

LAB4: Deep Generative Model: VAE

21304219 刘文婧

研究问题的背景和动机

Deep Generative Model

传统生成模型（如高斯混合模型、隐马尔科夫模型）在复杂数据上表现有限。而深度生成模型（如VAE、GAN、Diffusion模型）利用深度神经网络的强大表示能力，能在高维空间中进行有效建模，性能比传统生成模型更好。

生成模型应用广泛，在内容创作、图像修复方面都能得到应用。

解决该问题的主要方法

VAE

变分自编码器

原理见下面详细的模型阐述。

VAE的训练过程较稳定，但相比下面2种生成模型，生成的样本质量较低，生成的图像会比较模糊。

自回归模型

自回归模型将高维数据的联合分布分解为一系列条件概率的乘积。每一步生成的数据点是基于前面生成的数据点（或历史信息）生成的。

对目标数据 $x = (x_1, x_2, \dots, x_N)$ ，它的联合分布可以分解为：

$$p(x) = p(x_1)p(x_2 | x_1)p(x_3 | x_1, x_2)\dots p(x_N | x_1, x_2, \dots, x_{N-1})$$

GAN

生成对抗网络 Generative adversarial network

GAN由两个网络组成：

- **生成器 (Generator)**：输入一个随机噪声 z ，生成伪造样本 $G(z)$ 。
- **判别器 (Discriminator)**：输入真实样本 x 和生成样本 $G(z)$ ，判别输入是“真实”还是“伪造”。

生成器和判别器之间形成一个“对抗”关系：

- **生成器**尝试“欺骗”判别器，使其认为生成的样本是“真实的”。
- **判别器**尝试提高自身的判断能力，区分真实和伪造样本。

损失函数是一个对抗性优化目标 ($\min \max$)，通常采用交叉熵形式：

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

GAN的好处：

- 能生成非常逼真的高分辨率样本。
- 适合无监督学习，无需明确的样本标签。

缺点

- 训练不稳定：可能出现模式崩塌（生成样本缺乏多样性）。
- 优化难度大：生成器和判别器之间的对抗性优化可能导致不收敛。

Diffusion

扩散模型

扩散模型是一种基于概率扩散过程的生成模型，包括两个阶段：

- **正向扩散过程：**

- 将原始数据 x_0 添加逐步噪声，生成一系列更噪的样本 x_1, x_2, \dots, x_T ，最终接近标准高斯分布。
- 每一步扩散过程满足特定的马尔科夫链：

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

- **反向生成过程：**

- 学习逆向去噪过程，从噪声样本逐步还原到原始数据分布。
- 学习一个参数化的模型 $p_\theta(x_{t-1} | x_t)$ ，优化目标为极大似然估计。
- 扩散模型通常通过一个神经网络（如 U-Net）实现去噪步骤。

优点

- 能够生成高质量的样本，尤其在生成图像细节方面非常出色。
- 模型训练稳定，无模式崩塌问题。

缺点

- 采样效率低：生成样本需要多步去噪，耗时较长。
- 训练时间较长：需要大量的计算资源。

VAE 模型阐述

从 VB-EM 到 VAE

VB-EM

- 回顾EM算法，我们的目标是：找

$$p(\mathbf{x}; \boldsymbol{\theta}) \quad \text{s.t.} \quad \log p(\mathbf{x}; \boldsymbol{\theta}) \text{ larger}$$

如果 $p(\mathbf{x}; \boldsymbol{\theta})$ 效果不错，那么也能用来生成新的数据点 \mathbf{x}

- EM算法采取的措施是：

$$\begin{aligned} \text{E-step : } Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) &= \frac{1}{N} \sum_{n=1}^N \mathbb{E}_{p(\mathbf{z}^{(n)} | \mathbf{x}^{(n)}, \boldsymbol{\theta}^{(t)})} [\log p(\mathbf{x}^{(n)}, \mathbf{z}^{(n)}; \boldsymbol{\theta})] \\ \text{M-step : } \boldsymbol{\theta}^{(t+1)} &= \arg \max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) \end{aligned}$$

- 得到后验分布 $p(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta}^{(t)})$ 未必容易，不妨直接用一个分布（比如取高斯分布） $q(\mathbf{z}; \boldsymbol{\phi})$ 来近似，那么在 EM 算法的步骤就变成：

