Denoising Diffusion Probabilistic Models(DDPM)

论文地址

代码地址

基本公式

- 连续性随机变量函数的期望

$$E(f(x)) = \int_x p(x)f(x)dx \tag{1}$$

- `KL` 散度公式

$$D_{KL}(p(x)||q(x)) = E(log(\frac{p(x)}{q(x)}) = \int_x p(x)log\frac{p(x)}{q(x)}dx \tag{2}$$

- 参数重整化

若希望从高斯分布 $\mathcal{N}(\mu, \sigma^2)$中采样，可以先从标准分布 $\mathcal{N}(0, \mathbf{I})$采样出 z，再得到 $\sigma * z + \mu$，这就是我们想要采样的结果。这样做的好处是将随机性转移到了z这个常量上，而σ和μ则当作仿射变换网络的一部分。

- 先验概率和后验概率
  - 先验概率：根据以往经验和分析得到的概率,如全概率公式,它往往作为“由因求果”问题中的“因”出现，如$q(x_t|x_{t-1})$
  - 后验概率：指在得到“结果”的信息后重新修正的概率,是“执果寻因”问题中的“因", 如$p(x_{t-1}|x_t)$

什么是扩散模型

相较于 GANs, VAEs 与 Normalizing Flows 等生成式模型，扩散模型（Diffusion Model）的思路非常简单：向数据中逐步地加入随机噪声，然后学习其逆向过程，希望最终能够从噪声中重建所需的数据样本 —— 有意思的地方在于，我们可以训练一个神经网络模型来学习逆向扩散过程。

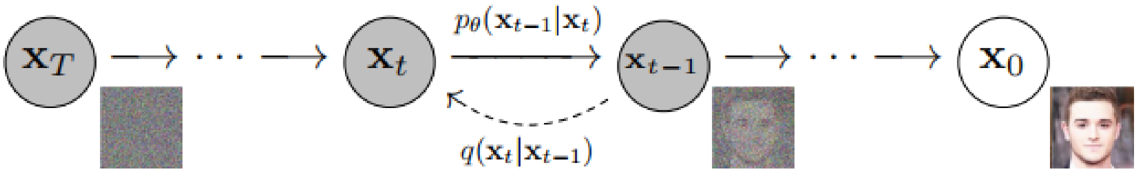


Figure 2: The directed graphical model considered in this work.

Fig 1. DDPM前向逆向过程

在逆向扩散的每一步，神经网络模型的输入是当前的样本x_t和加噪程度`t`，想一想预期输出可以是什么呢？有多种可能：比如采取最粗暴的一种做法，即一步到位，要求模型直接对x_0进行预测，发现实验效果不佳；想一下扩散的样子... 逐步地，渐渐地变化，因此扩散模型要求对去噪的下一个状态进行预测，即对x_{t-1}进行预测。而在实际实验的过程中，作者发现：先让模型对去噪过程中的噪声ϵ进行预测，再通过相关计算得到x_{t-1}会有更好的效果。

前向扩散过程

给定在真实数据分布（训练数据集）上的数据点（一张图片），$x_0 \sim q(x_0)$，前向过程就是有限`T`步内不断给数据点加入随机噪声（高斯噪声），这个加噪过程形成中间序列$x_1, x_2, \cdots, x_T$。`T`越大，效果越好，但是会增加训练难度以及采样（去噪）的时间（扩散模型的缺点就是从随机噪声去噪的时间太长），因此选择一个合适的`T`满足要求（要求β很小）即可。在每次加噪程度β是超参数，参数`T`以及β序列生成算法决定了β，如果没有特殊说明，本篇文章的`T`为1000。β在训练之前就已经确定，这个超参满足：

$$0 < \beta_1 < \beta_2 < \cdots < \beta_T < 1$$

β_i与T有以下关系：

```
# 一种是线性变化，一种是余弦变化，可以根据需要选择
def linear_beta_schedule(timesteps):
    scale = 1000 / timesteps
    beta_start = scale * 0.0001
    beta_end = scale * 0.02
    return torch.linspace(beta_start, beta_end, timesteps, dtype =
torch.float64)

def cosine_beta_schedule(timesteps, s = 0.008):
    """
    cosine schedule
    as proposed in https://openreview.net/forum?id=-NEXDKk8gZ
    """
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps, dtype = torch.float64)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * math.pi * 0.5)
** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return torch.clip(betas, 0, 0.999)
```

由β可以写出条件概率公式：

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t \mathbf{I}) \tag{3}$$

条件概率公式参数重整化

对$\mathcal{N}(\mu, \sigma^2)$采样操作是不可导的，所以无法使用梯度下降，因此可以先从一个高斯分布$\mathcal{N}(0, \mathbf{I})$上进行采样，而$\mu$和$\sigma$在放射变化中可导，这样就相当于在采样操作上可以进行反向传播，这个过程实际上就是参数重整化。$q(x_t)|x_{t-1}$的采样过程等价于：

