DDMP-去噪扩散概率模型

2024 年 1 月 10 日

https://arxiv.org/abs/2006.11239

1 模型简介

1.1 生成模型

先给生成模型下一个宽泛的定义: 生成模型可以描述成一个生成数据的模型，属于一种概率模型。通过这个模型我们可以生成不包含在训练数据集中的新的数据。比如我们有很多马的图片通过生成模型学习这些马的图像，从中学习到马的样子，生成模型就可以生成看起来很真实的马的图像并却这个图像是不属于训练图像的。

训练集中的每张图片就是一个 (Observation) 观测，每个观测有很多我们需要生成图像所要包含的特征。这些特征往往就是一些像素值。我们的目标就是让这个生成模型生成新的特征集合，这些新生成的特征集合仿佛就是和原数据集的特征集合一样。可想而知这个生成模型完成的工作时相当复杂的，它要从每个像素的庞大的可选空间选出可用的值是多么不容易的事情。

从这个对生成模型的简单介绍我们知道，生成模型一定是一个概率模型而不是判别模型，因为每次生成模型要输出不同的内容。如果说某些特定的图片服从某些概率分布，生成模型就是去尽可能的去模范这个未知概率分布产生新的图像。

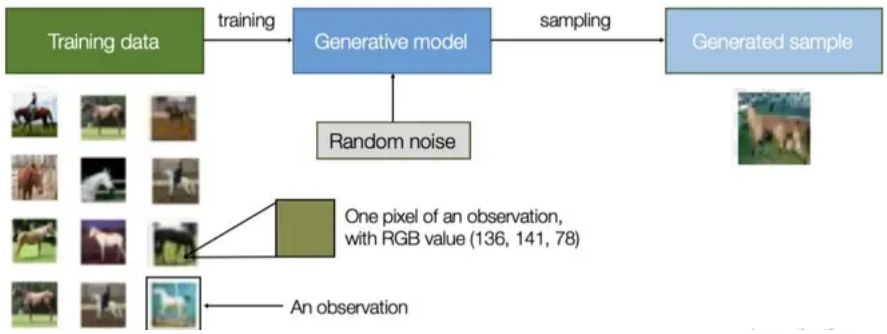

图 1: 生成模型

简化表达就是：生成模型 (generative model) 描述的是这一类的模型：我们接收了从分布 p_{data} 取样的若干样本构成我们的训练集，我们的模型会学习到一个模拟这一分布的概率分布 p_{model}，在有些情况下，我们可以直接的估计概率分布。

1.2 DDMP 模型

与常见的生成模型的机制不同，Denoising Diffusion Probabilistic Model (以下简称 Diffusion Model) 不再是通过一个“限制”（比如种类，风格等等）的输入，逐步添加信息，最终得到生成的图片/ 语音。而是采用从高斯噪音中逐步依照一定条件“采样”特殊的分布，随着“采样”轮次的增加最终得到生成的图片/语音。换句话说，Diffusion Model 的合成过程是通过一次次迭代在噪声中提取出所需要的图像/音频，随着迭代步数的增加，合成质量也在越来越好。

2 模型框架

扩散模型包含两个过程：前向扩散过程和反向生成过程，前向扩散过程是对一张图像逐渐添加高斯噪音直至变成随机噪音，而反向生成过程是去噪音过程，我们将从一个随机噪音开始逐渐去噪音直至生成一张图像，这也是我们要求解或者训练的部分。由下面图像可以更直观地看出：

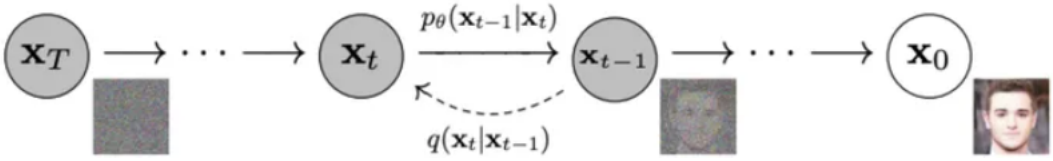

图 2: 主要过程

- 扩散过程（从右往左）：往原始分布 x_0 上不断加入噪声，经过 T 次加噪，最终得到一个各向同性的标准高斯分布 $x_T \sim N(0,1)$，这也是重构过程的基础。
- 重构过程（从左往右）：从一个标准的高斯分布 $x_T \sim N(0,1)$，通过不断的降噪，最终得到一个清晰的图像。

