归的算法已经证明，通过利用渲染中的一些副产物（例如，特征 buffers），可以很好地对蒙特卡洛（MC）渲染进行去噪。然而，当使用高阶模型来处理复杂情况时，这些技术通常会在输入中对噪声过拟合。

出于这个原因，已经提出了进行训练的监督学习方法，但它们使用了显式的滤波器，这限制了它们的去噪能力。

为了解决这些问题，我们提出了一种新颖的、有监督的学习方法，通过利用深度卷积神经网络（CNN）架构，允许滤波器的内核更复杂和和更一般化。

在我们的框架的一个实施中，CNN 将最终去噪的像素值直接预测为输入特征的高度非线性组合。在第二种方法中，我们引入了一种新颖的核预测网络，该网络使用 CNN 来估计用于从相邻像素计算每个去噪像素的局部加权核。我们在生成的数据上训练和估计我们的网络，并观察到对现有 MC 去噪器的改进，表明我们的方法可以很好地推广到各种场景。

我们通过分析我们架构的各个部分得出了一些结论，并确定了深度学习 MC 去噪的进一步研究的方向。


# 一 Introduction 和相关工作

尽管已经提出了多种图像空间 MC 去噪方法，但大多数最先进的技术都是使用回归框架 [Moon et al. 2014; Bitterli et al. 2016]，然而，这些进步是以不断增加的复杂性为代价的，而回报却在逐渐减少，这部分是因为高阶回归模型容易过拟合输入的噪声。

为了避免噪声干扰问题，Kalantari 等人最近提出了一种基于监督学习的 MC 去噪器，该去噪器通过一组有噪声的输入和相应 reference 进行训练。然而，该方法使用了相对简单的多层感知器（MLP）作为学习模型，并在少量场景上进行了训练，而且他们的方法将滤波器硬编码为联合双边或联合非局部方法，这限制了系统的灵活性。

我们提出了一种新颖的、有监督的学习方法，通过利用深度卷积神经网络（CNN）架构，允许滤波器的内核更复杂和和更一般化。我们产生了大量的数据用于训练，好处是 CNN 是强大的非线性模型，而且 CNN 估计的速度快，不需要手动调整或参数调整 (manual tuning or parameter tweaking)。虽然可能还有很多用途，但我们专注于静态图像的去噪。

贡献列为以下几个方面：

- 首次用深度学习方法进行的 MC 图像去噪。
- 以前方法都是对有噪像素的邻域加权平均，我们提出了新颖的核预测 CNN 架构，计算局部最优邻域权重，这为更好的训练收敛率提供了正则化。
- 我们探索并分析了我们系统的各种处理和设计决策，包括我们分别对图像的漫反射和镜面反射分量进行去噪的两个网络框架，以及一个简单的标准化 (normalization) 过程，该过程显著改进了我们的（以及以前的）对高动态范围图像的处理方法。

对于图像空间滤波，比较成功的都是基于 [Rushmeier and Ward 1994] 的非线性图像空间滤波器和辅助特征（法向量，albedo 等）。Sen and Darabi [2012] 利用有噪的辅助特征来执行联合双边滤波过程，关键在于他们利用样本的统计特征来计算滤波带宽。Moon 等人 [2014] 使用渐近偏差分析来实现滤波带宽的计算。在我们的系统中，训练过程隐式地学习各种辅助缓冲区的适当权重。

非局部均值滤波器（Buades et al. [2005]）对去噪 MC 渲染有着很大的吸引力，引入辅助特征的联合滤波方案进行去噪（[Rousselle et al. 2013; Moon et al. 2013; Zimmer et al. 2015]），这很大程度上是由于其通用性。BM3D 也是很好的方法，但是不容易使用辅助特征来去噪。

Kalantari[2015] 等人提出了基于学习的滤波方法，但是他们的网络使用固定的滤波器作为后端，因此具有与之而来的局限性。相比之下，我们提出了一种隐式学习滤波器的方案，从而产生更好的结果。