- 首先最小化近似分布和实际后验分布之间的 \mathcal{KL} 散度：

$$\begin{aligned} \boldsymbol{\phi}^{(t)} &= \arg \min_{\boldsymbol{\phi}} \text{KL} (q(\mathbf{z}; \boldsymbol{\phi}) \| p(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta}^{(t)})) \\ &= \arg \min_{\boldsymbol{\phi}} \int_{\mathbf{z}} q(\mathbf{z}; \boldsymbol{\phi}) \log \frac{q(\mathbf{z}; \boldsymbol{\phi})}{p(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta}^{(t)})} d\mathbf{z} \end{aligned}$$

- 然后就是 EM 算法的步骤：

$$\begin{aligned} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) &= \mathbb{E}_{q(\mathbf{z}; \boldsymbol{\phi}^{(t)})} [\log p(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta})] \\ \boldsymbol{\theta}^{(t+1)} &= \arg \max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) \end{aligned}$$

- 根据之前 EM 算法的推导，我们已经知道上述步骤2的目的是：

$$\boldsymbol{\theta}^{(t+1)} = \arg \max_{\boldsymbol{\theta}} \int_{\mathbf{z}} q(\mathbf{z}; \boldsymbol{\phi}^{(t)}) \log \frac{p(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta})}{q(\mathbf{z}; \boldsymbol{\phi}^{(t)})} d\mathbf{z}$$

- 而步骤1其实是在：

$$\begin{aligned} \text{Step 1: } \max_{\boldsymbol{\phi}} \int_{\mathbf{z}} q(\mathbf{z}; \boldsymbol{\phi}) \log \frac{p(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta}^{(t)})}{q(\mathbf{z}; \boldsymbol{\phi})} d\mathbf{z} \\ \iff \max_{\boldsymbol{\phi}} \int_{\mathbf{z}} q(\mathbf{z}; \boldsymbol{\phi}) \log \frac{p(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta}^{(t)}) p(\mathbf{x}; \boldsymbol{\theta}^{(t)})}{q(\mathbf{z}; \boldsymbol{\phi})} d\mathbf{z} \\ \iff \max_{\boldsymbol{\phi}} \int_{\mathbf{z}} q(\mathbf{z}; \boldsymbol{\phi}) \log \frac{p(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta}^{(t)})}{q(\mathbf{z}; \boldsymbol{\phi})} d\mathbf{z} \end{aligned}$$

- 可见二者目标一致，也就是说，可以同时地使用梯度下降法优化 $\boldsymbol{\phi}, \boldsymbol{\theta}$ 这2个参数：

$$\phi_t, \theta_t = \arg \max_{\phi, \theta} \mathcal{L}(\mathbf{x}; \theta, \phi) = \int_z q(z; \phi) \log \frac{p(\mathbf{x}, z; \theta)}{q(z; \phi)} dz.$$

$$\Rightarrow \begin{cases} \phi^{(t+1)} = \phi^{(t)} + \gamma \frac{\partial \mathcal{L}(\mathbf{x}; \theta, \phi)}{\partial \phi} \Big|_{\phi=\phi^{(t)}} \\ \theta^{(t+1)} = \theta^{(t)} + \gamma \frac{\partial \mathcal{L}(\mathbf{x}; \theta, \phi)}{\partial \theta} \Big|_{\theta=\theta^{(t)}} \end{cases} \quad \text{Simultaneously}$$

VAE

1. 回到目标:

$$p(\mathbf{x}; \theta) \quad \text{s.t.} \quad \log p(\mathbf{x}; \theta) \text{ larger}$$

$$p(\mathbf{x}) = \int_z p(\mathbf{x}, z) dz, \quad p(\mathbf{x}, z) = p(\mathbf{x}|z)p(z)$$

2. 隐变量仍然用较简单的生成方式:

$$\mathbf{z} \sim \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$$

3. 考虑对后验分布 $p_\theta(\mathbf{x}|z)$, 引入神经网络 μ_θ 来训练得到高斯分布的均值:

$$p_\theta(\mathbf{x}|z) = \mathcal{N}(\mathbf{x}; \mu_\theta(z), \mathbf{I}),$$

$$q_\phi(z|x) = \mathcal{N}(z; \lambda, \text{diag}(\eta)), \phi = \{\lambda, \eta\}$$