2.1 扩散过程

扩散过程是指的对数据逐渐增加高斯噪音直至数据变成随机噪音的过程。对于原始数据 $x_0 \sim q(x_0)$，总共包含 T 步的扩散过程的每一步都是对上

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$$

图 3: 噪声公式

一步得到的数据: 按如下方式增加高斯噪音：这里 $\{\beta_t\}_{t=1}^T$ 为每一步所采用的方差，它介于 0～1 之间。对于扩散模型，我们往往称不同 step 的方差设定为 variance schedule 或者 noise schedule，通常情况下，越后面的 step 会采用更大的方差，即满足 $\beta_1 < \beta_2 < \ldots < \beta_T$。在一个设计好的 variance schedule 下，的如果扩散步数 T 足够大，那么最终得到的 x_T 就完全丢失了原始数据而变成了一个随机噪音。扩散过程的每一步都生成一个带噪音的数据 x_t，整个扩散过程也就是一个马尔卡夫链： 另外要指出的是，扩散过

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

图 4: 马尔卡夫链

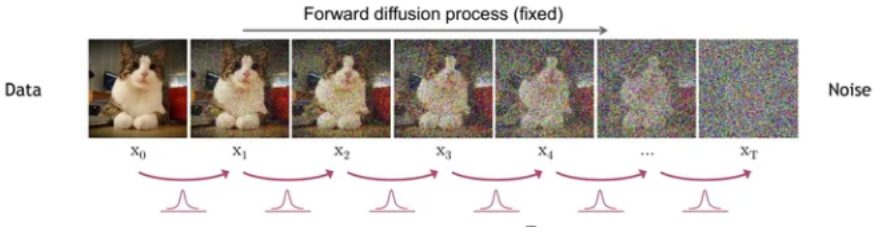

图 5: 扩散过程

程往往是固定的，即采用一个预先定义好的 variance schedule，比如 DDPM 就采用一个线性的 variance schedule。

扩散过程的一个重要特性是我们可以直接基于原始数据 x_0 来对任意 t 步的 x_t 进行采样：$x_t \sim q(x_t|x_0)$。

这里定义 $\alpha_t = 1 - \beta_t$ 和 $\overline{\alpha} = \prod_{i=1}^{t} \alpha_i$，通过重参数技巧（和 VAE 类似），那么有：上述推到过程利用了两个方差不同的高斯分布 $N(0, \sigma_1^2 I)$ 和 $N(0, \sigma_2^2 I)$ 相加等于一个新的高斯分布 $N(0, (\sigma_2^2 + \sigma_1^2)I)$。反重参数化后，我们得到：$q(x_t|x_0) = N(x_t; \sqrt{\overline{\alpha_t}}x_0, (1 - \overline{\alpha_t})I)$

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad \Rightarrow \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

图 6: 扩散过程公式

$$\begin{aligned}
\mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} \\
&= \sqrt{\alpha_t}(\sqrt{\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_{t-1}}\epsilon_{t-2}) + \sqrt{1-\alpha_t}\epsilon_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\sqrt{\alpha_t - \alpha_t\alpha_{t-1}}^2 + \sqrt{1-\alpha_t}^2}\,\bar{\epsilon}_{t-2} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\bar{\epsilon}_{t-2} \\
&= \dots \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon
\end{aligned}$$

图 7: 迭代

扩散过程的这个特性很重要。首先，我们可以看到 x_t 其实可以看成是原始数据 x_0 和随机噪音 ϵ 的线性组合，其中 $\sqrt{\bar{\alpha}_t}$ 和 $\sqrt{1-\bar{\alpha}_t}$ 为组合系数，它们的平方和等于 1，我们也可以称两者分别为 signal_rate 和 noise_rate。