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}z_{t-1} \tag{4}$$

其中z_{t-1}是独立随机变量z_{t-1}~$\mathcal{N}(0, \mathbf{I})$，保证采样的随机性。上面这个式子不存在可训练参数，所以`DDPM`的前向过程是不需要进行训练，通过递推或者迭代便可以通过x_0以及β得出x_t。在这里设$\alpha_t = 1 - \beta_t$，x_t关于x_0的显式表达式的推理过程如下：

$$\begin{aligned}
x_t &= \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}z_{t-1} \\
&= \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}z_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}z_{t-2} \\
&= \cdots \\
&= \sqrt{\tilde{\alpha}_t}x_0 + \sqrt{1-\tilde{\alpha}_t}z
\end{aligned} \tag{5}$$

最终得出$q(x_t|x_0)=\mathcal{N}(x_t,\sqrt{\tilde{\alpha}_t}x_0,1-\tilde{\alpha}_t\mathbf{I})$，$\tilde{\alpha}_t=\prod_{i=1}^{T}\alpha_i$，这么做的好处是可以加速$x_t$的生成速度，当然根据公式(3)，可以从$x_0->x_1->x_2->\cdots->x_t$，计算难度与计算时间都会变的更大。因此在接下来的代码注释中，直接讲解公式(5)的生成方式。

代码注释

- 首先通过`beta_schedule`得到beta序列，这里的`timesteps=T=1000`
- 然后通过α_i以及$\tilde{\alpha}_i$与`\beta_i`之间的关系，得出相应序列。`alphas = 1. - betas`好理解，torch.cumprod是将该之前的项累乘，`alphas_cumprod`的每一项对应$\tilde{\alpha}_i$，`alphas_cumprod_prev`是取$\tilde{\alpha}$下标`0~T-2`项作为这个序列的`1-T-1`，第一项为1，具体形式为`1,alphas_cumprod[0],alphas_cumprod[1],...,alphas_cumprod[T-2]`。
- `sqrt_alphas_cumprod`~$\sqrt{\tilde{\alpha}}$，`sqrt_one_minus_alphas_cumprod`~$\sqrt{1-\tilde{\alpha}}$，`log_one_minus_alphas_cumprod`~$log(1-\tilde{\alpha})$，`sqrt_recip_alphas_cumprod` ~ $\sqrt{\frac{1}{\tilde{\alpha}}}$，`sqrt_recipm1_alphas_cumprod` ~ $\sqrt{\frac{1}{\tilde{\alpha}}-1}$，

```
if beta_schedule == 'linear':
    betas = linear_beta_schedule(timesteps)
elif beta_schedule == 'cosine':
    betas = cosine_beta_schedule(timesteps)
else:
    raise ValueError(f'unknown beta schedule {beta_schedule}')
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value = 1.)
register_buffer('sqrt_alphas_cumprod', torch.sqrt(alphas_cumprod))
register_buffer('sqrt_one_minus_alphas_cumprod', torch.sqrt(1. -
alphas_cumprod))
register_buffer('log_one_minus_alphas_cumprod', torch.log(1. - alphas_cumprod))
register_buffer('sqrt_recip_alphas_cumprod', torch.sqrt(1. / alphas_cumprod))
register_buffer('sqrt_recipm1_alphas_cumprod', torch.sqrt(1. / alphas_cumprod -
1))
```

下面是从x_0生成的x_t的代码

- 首先是`extract`，这个函数十分重要，在代码中出现的频率非常高，其作用是选取特定下标`t`的信息并转换成特定维度。因为`DDPM`无论是前向还是逆向都会遵循一个长度为1000的时间序列，使用这个函数可以较为轻松该时间点对应的权重或者序列。
- `q_sample`的实际作用就是返回$q(x_t|x_0)$的采样结果，`extract(self.sqrt_alphas_cumprod, t, x_start.shape)`返回的是$\sqrt{\tilde{\alpha}_t}$，`extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape)`返回的是$\sqrt{1-\tilde{\alpha}_t}$，注意下标`t`和参数列表中的`t`是相同的。
- `noise`就是在标准正太分布上采样生成的与`x_start`的维度形状完全相同的`tensor`，`return`的结果就是$q(x_t|x_0)$

```python
def extract(a, t, x_shape):
    b, *_ = t.shape
    out = a.gather(-1, t)
    return out.reshape(b, *((1,) * (len(x_shape) - 1)))       #为了提取第t列的索引
def q_sample(self, x_start, t, noise=None):    # q(x_t|x_0) 公式（4）
        noise = default(noise, lambda: torch.randn_like(x_start))
        return (
            extract(self.sqrt_alphas_cumprod, t, x_start.shape) * x_start +
            extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape) *
noise
        )
```

扩散过程的联合分布公式

根据马尔可夫链的无记忆性（下一个状态的概率分布只与当前状态有关），可以得到扩散过程联合概率分布：

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \tag{6}$$

随着 `t` 不断增大 `t->T`，x_0逐渐地丢失掉原有的特征，最终得到的x_T将会是一个标准正态分布$\mathcal{N}(0, \mathbf{I})$。