那么生成 \mathbf{x} 的方式就是:

$$\epsilon \sim \mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I}), \quad \mathbf{x} = \mu_\theta(z) + \epsilon.$$

1. VB-EM 的方法来训练目标概率函数, 即用梯度下降法训练如下目标:

$$\begin{aligned} \arg \max_{\theta, \phi} \mathcal{L}(\mathbf{x}; \theta, \phi) &= \arg \max_{\theta, \phi} \int_z q_\phi(z|x) \log \frac{p_\theta(\mathbf{x}, z)}{q_\phi(z|x)} dz \\ &= \arg \max_{\theta, \phi} \int_z q_\phi(z|x) \log p_\theta(\mathbf{x}|z) dz - \int_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z)} dz \\ &= \arg \max_{\theta, \phi} \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(\mathbf{x}|z)] - \text{KL}(q_\phi(z|x) \parallel p(z)) \end{aligned}$$

2. 对 KL 散度的处理, 是有解析解的:

$$\text{KL}(q_\phi(z|x_n) \parallel p(z)) = \int q_\phi(z|x_n) \log q_\phi(z|x_n) dz - \int q_\phi(z|x_n) \log p(z) dz$$

代入如下分布: (对后验分布 q_ϕ 也引入神经网络 ϕ_1, ϕ_2 训练均值和方差)

$$q_\phi(z|x) = \mathcal{N}(z; g_{\phi_1}(x), \text{diag}(g_{\phi_2}^2(x)))$$

$$p(z) = \mathcal{N}(z; \mathbf{0}, \mathbf{I})$$

得到

$$q_{\phi}(\mathbf{z}|\mathbf{x}_n) = \frac{1}{(2\pi)^{\frac{N}{2}} |g_{\phi_2}^2 \mathbf{I}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{z} - g_{\phi_1})^\top (g_{\phi_2}^2 \mathbf{I})^{-1} (\mathbf{z} - g_{\phi_1})\right)$$

$$\log q_{\phi}(\mathbf{z}|\mathbf{x}_n) = -\frac{N}{2} \log g_{\phi_2}(\mathbf{x}_n) - \frac{1}{2} \left\| \frac{\mathbf{z} - g_{\phi_1}(\mathbf{x}_n)}{g_{\phi_2}(\mathbf{x}_n)} \right\|_2^2 + c$$

那么，散度的第1项：(不妨先假设维度 $N = 1$ 来推导)

$$\begin{aligned} \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \log q_{\phi}(\mathbf{z}|\mathbf{x}_n) d\mathbf{z} &= -\frac{1}{2} \log g_{\phi_2}(\mathbf{x}_n) \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) d\mathbf{z} + C \\ &\quad - \frac{1}{2} \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \left\| \frac{\mathbf{z} - g_{\phi_1}(\mathbf{x}_n)}{g_{\phi_2}(\mathbf{x}_n)} \right\|_2^2 d\mathbf{z} \\ &= -\frac{1}{2} \log g_{\phi_2}(\mathbf{x}_n) + C \\ &\quad - \frac{1}{2} \left\| g_{\phi_2}(\mathbf{x}_n) \right\|_2^2 \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \|\mathbf{z} - g_{\phi_1}(\mathbf{x}_n)\|^2 d\mathbf{z} \\ &= -\frac{1}{2} \log g_{\phi_2}(\mathbf{x}_n) + C - \frac{1}{2} \left\| g_{\phi_2} \right\|_2^2 \mathbb{E}[\|\mathbf{z} - g_{\phi_1}(\mathbf{x}_n)\|^2] \\ &= -\frac{1}{2} \log g_{\phi_2}(\mathbf{x}_n) + C - \frac{1}{2} \left\| g_{\phi_2} \right\|_2^2 \\ &= -\frac{1}{2} \log g_{\phi_2}(\mathbf{x}_n) + C' \end{aligned}$$