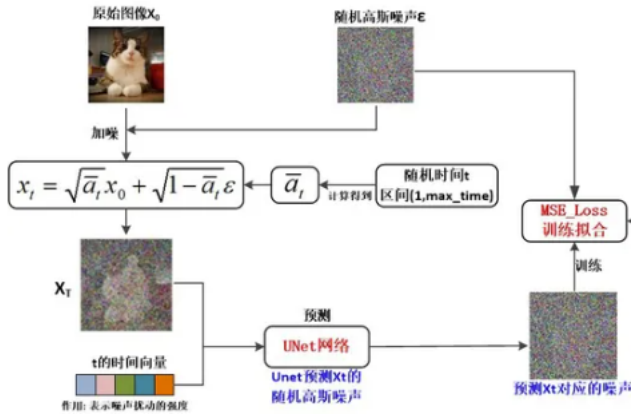

图 8: 扩散总流程

2.2 重构过程

扩散过程是将数据噪音化，那么反向过程就是一个去噪的过程，如果我们知道反向过程的每一步的真实分布 $q(x_{t-1}|x_t)$，那么从一个随机噪音 $x_T \sim N(0, I)$ 开始，逐渐去噪就能生成一个真实的样本，所以反向过程也就是生成数据的过程。

估计分布 $q(x_{t-1}|x_t)$ 需要用到整个训练样本，我们可以用神经网络来估计这些分布。这里，我们将反向过程也定义为一个马尔卡夫链，只不过它是由一系列用神经网络参数化的高斯分布来组成：这里 $p(x_T) = N(x_T; o, I)$，而

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

图 9: 高斯

$p_\theta(x_{t-1}|x_t)$ 为参数化的高斯分布，它们的均值和方差由训练的网络 $\mu_\theta(x_t, t)$ 和 $\sum_\theta(x_t, t)$ 给出。实际上，扩散模型就是要得到这些训练好的网络，因为它们构成了最终的生成模型。

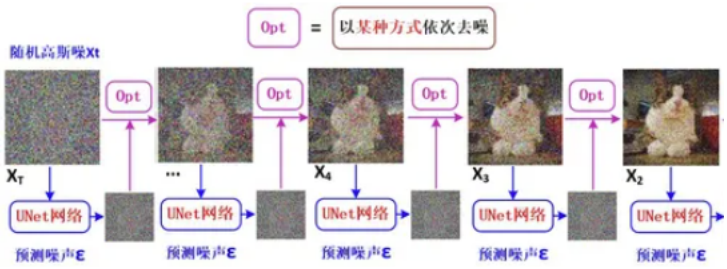

图 10: 重构过程

推理时，先生成一个随机噪声 Xt，其实对应训练阶段 Xt 输入 UNet 网络后的输出结果，然后分 T 次，一步一步地以某种方式去除噪声，即 Xt -> Xt-1 -> Xt-2 -> ... -> X0,X0 就是最终要得到的 RGB 图像。

- 先生成随机高斯正态分布的噪声 Xt, 对应的是第 t 步
- Xt 先经过 UNet 网络预测其对应的噪声 (Xt,t)，相当于训练时，Xt-1 到 Xt 过程中，需要添加的噪声 (Xt-1,t)

- Xt 以某种方式去除噪声 (Xt,t)
- 经过 t 步去噪后，得到 X0

3 UNet

3.1 模型介绍

语义分割 (Semantic Segmentation) 是图像处理和机器视觉一个重要分支。与分类任务不同，语义分割需要判断图像每个像素点的类别，进行精确分割。语义分割目前在自动驾驶、自动抠图、医疗影像等领域有着比较广泛的应用。U-Net 是比较早的使用全卷积网络进行语义分割的算法之一，论文中使用包含压缩路径和扩展路径的对称 U 形结构在当时非常具有创新性，且一定程度上影响了后面若干个分割网络的设计，该网络的名字也是取自其 U 形形状。

3.2 模型内容

3.2.1 网络结构

直入主题，U-Net 的 U 形结构如下图所示。网络是一个经典的全卷积网络（即网络中没有全连接操作）。网络的输入是一张 $572 * 572$ 的边缘经过镜像操作的图片（input image tile），网络的左侧（红色虚线）是由卷积和 Max Pooling 构成的一系列降采样操作，论文中将这一部分叫做压缩路径（contracting path）。压缩路径由 4 个 block 组成，每个 block 使用了 3 个有效卷积和 1 个 Max Pooling 降采样，每次降采样之后 Feature Map 的个数乘 2，因此有了图中所示的 Feature Map 尺寸变化。最终得到了尺寸为 $32 * 32$ 的 Feature Map。