本工作与 RAE（Chakravarty et al. [2017]）是共同工作的，但是他们更注重可交互渲染，而我们重在高样本量高质量产业级的渲染。

应用 CNN 去噪需要解决一些问题：

- 训练一个网络时，输入是有噪声的，输出可能会过模糊，因为难以区分高频噪声和场景细节。
- 渲染的图像是高动态范围的，直接用来训练会导致权重不稳定。

通过对特征进行预处理，diffuse 和 specular 组件分离，能够在保证细节的同时去噪图像。

# 二　理论背景

这里我把所有的公式都贴上来（红色标记的公式是论文里没有给序号的，引用时我会说明）：

$$\widehat{\theta}_p = \underset{\theta}{\operatorname{argmin}}\, \ell(\bar{\mathbf{c}}_p, g(\mathbf{X}_p;\theta)), \tag{1}$$

$$\widehat{\theta}_p = \underset{\theta}{\operatorname{argmin}} \sum_{q\in\mathcal{N}(p)} \left(\mathbf{c}_q - \boldsymbol{\theta}^\top \phi(\mathbf{x}_q)\right)^2 \omega(\mathbf{x}_p, \mathbf{x}_q), \tag{2}$$

$$\widehat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \ell(\bar{\mathbf{c}}_i, g(\mathbf{X}_i;\theta)), \tag{3}$$

$$\widehat{\mathbf{c}}_p = g_{\text{direct}}(\mathbf{X}_p;\theta) = \mathbf{z}_p^L. \quad \textbf{(1)}$$

$$\widetilde{\mathbf{c}}_{\text{diffuse}} = \mathbf{c}_{\text{diffuse}} \oslash (\mathbf{f}_{\text{albedo}} + \epsilon) \quad \textbf{(4)}$$

$$\widetilde{\mathbf{c}}_{\text{specular}} = \log(1 + \mathbf{c}_{\text{specular}}) \quad \textbf{(5)}$$

$$w_{pq} = \frac{\exp([\mathbf{z}_p^L]_q)}{\sum_{q'\in\mathcal{N}(p)} \exp([\mathbf{z}_p^L]_{q'})}, \quad \textbf{(2)}$$

$$\widehat{\mathbf{c}}_p = g_{\text{weighted}}(\mathbf{X}_p;\theta) = \sum_{q\in\mathcal{N}(p)} \mathbf{c}_q w_{pq}. \quad \textbf{(3)}$$

$$\widehat{\mathbf{c}} = (\mathbf{f}_{\text{albedo}} + \epsilon) \odot \widehat{\mathbf{c}}_{\text{diffuse}} + \exp(\widehat{\mathbf{c}}_{\text{specular}}) - 1, \tag{4}$$

$$(\widetilde{\sigma}_{\text{diffuse}})^2 \approx \sigma^2_{\text{diffuse}} \oslash (\mathbf{f}_{\text{albedo}} + \epsilon)^2,$$
$$(\widetilde{\sigma}_{\text{specular}})^2 \approx \sigma^2_{\text{specular}} \oslash (\widetilde{\mathbf{c}}_{\text{specular}})^2. \tag{5}$$

$$\mathbf{x} = \{\widetilde{\mathbf{c}}, G_x(\{\widetilde{\mathbf{c}}, \mathbf{f}\}), G_y(\{\widetilde{\mathbf{c}}, \mathbf{f}\}), \widetilde{\sigma}^2, {\sigma_{\mathbf{f}}}^2\} \quad \textbf{(6)}$$

每个像素的数据表示为 $\mathbf{x}_p = \{\mathbf{c}_p, \mathbf{f}_p\}$，$\mathbf{x}_p \in \mathbb{R}^{3+D}$，3 维是 RGB，$\mathbf{c}_p$；$D$ 维是 $\mathbf{f}_p$。ground truth 表示为 $\bar{\mathbf{c}}_p$，去噪输出为 $\widehat{\mathbf{c}}_p$。给定去噪函数 $g$，去噪目标就是最小化公式 (1)。去噪结果就是 $\widehat{\mathbf{c}}_p = g(\mathbf{X}_p;\widehat{\theta}_p)$，公式 (1) 中的 $l$ 就是损失函数。