总结

前向扩散过程完全不含有可训练的参数，**也即是说扩散模型的前向扩散过程不需要对网络进行训练。** 只要给定x_0和β, 通过不断迭代，就可以算出对应时刻 `t` 的$q(x_t)$分布。实际上在逆扩散的过程中网络也没有进行训练，我们做的仅仅是使用训练好的网络对噪声进行 “采样”。这也是扩散模型的特点之一：你可以将不同的扩散策略与不同的预测模型进行组合使用，只需提供一致的维度与超参数等信息（比如 `T` 值）。

逆向扩散过程

我们已经知道了$q(x_t|x_{t-1})$以及$q(x_t|x_0)$，前向的过程的性质基本上已经搞清楚了，那么下面就是这篇论文最为重要的过程——**逆向扩散过程**。逆向过程是什么？实际上就是 `fig1` 从左到右的过程，但是上文的推导是从右到左的过程，那么如何得到从左到右的过程即$q(x_{t-1}|q(x_t))$?

从理论上，只需要采样出一些随机的高斯分布噪声$x_T \sim \mathcal{N}(0, \mathbf{I})$，逐步去噪，就能最终得到真实分布$q(x_0)$的采样，但是这需要$q(x_{t-1}|x_t)$知道整个数据分布，这是不太现实的。我们所能做的就是尽可能近似这个后验概率分布，所以我们使用神经网络来实现这个过程。论文中把去噪模型称作$p_\theta(x_{t-1}|x_t)$，其中p_θ是神经网络中的参数。

在上面的论述中，我们让 `T` 很大，并保证β很小，所以我们可以假设这个逆向过程也是一个高斯分布（强假设，这个点我不太清楚，貌似是β很小，然后保证独立同分布？）。$p_\theta(x_{t-1}|x_t)$有如下的形式:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}, \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{7}$$

其中μ_θ为参数定义的均值，Σ_θ是参数定义的方差。

均值和方差也是基于噪声等级 `t` 来决定的。因此我们的神经网络需要根据条件输入x_t和 `t` 来学习（表示）对应的均值和方差，但在 DDPM 的原始论文中，作者选择了固定方差Σ_θ为只与 `t` 有关的常数，只要求神经网络去学习均值。理由是经过试验，固定与不固定方差最终的预测效果差不多。在后续的研究论文中尝试对此进行改进，其中神经网络除了学习均值之外，还学习了逆向过程的方差。在这里我们选择和原始 DDPM 论文做法一致，仅要求模型学得后验概率分布的均值，固定方差。

公式（7）的继续推导

由贝叶斯公式：

$$
\begin{aligned}
q(x_{t-1}|x_t,x_0) &= \frac{q(x_{t-1},x_t,x_0)}{q(x_t,x_0)} \\
&= \frac{q(x_t|x_{t-1},x_0)q(x_{t-1}|x_0)q(x_0)}{q(x_t,x_0)} \\
&= q(x_t|x_{t-1},x_0)\frac{q(x_{t-1}|x_0)}{q(x_t,x_0)/q(x_0)} \\
&= q(x_t|x_{t-1},x_0)\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)}
\end{aligned} \tag{8}
$$

$q(x_t|x_{t-1},x_0)$根据马尔可夫性质可以将不造成影响的x_0去掉，变为
$q(x_t|x_{t-1})=\mathcal{N}(x_t;\sqrt{1-\beta_t}x_{t-1},\beta_t\mathbf{I})$。

对于$x=\mathcal{N}(\mu,\sigma)$，可以表示为：

$$
f(x)=\frac{1}{\sqrt{2\pi\sigma}}exp(-\frac{(x-\mu)^2}{2\sigma^2})\propto exp(-\frac{1}{2}(\frac{(x-\mu)^2}{\sigma^2})) \tag{9}
$$

按照公式(9)，$q(x_t|x_{t-1},x_0)\propto exp(-\frac{1}{2}(\frac{(x_t-\sqrt{\alpha_t}x_{t-1})^2}{\beta_t^2}))$，$q(x_{t-1}|x_0)\propto exp(-\frac{1}{2}(\frac{(x_{t-1}-\sqrt{\tilde{\alpha}_{t-1}}x_0)^2}{1-\tilde{\alpha}_{t-1}}))$以及$q(x_t|x_0)\propto exp(-\frac{1}{2}(\frac{(x_t-\sqrt{\tilde{\alpha}_t}x_0)^2}{1-\tilde{\alpha}_t}))$。因此有以下推导：