散度的第2项：

$$\begin{aligned} \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \log p(\mathbf{z}) d\mathbf{z} &= \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \left(-\frac{1}{2} \|\mathbf{z}\|_2^2 + C \right) d\mathbf{z} \\ &= -\frac{1}{2} \mathbb{E}[\|\mathbf{z}\|_2^2] + C \\ &= -\frac{1}{2} \left(g_{\phi_1}^2 + g_{\phi_2}^2 \right) + C \end{aligned}$$

于是，

$$\arg \max_{\theta, \phi} (-\text{KL}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})))$$

的目标，变成：

$$\arg \max_{\phi_1, \phi_2} \left\{ \log [g_{\phi_2}] - g_{\phi_1}^2 - g_{\phi_2}^2 \right\}$$

如果是独立的 N 维变量，在每个维度上都像上面这么做就好。

梯度下降法训练。

3. 对

$$\arg \max_{\theta, \phi} \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z})]$$

没有解析解。

用类似MCMC-EM的想法，每次迭代中，生成 K 个：

$$\mathbf{z}_k \sim q_\phi(\mathbf{z}|\mathbf{x})$$

于是

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z})] \approx \frac{1}{K} \sum_{k=1}^K \log p_\theta(\mathbf{x}|z_k)$$

然后直接梯度下降法训练，但发现如果直接对上式求导，由于上式的 z_k 直接由采样得到，已经没有了 $\phi = \{\boldsymbol{\lambda}, \boldsymbol{\eta}\}$ 的信息，所以也没办法对这2个参数求导然后梯度下降。

重参数化

于是就保留信息，和前面采样的方式类似，每次采样这样表示：

$$z_k = \lambda_n + \eta_n \epsilon^k, \quad \epsilon^k \sim \mathcal{N}(0, \mathbf{I})$$

不同的样本当然会带来不同的后验分布，于是 λ_n, η_n 是对不同的样本 x_n 不同的。

但如果要保留 $\phi = \{\boldsymbol{\lambda}_1, \boldsymbol{\eta}_1, \boldsymbol{\lambda}_2, \boldsymbol{\eta}_2, \dots, \boldsymbol{\lambda}_N, \boldsymbol{\eta}_N\}$ 这么多参数，每次更新的过程就要太多步了，不妨直接用2个神经网络，对不同的样本 x_n 能输出不同的 $\boldsymbol{\lambda}_n = g_{\phi_1}(\mathbf{x}_n), \boldsymbol{\eta}_n = g_{\phi_2}(\mathbf{x}_n)$

于是

$$\mathbf{z}_k = g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n) \epsilon^k, \quad \epsilon^k \sim \mathcal{N}(0, \mathbf{I})$$

即

$$q_\phi(\mathbf{z}|\mathbf{x}_n) = \mathcal{N}(\mathbf{z}; g_{\phi_1}(\mathbf{x}_n), \text{diag}(g_{\phi_2}^2(\mathbf{x}_n)))$$

代入

$$p_\theta(\mathbf{x}|\mathbf{z}_k) = \frac{1}{(2\pi)^{D/2} |\mathbf{I}|^{1/2}} \exp \left(-\frac{1}{2} (\mathbf{x}_n - \mu_\theta(\mathbf{z}_k))^\top \mathbf{I}^{-1} (\mathbf{x}_n - \mu_\theta(\mathbf{z}_k)) \right)$$

得到

$$\log p_\theta(\mathbf{x}|\mathbf{z}_k) = C - \frac{1}{2} \|\mathbf{x}_n - \mu_\theta(g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n) \epsilon^k)\|_2^2$$

再代入

$$\begin{aligned}\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})] &\approx \frac{1}{K} \sum_{k=1}^K \log p_{\theta}(\mathbf{x}|z_k) \\ &= \frac{1}{K} \sum_{k=1}^K \left(C - \frac{1}{2} \|\mathbf{x}_n - \mu_{\theta}(g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n)\epsilon^k)\|_2^2 \right)\end{aligned}$$