一般来说，先前的去噪方法主要是将 $g(\mathbf{X}_p;\widehat{\theta}_p)$ 替换为 $\theta^T\phi(\mathbf{x}_q)$，函数 $\phi : \mathbb{R}^{3+D} \to \mathbb{R}^M$ 通常是非线性特征转换函数，之后这个问题就可以转换为最小二乘问题，见公式 (2)。

Kalantari[2015] 提出用监督学习和样本来训练，但是它的函数 $g(\mathbf{X}_p;\widehat{\theta}_p)$ 被编写为联合双边或非局部平均滤波器，带宽由多层感知机及其权重 $\theta$ 来提供。因为滤波器是固定的，所以缺少灵活性。

我们希望使用神经网络，这需要解决三个重要问题：

- 函数 $g$ 必须足够灵活来拟合输入数据和 reference 的关系。
- 损失函数 $l$ 很重要，它需要很容易去实现并优化。
- 为了避免过拟合，需要大规模数据集。因为我们需要 reference 渲染很多帧，这是计算昂贵的。数据的泛化性也要足够强。

# 三　深度卷积去噪

图 (2) 表示我们的去噪管线，将 diffuse 组件和 specular 组件区分开分别去噪。

## 3 1　网络架构

网络每一层都有激活函数，使用 ReLU 来激活，

重建有两种方法，一种是直接预测的卷积网络 (direct-prediction convolutional network (DPCN))，另一种是核预测的卷积网络 (kernel-prediction convolutional network (KPCN))。

直接预测就是输入有噪图像，输出去噪的结果。但是由于去噪本身的问题复杂性，使得 CNN 模型收敛很慢。

核预测就是输出一个权重核，这个核作用于 $p$ 的邻域，来得到 $\widehat{\mathbf{c}}_p$。核大小 $k$ 在训练之前与其他网络超参数一起指定（比如网络有多少层、CNN 核的大小），并且把相同的权重赋予每个 RGB 通道。最终输出的核就是一个 $\mathbf{z}_p^L \in \mathbb{R}^{k\times k}$。通过将核归一化（就是用 softmax），然后作用于邻域，就能得到当前值。

这么做有三个主要的好处：

- 它确保最终颜色估计始终位于输入图像的相应邻域的凸包内 (convex hull)。与直接预测方法相比，这大大减少了输出值的搜索空间，并避免了潜在的伪影（例如，颜色偏移 (color shifts)）。
- 它确保相对于内核权重的误差梯度表现良好，这防止了输入的高动态范围导致的网络参数的大振荡变化。直觉上，权重只需要编码邻域的相对重要性；网络不需要学习整个图像规模。一般而言，尺度重新参数化方案 (scale-reparameterization schemes) 最近被证明对于获得低方差梯度和加速收敛至关重要 [Salimans and Kingma 2016]。
- 通过对每个分量应用相同的重建权重，它可以潜在地用于给定帧的各个层的去噪，这是工业界中的常见情况。

### 3 2 Diffuse 和 Specular 组件的分离

分离时，漫反射组件要除以 albedo，见公式红 (4)，这里的 $\oslash$ 表示按元素相除，加个 $\epsilon$ 防止除以 0。由于 irradiance buffer 更平滑（只与光照有关，与复杂的纹理无关），所以可以使用更大的滤波核。镜面组件与高光有关，这里使用 log 函数来降低它的值，使得网络权重更稳定，见公式红 (5)。

最终计算的输出颜色见公式 (4)。

### 3 3 网络结构

用八个隐含层（也就是总共 $L = 9$ 个卷积层）

网络层每层使用 100 个核（具体含义可以见 keras.layers.Conv2D() 函数的第一个参数），每个核是 $5 \times 5$ 大小的（在图 (2) 中有显示）。KPCN 的核尺寸 $k = 21$。

之前有人问过我关于这个网络的输入和输出到底是什么形式，是每次输入图像的一个个小区域 (patch) 还是直接输入整图，以及输出是一个滤波核还是整个图的所有滤波核。