网络的右侧部分 (绿色虚线) 在论文中叫做扩展路径（expansive path）。同样由 4 个 block 组成，每个 block 开始之前通过反卷积将 Feature Map 的尺寸乘 2，同时将其个数减半（最后一层略有不同），然后和左侧对称的压缩路径的 Feature Map 合并，由于左侧压缩路径和右侧扩展路径的 Feature Map 的尺寸不一样，U-Net 是通过将压缩路径的 Feature Map 裁剪到和扩展路径相同尺寸的 Feature Map 进行归一化的（即上图中左侧虚线部分）。扩展路径的卷积操作依旧使用的是有效卷积操作，最终得到的 Feature Map

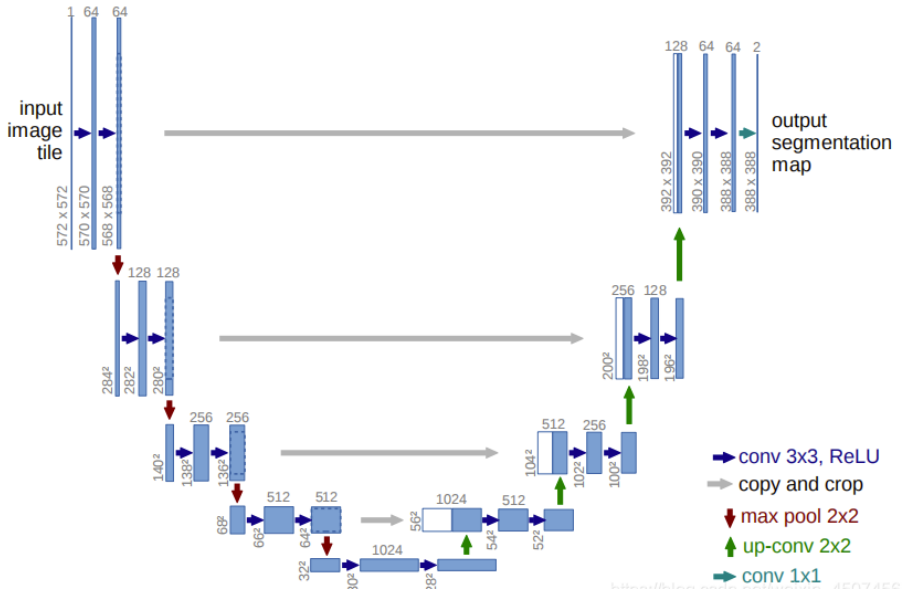

图 11: 网络结构

的尺寸是 $388 * 388$ 。由于该任务是一个二分类任务，所以网络有两个输出 Feature Map。

3.2.2 镜像操作

数据集我们的原始图像的尺寸都是 $512 * 512$ 的。为了能更好的处理图像的边界像素，U-Net 使用了镜像操作（Overlay-tile Strategy）来解决该问题。镜像操作即是给输入图像加入一个对称的边，那么边的宽度是多少呢？一个比较好的策略是通过感受野确定。因为有效卷积是会降低 Feature Map 分辨率的，但是我们希望 $512 * 512$ 的图像的边界点能够保留到最后一层 Feature Map。所以我们需要通过加边的操作增加图像的分辨率，增加的尺寸即是感受野的大小，也就是说每条边界增加感受野的一半作为镜像边。

Unet 的好处我感觉是：网络层越深得到的特征图，有着更大的视野域，浅层卷积关注纹理特征，深层网络关注本质的那种特征，所以深层浅层特征都是有格子的意义的；另外一点是通过反卷积得到的更大的尺寸的特征图的边缘，是缺少信息的，毕竟每一次下采样提炼特征的同时，也必然会损失一些边缘特征，而失去的特征并不能从上采样中找回，因此通过特征的拼接，