$$
\begin{aligned}
q(x_{t-1}|x_t,x_0) &\propto exp(-\frac{1}{2}(\frac{(x_t-\sqrt{a_t}x_{t-1})^2}{\beta_t}+\frac{(x_{t-1}-\sqrt{\tilde{\alpha}_{t-1}}x_0)^2}{1-\tilde{\alpha}_{t-1}}-\frac{(x_t-\sqrt{\tilde{\alpha}_t}x_0)}{1-\tilde{\alpha}_t}) \\
&= exp(-\frac{1}{2}(\frac{x_t^2-2\sqrt{a_t}x_tx_{t-1}+a_tx_{t-1}^2}{\beta_t}+\frac{x_{t-1}^2-2\sqrt{\tilde{\alpha}}_{t-1}x_0x_{t-1}+\tilde{\alpha}_{t-1}x_0^2}{1-\tilde{\alpha}_{t-1}}-\cdots)) \\
&= exp(-\frac{1}{2}((\frac{\alpha_t}{\beta_t}+\frac{1}{1-\tilde{\alpha}_{t-1}})x_{t-1}^2)-(\frac{2\sqrt{\alpha_t}}{\beta_t}x_t+\frac{2\sqrt{\tilde{\alpha}_{t-1}}}{1-\tilde{\alpha}_{t-1}}x_0)x_{t-1}+C(x_t,x_0))
\end{aligned}
$$

对于概率密度函数为$f(x)=ax^2+bx+c=a(x+\frac{b}{2a})+c$的标准高斯分布有$\mu=-\frac{b}{2a}$和$\Sigma=\frac{1}{a}$。与上面推导出的最后的式子进行对照，我们可以得到$a=\frac{\alpha_t}{\beta_t}+\frac{1}{1-\tilde{\alpha}_{t-1}}$和$b=\frac{2\sqrt{\alpha_t}}{\beta_t}x_t+\frac{2\sqrt{\tilde{\alpha}_{t-1}}}{1-\tilde{\alpha}_{t-1}}x_0$

。我们进行整理，分别求出μ以及Σ。

Σ求解以及代码注释

$$
\begin{aligned}
\tilde{\beta}_t &= 1/(\frac{\alpha_t}{\beta_t}+\frac{1}{1-\tilde{\alpha}_{t-1}}) \\
&= 1/\frac{\alpha_t-\alpha_t\tilde{\alpha}_{t-1}+\beta_t}{(1-\tilde{\alpha}_{t-1})\beta_t} \\
&= \frac{1-\tilde{\alpha}_{t-1}}{1-\tilde{\alpha}_t}*\beta_t
\end{aligned} \tag{10}
$$

注意

$\tilde{\beta}_t$仅仅与`t`有关，当`t`确定的时候$\tilde{\beta}_t$便可以被计算出来，因此代码在一开始时，便已经求出来对应序列。

代码

这里就不做详细解释了，`posterior_variance`（也就是$\tilde{\beta}_t$）的右式的含义已经在上文全部给出。

```
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
register_buffer('posterior_variance', posterior_variance)
```

μ求解

$$
\begin{aligned}
\tilde{\mu}_t(x_t, x_0) &= (\frac{\sqrt{\alpha_t}}{\beta_t}x_t + \frac{\sqrt{\tilde{\alpha}_{t-1}}}{1-\tilde{\alpha}_{t-1}}x_0)/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\tilde{\alpha}_{t-1}}) \\
&= (\frac{\sqrt{\alpha_t}}{\beta_t}x_t + \frac{\sqrt{\tilde{\alpha}_{t-1}}}{1-\tilde{\alpha}_{t-1}}x_0) * \frac{1-\tilde{\alpha}_{t-1}}{1-\tilde{\alpha}_t}\beta_t \\
&= \frac{\sqrt{\alpha_t}(1-\tilde{\alpha}_{t-1})}{1-\tilde{\alpha}_t}x_t + \frac{\sqrt{\tilde{\alpha}_{t-1}}\beta_t}{1-\tilde{\alpha}_t}x_0
\end{aligned} \tag{11}
$$

由式(5)x_0对x_t的表示，反过来可以得到x_t对x_0的表示：

$$
x_0 = \frac{1}{\sqrt{\tilde{\alpha}_t}}(x_t - \sqrt{1-\tilde{\alpha}_t}z_t) \tag{12}
$$