在论文 5.1 节里解释了，图像被分割为 $65\times65$ 大小的 patches，这些 patches 被采样以及打乱顺序，（尽管可以使用均匀采样来从每个帧中选择 patches，但我们发现这样并不够好，因为网络经常有包含平滑区域的简单情况，这些平滑区域很容易去噪。我们希望网络能够接触并学习如何处理更困难的情况。）

根据 Gharbi[2016] 的策略，每帧 $1920 \times 1080$ 大小的图像我们得到 400 个 patches（肯定有 patches 是相互重叠的）。用这么大的区域去预测仅仅中心一个像素的滤波器范围，好像确实有些浪费资源，我去 [1] 这里下载了一下源码来分析（下载真的好慢，github 上的一些人自己写的代码都不是很对，跟原文并不贴合，所以不建议阅读）。放在最后一章节。

## 四 启发性思考

角度是可以的，利用神经网络来拟合一个大滤波核的权重。但是阅读下来不免很多疑问，而文章中又缺少叙述，使得不好分析。

但是论文对于网络的设置等细节讲得不清不楚的，而源码又比较长，阅读起来感觉很不友好，这也难怪为什么有些人自己实现网络，但是实现的结构不是很对。

# 五　网络结构源码分析

## 5 1　数据参数

initialize.py 定义了下面这些量：

```
1  inputDir：输入的训练图像目录
2  outputDir：训练tfrecords的输出目录（即patches的存储目录）
3  gtDir：ground truth图像
4  dataName：tfrecord的名称（'TrainData'）
5  numPatchSamples：每个图像要采样的patch数（为400）
6  shuffleFiles：在循环之前洗牌图像
7  discardBlackPatches：丢弃全黑的patches
8  saveImages：保存debug图像（为0）
9  debugDir：存储debug图像的目录
10 patchSize：patch的尺寸（为65）
11 reconPatchSize：重建核的尺寸（为21）
12 inputChannels：输入patch的通道（为27）
13 outputChannels：输出patch的通道：3
14 featureListFull：全部的特征（对于漫反射featureListDiffuse和镜面组件
       featureListSpecular各自占用其中的一部分），即['default', 'Reference', '
       gradNormal','gradDepth', 'gradAlbedo', 'gradSpecular', 'gradIrrad', '
       diffMap','finalInput', 'finalGt']
15 featureLengthsDiffuse：特征的长度（下面会讲解每个数字的含义，且Specular也是
       同理）[3, FLAGS.outputChannels, 7, 3, 7, 7]
16 finalLayerSize：核预测是21 X 21；直接输出（非核预测，而是去噪像素值预测）就
       是 3
17 layersDiffuse：每个卷积层有多少层卷积核（Specular同理），[FLAGS.
       inputChannels, 100, 100, 100, 100, 100, 100, 100, 100, finalLayerSize]
18 kernels：卷积核大小，为[5,5,5,5,5,5,5,5,5,None]
```

公式红 (6) 中描述了网络的输入，$\tilde{\mathbf{c}}$ 要么是 diffuse 要么是 specular 组件，为 3 通道。$\tilde{\sigma}^2$ 表示根据 luminance 计算出的方差，specular 和 diffuse 是分开的，为 1 通道；$\mathbf{f}$ 是特征，一共有三个特征，包含 3 通道表面法向量，3 通道 albedo 和 1 通道的 depth。$\sigma_{\mathbf{f}}{}^2$ 是每像素特征方差，三通道的特征也是要转换为 luminance，所以 $\sigma_{\mathbf{f}}{}^2$ 是三个特征分别的方差，总共三通道。$G_x$ 和 $G_y$ 表示 $x$ 方向和 $y$ 方向的梯度，每个梯度包含三通道颜色梯度，7 通道特征梯度，总共 10 通道，因此，最终通道数就是：$3+2\times(3+7)+1+3=27$ 通道，即上面的 inputChannels 值。