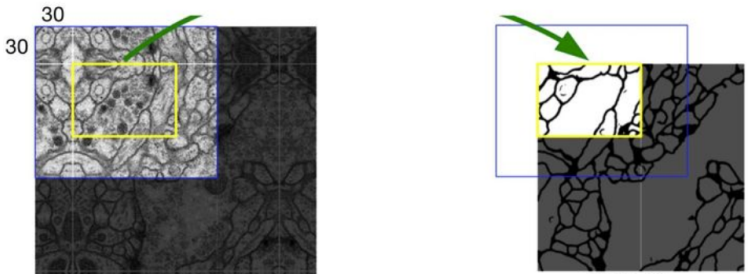

图 12: 镜像操作

来实现边缘特征的一个找回。

4 代码部分

4.1 ddpm

这部分主要是对图片输入的格式，模型网络卷积大小等进行了设置，可以根据电脑的 gpu 大小选择图片大小和 batch_size 的大小，进而改变模型的训练速度：

```
_defaults = {
    #-----------------------------------------------#
    #   model_path指向logs文件夹下的权值文件
    #-----------------------------------------------#
    # "model_path"        : 'model_data/Diffusion_Flower.pth',
    "model_path"        : 'logs\diffusion_model_last_epoch_weights.pth',
    #-----------------------------------------------#
    #   卷积通道数的设置
    #-----------------------------------------------#
    # "channel"           : 128,
    "channel"           : 32,
    #-----------------------------------------------#
    #   输入图像大小的设置
    #-----------------------------------------------#
    "input_shape"       : (32, 32),
    #-----------------------------------------------#
    #   betas相关参数
    #-----------------------------------------------#
    "schedule"          : "linear",
    "num_timesteps"     : 1000,
    "schedule_low"      : 1e-4,
    "schedule_high"     : 0.02,
    #-------------------------------#
    #   是否使用Cuda
    #   没有GPU可以设置成False
    #-------------------------------#
    "cuda"              : True,
}
```

图 13: 模型设置

4.2 unet

这部分是对 UNet 模型的一个设置，涵盖了全连接层，模型卷积，通道数目设定和特征提取等多个方面。

```
class UNet(nn.Module):
    def __init__(
        self, img_channels, base_channels=128, channel_mults=(1, 2, 4, 8),
        num_res_blocks=3, time_emb_dim=128 * 4, time_emb_scale=1.0, num_classes=None, activation=SiLU(),
        dropout=0.1, attention_resolutions=(1,), norm="gn", num_groups=32, initial_pad=0,
    ):
        super().__init__()
        # 使用到的激活函数，一般为SILU
        self.activation = activation
        # 是否对输入进行padding
        self.initial_pad = initial_pad
        # 需要去区分的类别数
        self.num_classes = num_classes

        # 对时间轴输入的全连接层
        self.time_mlp = nn.Sequential(
            PositionalEmbedding(base_channels, time_emb_scale),
            nn.Linear(base_channels, time_emb_dim),
            SiLU(),
            nn.Linear(time_emb_dim, time_emb_dim),
        ) if time_emb_dim is not None else None

        # 对输入图片的第一个卷积
        self.init_conv  = nn.Conv2d(img_channels, base_channels, 3, padding=1)

        # self.downs用于存储下采样用到的层，首先利用ResidualBlock提取特征
        # 然后利用Downsample缩低特征图的高宽
```

图 14: UNet 设置

4.3 diffusion

这部分对高斯扩散进行了定义，涵盖了噪声生成和噪声移除，并且具有优化

```
class GaussianDiffusion(nn.Module):
    def __init__(
        self, model, img_size, img_channels, num_classes=None, betas=[], loss_type="l2", ema_decay=0.9999
    ):
        super().__init__()
        self.model      = model
        self.ema_model  = deepcopy(model)

        self.ema                = EMA(ema_decay)
        self.ema_decay          = ema_decay
        self.ema_start          = ema_start
        self.ema_update_rate    = ema_update_rate
        self.step               = 0

        self.img_size       = img_size
        self.img_channels   = img_channels
        self.num_classes    = num_classes

        # L1或者L2损失
        if loss_type not in ["l1", "l2"]:
            raise ValueError("__init__() got unknown loss type")

        self.loss_type      = loss_type
        self.num_timesteps  = len(betas)

        alphas              = 1.0 - betas
```

图 15: gaussdiffusion

5 成果展示

图 16: 预测结果