将式(12)代入式(11)进行化简：

$$
\begin{aligned}
\tilde{\mu}_t(x_t, t) &= \frac{\sqrt{\alpha_t}(1-\tilde{\alpha}_{t-1})}{1-\tilde{\alpha}_t}x_t + \frac{\sqrt{\tilde{\alpha}_{t-1}}\beta_t}{1-\tilde{\alpha}_t}\frac{1}{\sqrt{\tilde{\alpha}_t}}(x_t - \sqrt{\tilde{\alpha}_t}z_t) \\
&= (\frac{\sqrt{\alpha_t}(1-\tilde{\alpha}_{t-1})}{1-\tilde{\alpha}_t} + \frac{\sqrt{\tilde{\alpha}_{t-1}}\beta_t}{1-\tilde{\alpha}_t}\frac{1}{\sqrt{\tilde{\alpha}_t}})x_t - \frac{\sqrt{\tilde{\alpha}_{t-1}}\beta_t}{(1-\tilde{\alpha}_t)\sqrt{\tilde{\alpha}_t}}z_t \\
&= (\frac{\sqrt{\alpha_t} - \frac{\tilde{\alpha}_t}{\sqrt{\alpha}_t} + \frac{1}{\sqrt{\alpha_t}}(1-\alpha_t)}{1-\tilde{\alpha}_t})x_t - \frac{\sqrt{\tilde{\alpha}_{t-1}}\beta_t}{\sqrt{1-\tilde{\alpha}_t}\sqrt{\tilde{\alpha}_t}}z_t \\
&= (\frac{\frac{1}{\sqrt{\alpha_t}}(1-\tilde{\alpha}_t) + \sqrt{\alpha_t} - \sqrt{\alpha_t}}{1-\tilde{\alpha}_t})x_t - \frac{\beta_t}{\sqrt{1-\tilde{\alpha}_t}\sqrt{\alpha_t}}z_t \\
&= \frac{1}{\sqrt{\alpha}_t}x_t - \frac{1}{\sqrt{\alpha_t}}\frac{\beta_t}{\sqrt{1-\tilde{\alpha}_t}}z_t \\
&= \frac{1}{\sqrt{\alpha}_t}(x_t - \frac{\beta_t}{\sqrt{1-\tilde{\alpha}_t}}z_t)
\end{aligned} \tag{13}
$$

总结

从Σ求解中，我们可以知道，$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}, \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$的在`t`可知时方差是固定的，为一个常数，所以在训练神经网络时自然不把方差作为优化项。在计算损失或者在训练网络时均以μ为主。

目标分布的似然函数（损失Loss）

我们需要用的式子：

$$
q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \tag{6}
$$

$$
p_\theta(x_{0:T}) = p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \tag{14}
$$

我们的优化目标是使得经由p_θ逆扩散过程得到的数据分布尽可能与$q(x_0)$一致，可考虑最小化其负对数似然$-\log(p_\theta(x_0))$， 但它无法被直接计算。

可以借鉴 VAE 模型中用到的变分下界（VLB）处理思路，在负对数似然$-\log(p_\theta(x_0))$的基础上加上一个相关的 KL 散度（KL散度始终大于等于0），能够构成负对数似然的上界，上界越小，负对数似然也就越小，等同于目标分布的似然最大。我们希望使用p_θ去近似q_θ，因此有：

$$\begin{aligned}
-\log p_\theta(x_0) &\le -\log p_\theta(x_0) + D_{KL}(q(x_{1:T}|x_0)||p_\theta(x_{1:T}|x_0)) \\
&= -\log p_\theta(x_0) + E_{x_{1:T}\sim q(x_{1:T}|x_0)}[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})/p_\theta(x_0)}] \\
&= -\log p_\theta(x_0) + E_q[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})} + \log p_\theta(x_0)] \\
&= E_q[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}]
\end{aligned}$$

上式两边乘上$E_{q(x_0)}$，可以得到交叉熵形式上界，即：

$$L_{VLB} = E_{q(x_{0:T})}[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}] \ge -E_{q(x_0)} \log p_\theta(x_0)$$

我们希望最小化交叉熵，可以通过最小化交叉熵的上界，即最小化目标函数L_{VLB}来实现，它是可优化的。实际上经过一系列推导，最后得到的需要被优化的目标函数十分简洁。

目标函数重写

为了能够实际地进行计算，需要对上面的L_{VLB}进行重写，希望最终变成已知公式的组合：

$$\begin{aligned}
L_{VLB} &= E_q[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}] \\
&= E_q[\log \frac{\prod_{t=1}^T q(x_t|x_{t-1})}{p(x_T)\prod_{t=1}^T p_\theta(x_{t-1}|x_t)}] \\
&= E_q[-\log p_\theta(x_T) + \Sigma_{t=1}^T \log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)}] \\
&= E_q[-\log p_\theta(x_T) + \Sigma_{t=2}^T \log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)}]
\end{aligned} \tag{15}$$

上面式子的推导可由式子(6)、(14)以及较简单的变化得到。我们接下来看下面的推导，和之前的后验概率的计算类似。

$$\begin{aligned}
q(x_t|x_{t-1}) = q(x_t|x_{t-1}, x_0) &= \frac{q(x_{t-1}, x_t, x_0)}{q(x_{t-1}, x_0)} \\
&= \frac{q(x_{t-1}|x_t, x_0)q(x_t|x_0)q(x_0)}{q(x_{t-1}, x_0)} \\
&= q(x_{t-1}|x_t, x_0)\frac{q(x_t|x_0)}{q(x_{t-1}, x_0)/q(x_0)} \\
&= q(x_{t-1}|x_t, x_0)\frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}
\end{aligned} \tag{16}$$