## 5 2　数据读取

utilities.py 文件的 ConvertExrToTensor() 函数用来读取并且整理数据。
data 字典存储了对应不同名称的数据信息，这跟上一节中的 featureListFull 存储的信息是一样的。
再解释一下 ConvertExrToTensor() 中用到的 CalcGrad() 和 ApplyAlbedo() 函数：

- CalcGrad() 就是求梯度，然后把 $x$ 方向和 $y$ 方向的梯度拼接在一起（因为求梯度，比如 $x$ 方向梯度，需要后一列减前一列，所以得到的梯度值会比原图少一列，所以需要补 0）。
- ApplyAlbedo() 就是给值乘以 albedo 然后加上 $\epsilon$。

GenerateTrainingData() 就是用来调用 ConvertExrToTensor() 并进行采样和生成 $65\times65$ 大小的 patches 的程序。

Crop() 函数用于裁剪，patch 中心就是它的参数 ind，裁剪后大小就是 (2*halfPatch+1)X(2*halfPatch+1)，在本代码里就是 21X21。

实际训练的时候就会调用 InputExr() 函数，该函数下面两行代码负责把输入数据进行 reshape：

```
for key, val in dataList.viewitems():
    dataList[key] = tf.reshape(val, shape=[batchSize, imgHeight, imgWidth,
        dataLengths[key]])
```

AddDataExr() 函数会制作出最终要输入到网络的数据格式：

```
dataList['diffInput'] = tf.reshape(tf.concat(inputListDiff, 2), shape=[
    imgHeight * imgWidth * inputLengthDiff])
dataLengths['diffInput'] = inputLengthDiff
dataList['specInput'] = tf.reshape(tf.concat(inputListSpec, 2), shape=[
    imgHeight * imgWidth * inputLengthSpec])
dataLengths['specInput'] = inputLengthSpec
```

这里的 imgHeight 和 imgWidth 其实就是每个 patch 的长和宽（见 ConvertExrToTensor() 函数）：

```
[imgHeight, imgWidth] = data['default'].shape[:-1]
```

输入到网络里的数据就是 dataList['diffInput'] 和 dataList['specInput']。

### 5 3 大致步骤

训练步骤中，denoiser.py 文件里的 model.EvaluateNetwork() 函数中的 isTraining 参数传入 true，表示是训练过程。该函数构建网络，并且把参数初始化。然后，model.Loss() 函数计算损失函数。根据损失值来传入到 model.Train() 进行训练。

model.EvaluateNetwork() 的返回值就是去噪结果。

```
# Evaluate network
diffOut, specOut, finalOut = model.EvaluateNetwork(data, params, FLAGS.
    trainBatchSize, True, FLAGS)
# Calculate error
diffLoss = model.Loss(diffOut, data['diffGt'], FLAGS.errorFunc, FLAGS)
specLoss = model.Loss(specOut, data['specGt'], FLAGS.errorFunc, FLAGS)
finalLoss = model.Loss(finalOut, data['finalGt'], FLAGS.errorFunc, FLAGS)
```

以 diffuse 组件的去噪为例，在 model.EvaluateNetwork() 函数中，去噪输出是 networkOutDiffuse，它经历了三个过程：

```
# 数据输入到网络
networkOutDiffuse = FeedForward(networkInDiffuse, weightsDiffuse,
    biasesDiffuse, isTraining)
# 网络预测核
......
# 应用核到输出中
networkOutDiffuse = ApplyKernel(networkOutDiffuse, origInputColorDiffuse,
    batchSize, isTraining, FLAGS)
# 裁剪到目标尺寸
networkOutDiffuse = ApplyKernel(networkOutDiffuse, origInputColorDiffuse,
    batchSize, isTraining, FLAGS)
```

注意公式 (4) 中，可以分别训练两个组件，然后将训练结果合并输出：

```
trainOp = [model.Train(diffLoss, globalStep, FLAGS), model.Train(specLoss,
    globalStep, FLAGS)]
```

也可以直接使用公式 (4) 将两个子网络结果合并然后一起训练：

```
trainOp = [model.Train(finalLoss, globalStep, FLAGS)]
```