总结VAE的训练目标与过程

根据上面的推导，现在我们只需要最大化下面这个下界函数就好：

$$\begin{aligned}\mathcal{L} &\approx \frac{1}{NK} \sum_{n=1}^N \sum_{k=1}^K \left(C - \frac{1}{2} \left\| \mathbf{x}_n - \mu_{\theta} \left(g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n) \cdot \epsilon_n^{(k)} \right) \right\|_2^2 \right) \\ &\quad - \frac{1}{N} \sum_{n=1}^N \text{KL} (q_{\phi}(\mathbf{z}_n|\mathbf{x}_n) \| p(\mathbf{z})) \\ &= \frac{1}{NK} \sum_{n=1}^N \sum_{k=1}^K \left(C - \frac{1}{2} \left\| \mathbf{x}_n - \mu_{\theta} \left(g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n) \cdot \epsilon_n^{(k)} \right) \right\|_2^2 \right) \\ &\quad + \frac{1}{N} \sum_{n=1}^N \sum_{j=1}^J (\log [g_{\phi_2,j}] - g_{\phi_1,j}^2 - g_{\phi_2,j}^2)\end{aligned}$$

其中 J 指 隐变量 \mathbf{z} 所在的空间的维度数。

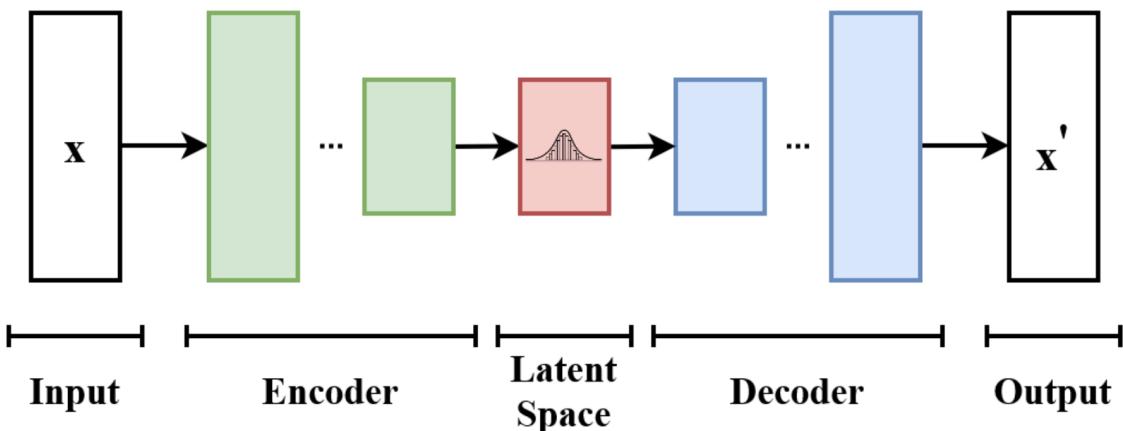
用梯度下降法训练。

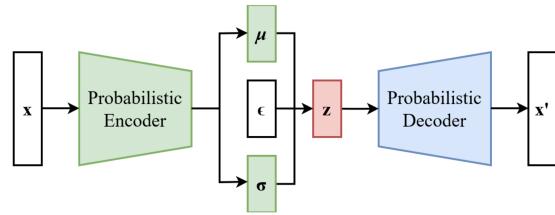
整个过程相当于是

$$\mathbf{x}_n \xrightarrow{\text{Encoder}} \mathbf{z} = g_{\phi_1}(\mathbf{x}_n) + g_{\phi_2}(\mathbf{x}_n) \cdot \epsilon^{(k)} \xrightarrow{\text{Decoder}} \hat{\mathbf{x}}_n = \mu_{\theta}(\mathbf{z})$$

其中

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}(g_{\phi_1}(\mathbf{x}), \text{diag}(g_{\phi_2}^2(\mathbf{x}))), \quad p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mu_{\theta}(\mathbf{z}), \mathbf{I}), \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$





代码实现

VAE 实际代码实现细节 (主要代码在 `model.py` 的 `class VAE` 中)

Encoder

- 在实际实现中， ϕ_1, ϕ_2 这2个神经网络并不是完全独立训练的。事实上，他们可以共享比如多个卷积层搭建的 Encoder，然后再分别通过不同的1层神经网络来得到自己要输出的均值和方差。（如上图所示）
- 因为 Encoder 其实是在提取样本特征，而 ϕ_1, ϕ_2 都是根据这些样本特征来得到数据分布的均值和方差的。所以可以共享前面的 Encoder，避免重复提取特征，更加高效。