将式(16)代入式(15)，

$$
\begin{aligned}
L_{VLB} &= E_q[-\log p_\theta(x_T) + \Sigma_{t=2}^T \log \frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)} \frac{q(x_t|x_0)}{q(x_{t-1}|x_0)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)}] \\
&= E_q[-\log p_\theta(x_T) + \Sigma_{t=2}^T log(\frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}) + \Sigma_{t=2}^T log(\frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}) + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)}] \\
&= E_q[-\log p_\theta(x_T) + \Sigma_{t=2}^T log(\frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}) + \log q(x_T|x_0) - \log p_\theta(x_0|x_1)] \\
&= E_q[\log \frac{q(x_T|x_0)}{p_\theta(x_T)} + \Sigma_{t=2}^T log(\frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}) - \log p_\theta(x_0|x_1)] \\
&= E_q[D_{KL}(q(x_T|x_0)||p_\theta(x_T)) + \Sigma_{t=2}^T (D_{KL}(q(x_{t-1}|x_t,x_0)||p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1)] \qquad (17)
\end{aligned}
$$

上面的式子中Eq对于KL散度实际上多余的，当然加上也没错。论文中的证明过于简单，下面是一些补充证明：

$$
\begin{aligned}
& \mathbb{E}_{q(x_{0:T})}\left[\log \frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}\right] \\
=\ & \mathbb{E}_{\color{red}{q(x_{t-1}|x_t,x_0)}q(x_t,x_0)q(x_{1:t-2,t+1:T}|x_{t-1},x_t,x_0)}\left[\log \frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}\right] \\
=\ & \mathbb{E}_{\bar{q}(x_{t-1})}\left[\mathbb{E}_{q(x_{t-1}|x_t,x_0)}\left[\log \frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}\right]\right] \\
=\ & \mathbb{E}_{\bar{q}(x_{t-1})}\left[D_{\mathrm{KL}}(q(x_{t-1}|x_t,x_0)||p_\theta(x_{t-1}|x_t))\right] \\
=\ & D_{\mathrm{KL}}(q(x_{t-1}|x_t,x_0)||p_\theta(x_{t-1}|x_t))
\end{aligned}
$$

为了更方便的描述这几项，论文中有以下的分类：

$$
\mathbb{E}_q\left[\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p(\mathbf{x}_T))}_{L_T} + \sum_{t>1}\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}}\underbrace{-\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0}\right]
$$

损失函数化简

> 第一项为L_T，中间的求和是L_1~L_{T-1}，最后一项设为L_0

$$
E_q[D_{KL}(q(x_T|x_0)||p_\theta(x_T) + \Sigma_{t=2}^T D_{KL}(q(x_{t-1}|x_t,x_0)|p_\theta(x_{t-1}|x_t)) - logp_\theta(x_0|x_1)]
$$

(1)L_0和中间的项合并，因为将中间项取$t=1$，那么

$$
D_{KL}(q(x_0|x_1,x_0)|p_\theta(x_0|x_1)) = D_{KL}(q(1)|p_\theta(x_0|x_1)) = -logp_\theta(x_0|x_1)
$$

(2)第一项损失没有可优化项，为常数。

$$
E_q[D_{KL}(q(x_T|x_0)||p_\theta(x_T)]\text{的第一项不含}\theta\text{，第二项就是一个高斯分布，也是不含}\theta
$$

(3)最终整理出来的式子为

$$
E_q[\Sigma_{t=1}^T D_{KL}(q(x_{t-1}|x_t,x_0)|p_\theta(x_{t-1}|x_t))) = L_0 + \cdots + L_{T-1}
$$

对于两个单一变量的高斯分布 `p` 和 `q` 而言，它们之间的 `KL` 散度为

$$
KL(p,q) = log\frac{\sigma 1}{\sigma 2} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}
$$

[KL(p,q)公式推导]

前面我们已经知道，不管是$q(x_{t-1}|x_t, x_0)$还是$p_\theta(x_{t-1}|x_t)$，它们的方差都只和`t`有关，所以如果单独分析某一项即固定`t`，两者的方差都是常数。通过上面式子$KL(p, q)$，σ_1与σ_2都是可知的，因此L_t(`0<=t<=T-1`)的只与μ_1和μ_2有关，因此我们在训练时，只需要关注μ_1和μ_2即可，其余部分都是权重或者参数，所以得到如下的式子（常数忽略）：

$$L_t = E_q[\frac{1}{2||\Sigma_\theta(x_t, t)||_2^2}||\mu_t(\tilde{x_t}, x_0) - \mu_\theta(x_t, t)||^2] \tag{18}$$

之前的求解中$\mu_t(\tilde{x}_t, x_0)$具体形式已经求出即式(13)，代入式(18)：

$$L_t = E_q[\frac{1}{2||\Sigma_\theta(x_t, t)||_2^2}||\mu_\theta(x_t, t) - \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\tilde{\alpha}_t}}z_t)||^2] \tag{19}$$

当神经网络在模拟逆扩散过程中，$\mu_t(\tilde{x}_t, x_0)$来预测$\tilde{\mu}_t$，而在模型训练时我们将x_t作为训练时的输入，所以x_t是不可优化的，因此我们可以将高斯噪声项进行参数化，从而在时刻`t`从`x_t`来预测`z_t`。如果我们尝试让模型$z_\theta(x_t, t)$去学习预测噪声z_t，则可以得到从模型预测的噪声计算出$p_\theta(x_{t-1}|x_t)$均值的公式：