### 5 4 网络结构

model.py 的 FeedForward() 函数里就是网络的结构：

```
# 计算每个隐含层，用Relu激活
n = len(weights)
for i in xrange(0, n-1):
    conv = tf.nn.conv2d(prevOut, weights[i], strides=[1, 1, 1, 1], padding=
        paddingType)
    prevOut = tf.nn.relu(tf.nn.bias_add(conv, biases[i]))
# 输出层不需要激活函数
conv = tf.nn.conv2d(prevOut, weights[n-1], strides=[1, 1, 1, 1], padding=
    paddingType)
out = tf.nn.bias_add(conv, biases[n-1])
```

weights 表示每个卷积层结构，在 EvaluateNetwork() 中有定义，我们仅看 diffuse 部分：

```
# 初始化网络权重
weightsDiffuse, biasesDiffuse = InitializeWeights(params['layersDiffuse'],
    params['kernels'], 'diff', FLAGS)
# 导入网络并计算
networkOutDiffuse = FeedForward(networkInDiffuse, weightsDiffuse,
    biasesDiffuse, isTraining)
```

InitializeWeights() 函数中初始化 weights 的步骤中（忽略 biases 部分，因为它们差不多），可以看到卷积网络的一层的结构：

```
shape=[kernels[i], kernels[i], layers[i], layers[i+1]]
```

只关注 diffuse 部分，那么 layers 和 kernels 分别是：

```
# inputChannels = 27; finalLayerSize = 21 X 21
layersDiffuse = [FLAGS.inputChannels, 100, 100, 100, 100, 100, 100, 100,
    100, finalLayerSize]
kernels = [5, 5, 5, 5, 5, 5, 5, 5, 5, None]
```

对第一次输入到卷积网络中的数据，tf.nn.conv2d 的第一个参数（也就是上面第一次输入时的 prevOut）是下面的这种格式：

```
# in_channels = 27
[batch, in_height, in_width, in_channels]
```

输入到第一层网络以后就是 100 层的图像了，不考虑 batch，每个卷积核都要卷积 27 层的数据，然后一共 100 个核就能生成输出的 100 层数据。stride 的结构应该是：

```
[batch, in_height, in_width, in_channels]
```

且由于 stride 在每个层上都是 1（当然，batch 和 in_channels 的值必须是 1），所以每层输出的分辨率都是原分辨率，也就是说，网络最终输出的输出尺寸是：

```
[batch, 65, 65, 21 X 21]
```

也就是 patch 里每个像素都相当于有一个滤波核。

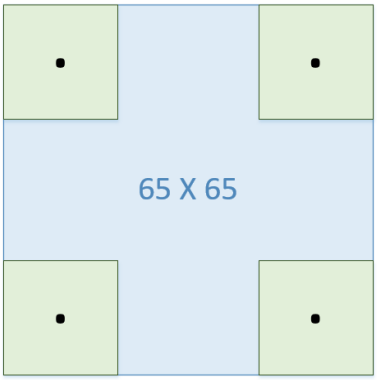
### 5 5 输出

网络输出的结果是滤波核，要对原图像进行滤波，因此会调用 ApplyKernel() 函数，该函数会将网络的输出使用 Softmax 激活，然后 reshape 到和输入同样的形状，然后调用 weighted_average() 来滤波，得到输出的输出。

但是毕竟滤波核有宽度（21×21），所以只能得到下面这个面积区域的去噪结果（根据 weighted_average.py 里的函数能看出）：

```
# k=20, w=65, h=65, bs是batchSize, c是通道数
k = int(sqrt(int(weights[3]))) - 1
[(bs, w-k, h-k, c)]
```

示意图是（绿框表示 $21 \times 21$ 的滤波核）：




也就是中间的 $(65-20) \times (65-20)$ 大小的区域。因此，网络不只是中心一个像素的滤波核权重，而是中间的一整个块内的像素的滤波核权重。

## 参考文献

[1] http://civc.ucsb.edu/graphics/Papers/SIGGRAPH2017_KPCN/