代码实现如下：

```

if nn_type == 'Linear':
    # several fully-connected linear layers
    self.encoder = nn.Sequential(
        nn.Flatten(),
        nn.Linear(image_channels * 32 * 32, out_features= 256),
        # nn.Linear(784, 256),
        nn.ReLU(),
        nn.Linear( in_features: 256, out_features: 64),
        nn.ReLU(),
        # 直接输出 mean 和 log_var
        nn.Linear( in_features: 64, latent_dim * 2)
    )

elif nn_type == 'Conv':
    # Convolution
    self.encoder = nn.Sequential(
        # 搜索得知一般使用较小的 kernel, kernel_size = 3 or 5
        # 针对 32*32 的 CIFAR-10 输入进行计算, 并且其实是[batch_size, channels, height, width] 格式
        # 3*32*32 -> 32*16*16
        nn.Conv2d(in_channels=image_channels, out_channels=32, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2),
        # 32*16*16 -> 64*8*8
        nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2),
        # 64*8*8 -> 128*4*4
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2),
        # 128*4*4 -> 256*2*2
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2)
    )

    # should flatten first
    self.flatten = nn.Flatten() # [batch_size, 256*2*2] -> [batch_size, 256*2*2]
    # train mean (for CNN output)
    self.fc_mean = nn.Linear(in_features=256*2*2, out_features=latent_dim)
    # train log_variance (for CNN output)
    self.fc_log_var = nn.Linear(in_features=256*2*2, out_features=latent_dim)
    # 要先把隐变量恢复成 256*2*2 的格式 (for CNN output)
    # [batch_size, latent_dim] -> [batch_size, 256*2*2]
    self.recover = nn.Linear(in_features=self.latent_dim, out_features=256*2*2)

```

并且Encoder之后的 ϕ_2 ，往往生成对数方差 $\log \sigma^2$ 而不是直接的方差 σ^2 ，搜索后得知，有如下原因：

- 方差 σ^2 在数学上需要是正值，但神经网络预测后可能会输出负值，从而导致负的方差在数学上无效；取对数就能避免这个问题，根据 $\exp(\log \sigma^2)$ 就能恢复得到正的方差。
- 取对数运算之后，输出的值不会过于极端，更具有数值稳定性，便于优化。

VAE-Loss

由于我们需要的是最小化损失函数，来进行梯度下降、反向传播训练神经网络参数，所以损失函数要对上面的下界函数公式取负号。

- 实际计算时，由于 CIFAR-10 输入数据是在 $[0, 1]$ 范围内的，所以把计算公式中的均方误差改为用交叉熵损失来衡量。
- 实际实现参考了这个公式：其中，采样数 K 取 1 就好。

$$\begin{aligned} \text{VAE-Loss} &= -\mathcal{L} = \text{Reconstruction-Loss} + \text{KL-Divergence} \\ &= -\sum_i (x_i \log \hat{x}_i + (1 - x_i) \log(1 - \hat{x}_i)) - \frac{1}{2} \sum_j (1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2) \end{aligned}$$

代码实现如下：

```
def VAE_loss(x, recon_x, mean, log_var):
    """
    loss function to be minimized, loss = -ELBO
    :param x: input sample x
    :param recon_x: reconstructed x
    :param mean:
    :param log_var:
    :return: loss
    """

    # BCELoss函数的前 1 个参数是预测值，后 1 个参数是样本真实值
    # reduction不能改成 mean，否则反而降低了这个recon_loss在计算公式中所占的权重，最后会导致特征丢失
    recon_loss = nn.BCELoss(reduction='sum')(recon_x, x)
    # use ".pow(2)"，按元素求幂
    kl_div = -0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp())
    loss = recon_loss + kl_div
    return loss
```