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\tilde{\alpha}_t}}z_\theta(x_t, t)) \tag{20}$$

值得注意的一点是，论文官方的实际代码实现中并没有直接使用化简后得到的公式(13)从预测的噪声计算上一时刻的状态，而是选择了先让模型输出$z_\theta(x_t, t)$作为z_t; 接着计算预测的原始数据$x_0 = \frac{1}{\sqrt{\tilde{\alpha}_t}}(x_t - \sqrt{1-\tilde{\alpha}_t}z_t)$, 再根据$\mu_\theta(x_t, t)$计算得到 x_{t-1}的均值，实际上两者的计算结果是理论等价的。对此笔者的解释是，作者在进行对比实验时，分别比对了让预测输出（模型均值）为x_{t-1}, ϵ和直接输出 x_0三种设置下的效果，为了复用从 x_0计算得到 x_{t-1}的代码（以及x_0可能在其它实验中需要被用到），故选择了从$z_\theta(x_t, t)$到x_0再到x_{t-1}的计算路线，而没有套用如上论文中所给出算法的$z_\theta(x_t, t)$直接到x_{t-1}的计算公式。

将式(20)代入式(19)，那么L_t化简之后的结果为:

$$L_t = E_q[\frac{\beta_t^2}{2\alpha_t(1-\tilde{\alpha}_t)2||\Sigma_\theta(x_t, t)||_2^2}||z_t - z_\theta(x_t, t)||^2] \tag{21}$$

在DDPM的原始论文中忽略掉目标函数中的权重项效果会更好，同时将重整形式的$x_t = \sqrt{\tilde{\alpha}_t}x_0 + \sqrt{1-\tilde{\alpha}_t}z$代入上式中，得到:

$$L_t^{simple} = E_{x_0, z_t, t}[||z_t - z_\theta(\sqrt{\tilde{\alpha}_t}x_0 + \sqrt{1-\tilde{\alpha}_t}z_t, t)||^2] \tag{22}$$

最终得到化简后的目标函数:

$$L_{simple} = L_t^{simple} + C \tag{23}$$

总结

（1） 首先从式(23)出发，为什么最终的目标函数是一项而不是$L_0 + \cdots + L_{t-1}$的和?

以我训练`DDPM`为例，我选择的`batch_size=32`，图片为三通道，大小`128*128`的人脸图片，将`32*3*128*128`的张量送入`DDPM`进行训练时:

```
 b, c, h, w, device, img_size, = *img.shape, img.device, self.image_size
assert h == img_size and w == img_size, f'height and width of image must be
{img_size}'
t = torch.randint(0, self.num_timesteps, (b,), device=device).long()
img = normalize_to_neg_one_to_one(img)
return self.p_losses(img, t, *args, **kwargs)
```

首先会生成一个`batch_size`大小的从`[0,1000)`随机生成的时间序列，注意这里的序列长度并不是1000，也就是说每次`iteration`中的每张图片的损失实际上只有`[0,1000)`中的一份，而不是将全部的1000项损失求和然后反向传播，这样既提高了训练速度，也增加了随机性，只要训练的足够多，就可以认为训练效果是均匀且充分的。

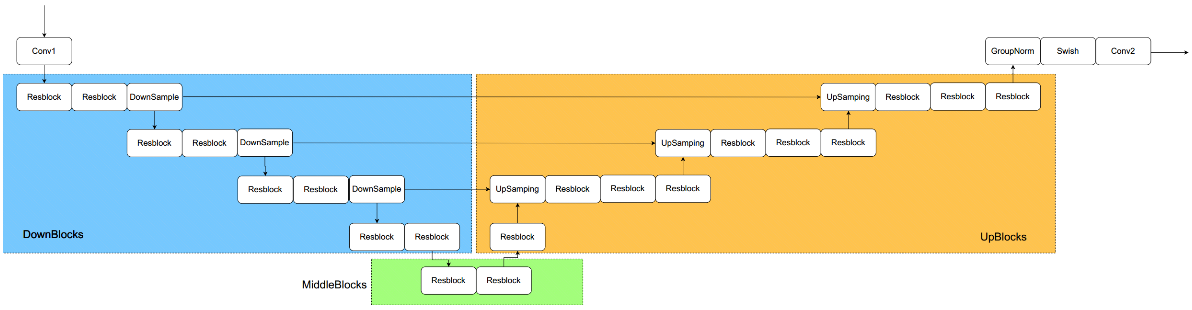
（2）这一部分对代码的讲述较少，主要原因是篇幅过长，而且如果分散将的话效果可能不佳。

（3）这部分看似很复杂，其中主要就是在求$q(x_{t-1}|x_t,x_0)$，当然我们无法准确的得到这个式子的方程式，所以要使用神经网络来逼近这个后验分布。模型如何训练呢，如何计算损失，论文借鉴了`VAE`损失以及它的处理思路，从这里可以看出`DDPM`与`VAE`关系很大，其中应用了非常多的技巧，简化到最后实际上就是两个分布之间的损失，但是无法直接计算，通过采样来得到它们之间的差距。

DDPM的结构以及训练过程

结构

DDPM的以Unet为主体，加入了自注意力机制。它的输入是一个分布（经过采样后）以及时间序列t，输出对应时刻的一个相同维度和形状的噪音。



Unet结构

> 为什么使用U-net

从输入和输出的 shape 来看，二者具有与最终所需重建的图片一致的维度和形状，比较经典的网络结构是 UNet. 为了保证 High-level 的信息能够在 Low-level 也能用到，原始 UNet 使用了所谓的 skip-connection 操作，即在要进行下采样的时候会将当前的数据存一份，与后续上采样到相同 Level 的数据做 concat 操作，在 Channel 维度上拼接在一起。对于同一个图像大小 Level, 原始 UNet 中的 skip-connection 操作只做一次，而且会对不匹配的宽高比进行处理；DDPM 论文中则是将每一次经过 ResBlock 前的数据也给存了下来，因此同一个 Level 会有多次 concat 操作。

> 为什么使用自注意力机制?

时间序列有1000个时刻，当输入一个100时刻的噪声，可以输出对应时刻的噪声，当输入200时刻的噪声，模型会输出对应的噪声吗？所以要加一个参数`t`，告诉模型这个噪声是`t`时刻的，需要输出对应`t`时刻的预测噪声。如果训练1000个模型当然是可以，可是却会消耗大量的显存，使用自注意力机制将时刻`t`作为一个权重嵌入到对应位置，相当于字典`key-value`的关系，查询`t`时刻，得到`t`时刻的预测值。

Unet相关的代码太长，这里就不详细讲解了。

训练过程

代码地址的主页给定了对应的训练代码，很容易看懂，所以这部分主要讲解`GaussianDiffusion`类，初始化代码已经讲了很多，这一部分省略。这个类没有需要训练的层，大部分都是对数据的处理，然后送入Unet，得到预测值，计算损失，反向传播。

```
def q_sample(self, x_start, t, noise=None):    # q(x_t|x_0) 公式（4）
```

```python
        noise = default(noise, lambda: torch.randn_like(x_start))

        return (
            extract(self.sqrt_alphas_cumprod, t, x_start.shape) * x_start +
            extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape) * noise
        )
    def p_losses(self, x_start, t, noise = None):
        b, c, h, w = x_start.shape
        noise = default(noise, lambda: torch.randn_like(x_start))
        # print("noise:{}".format(noise.shape))
        x = self.q_sample(x_start = x_start, t = t, noise = noise)
        # print("t is dim:{}".format(t.shape))
        model_out = self.model(x, t)

        if self.objective == 'pred_noise':
            target = noise
        elif self.objective == 'pred_x0':
            target = x_start
        else:
            raise ValueError(f'unknown objective {self.objective}')

        loss = self.loss_fn(model_out, target, reduction = 'none')
        loss = reduce(loss, 'b ... -> b (...)', 'mean')      # 在CHW上合并,然后求均值
        # print(loss.shape)
        loss = loss * extract(self.p2_loss_weight, t, loss.shape)    # 乘上对应的权重
        return loss.mean()
    def forward(self, img, *args, **kwargs):
        b, c, h, w, device, img_size, = *img.shape, img.device, self.image_size
        assert h == img_size and w == img_size, f'height and width of image must be {img_size}'
        t = torch.randint(0, self.num_timesteps, (b,), device=device).long()
        img = normalize_to_neg_one_to_one(img)
        return self.p_losses(img, t, *args, **kwargs)
```

（1）`forward`过程首先是随机生成长度为`batch_size`的时间序列，并将输入的图片的值由`[0,255]`转化到`[-1,1]`

（2）然后就是`p_losses`，现在随机噪音上采样出和输入图片的维度形状相同完全相同的噪声

（3）通过`q_sample`得到x_t，然后与时间序列`t`一起送入Unet（`self.model`），得到预测噪音$z_\theta(x_t,t)$

（4）计算`l1`损失，论文中的是`l2`，但是`l2`损失太小了，梯度不是很明显，最后的训练效果不太好，所以这份代码中使用的`l1`，当然也可以给`l2`加上一个权重。得到损失后反向传播。

（5）具体从高斯分布进行去噪的过程（逆向过程这里就不讲解了），可以参考：[扩散模型（一）：DDPM 基本原理与 MegEngine 实现]的代码实现逐步去噪部分。

参考资料

- 扩散模型（一）：DDPM 基本原理与 MegEngine 实现
- DDPM解读（一）| 数学基础，扩散与逆扩散过程和训练推理方法
- Diffusion Models for Deep Generative Learning
- DDPM代码详细解读(1)：数据集准备、超参数设置、loss设计、关键参数计算

- [Diffusion Model扩散模型理论与完整PyTorch代码详细解读]
- [什么是 Diffusion Models/扩散模型？]
- [生成扩散模型漫谈（一）：DDPM = 拆楼 + 建楼]