Decoder

解码器结构，相当于是Encoder反过来，比如卷积→反卷积。

不过这个时候要注意最后一层通过Sigmoid，因为要和CIFAR-10一致，控制输出的数据值在 $[0, 1]$ 范围内。

```

# decoder
if nn_type == 'Linear':
    # linear
    self.decoder = nn.Sequential(
        nn.Linear(latent_dim, out_features: 64),
        nn.ReLU(),
        nn.Linear( in_features: 64, out_features: 256),
        nn.ReLU(),
        # nn.Linear(256, 784),
        nn.Linear( in_features: 256, image_channels * 32 * 32),
        nn.Sigmoid()
    )

elif nn_type == 'Conv':
    # Convolution
    self.decoder = nn.Sequential(
        # 256*2*2 -> 128*4*4
        nn.ConvTranspose2d( in_channels: 256, out_channels: 128, kernel_size: 3, stride: 2, padding: 1, output_padding: 1),
        nn.LeakyReLU(0.2),
        # 128*4*4 -> 64*8*8
        nn.ConvTranspose2d( in_channels: 128, out_channels: 64, kernel_size: 3, stride: 2, padding: 1, output_padding: 1),
        nn.LeakyReLU(0.2),
        # 64*8*8 -> 32*16*16
        nn.ConvTranspose2d( in_channels: 64, out_channels: 32, kernel_size: 3, stride: 2, padding: 1, output_padding: 1),
        nn.LeakyReLU(0.2),
        # 32*16*16 -> 3*32*32
        nn.ConvTranspose2d( in_channels: 32, image_channels, kernel_size: 3, stride: 2, padding: 1, output_padding: 1),
        nn.Sigmoid()
    )

```

重参数化

按照隐变量 z 对变量 x 的后验分布来生成。

```

def reparameterize(self, mean, log_var):
    """
    reparameterization, to get z = mean + var * noise
    :param mean:
    :param log_var:
    :return: latent variable z
    """
    sqrt_var = torch.exp(0.5 * log_var)
    noise = torch.randn_like(sqrt_var)
    z = mean + sqrt_var * noise
    return z

```

VAE的前向传播过程

```

def forward(self, x):
    """
    forward: x -> z -> recon_x
    :param x: sample x
    :return: recon_x, latent_mean, latent_log_var
    """
    if self.nn_type == 'Conv':
        # encode
        e = self.encoder(x) # [batch_size, 256, 2, 2]
        e = self.flatten(e) # flatten: [batch_size, 256*2*2]
        mean = self.fc_mean(e) # [batch_size, latent_dim]
        log_var = self.fc_log_var(e) # [batch_size, latent_dim]

        # latent variable
        z = self.reparameterize(mean, log_var)

        # decode
        d = self.recover(z) # [batch_size, latent_dim] -> [batch_size, 256*2*2]
        # 要重塑为 4 维张量: [batch_size, 256*2*2] -> [batch_size, 256, 2, 2]
        d = d.view(-1, 256, 2, 2)
        recon_x = self.decoder(d)

    elif self.nn_type == 'Linear':
        # encode
        e = self.encoder(x)
        mean, log_var = e[:, :, :self.latent_dim], e[:, :, self.latent_dim:]

        # latent variable
        z = self.reparameterize(mean, log_var)

        # decode
        flatten_x = self.decoder(z)
        recon_x = flatten_x.view(x.size(0), self.image_channels, 32, 32)

    return recon_x, mean, log_var

```

Generate

- 生成新图片时，就不再需要再根据后验分布，就像上面进行的概率推导那样，直接依据下面的标准高斯分布来生成隐变量，再根据变量 x 对变量 z 的后验分布生成新的 x ，也就是通过训练得到的 decoder 结构来生成 x ：

$$z \sim \mathcal{N}(z; \mathbf{0}, I)$$

```

model.eval()
with torch.no_grad():
    # z ~ N(0, I)
    z = torch.randn(num, latent_dim).to(device)
    generated_img = model.generate(z)

```

```

def generate(self, z):
    """
    generate new x from latent variable z
    :param z: latent variable with noise
    :return: generated x
    """

    if self.nn_type == 'Conv':
        # decode
        d = self.recover(z) # [batch_size, latent_dim] -> [batch_size, 256*2*2]
        d = d.view(-1, 256, 2, 2) # [batch_size, 256*2*2] -> [batch_size, 256, 2, 2]
        generated_x = self.decoder(d)

    elif self.nn_type == 'Linear':
        # decode
        flatten_x = self.decoder(z)
        generated_x = flatten_x.view(-1, self.image_channels, 32, 32)

    return generated_x

```

Preprocess

- 在实验中，如果直接使用 CIFAR-10数据集，那当然会把10类数据的特征混合在一起，经过查找资料后，发现可以使用 CVAE 或者 只用1类的数据进行训练。

只使用1类训练：

```

def get_single_class_cifar10(class_label, batch_size=64):
    """
    only get one class Data in CIFAR-10
    :return : DataLoader
    """

    transform = transforms.Compose([
        transforms.ToTensor()
    ])
    full_dataset = datasets.CIFAR10(root='./data', train=True, transform=transform, download=True)
    indices = [idx for idx, (_, label) in enumerate(full_dataset) if label == class_label]
    subset = Subset(full_dataset, indices)
    return DataLoader(subset, batch_size=batch_size, shuffle=True)

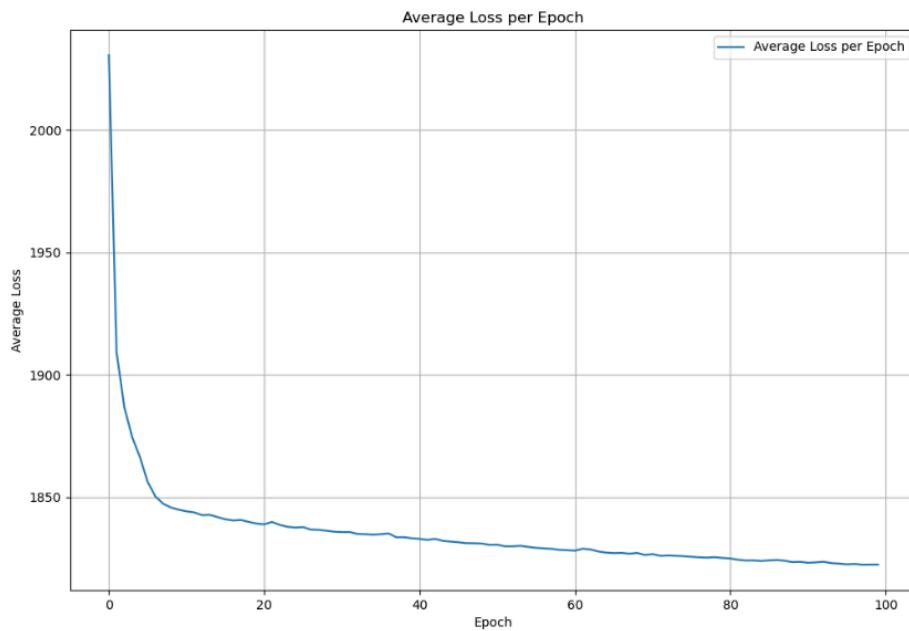
```

实验结果与分析

主要针对选择 `latent_dim=20` 时的实验结果进行分析，更多的实验结果在文件夹中。

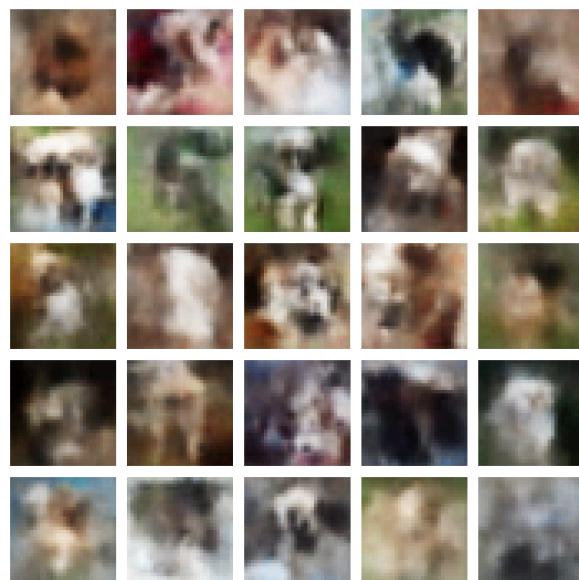
- 发现学习率取1e-3较合适，如果1e-2，就导致loss发散了。

在合适的学习率下，Loss的下降曲线：



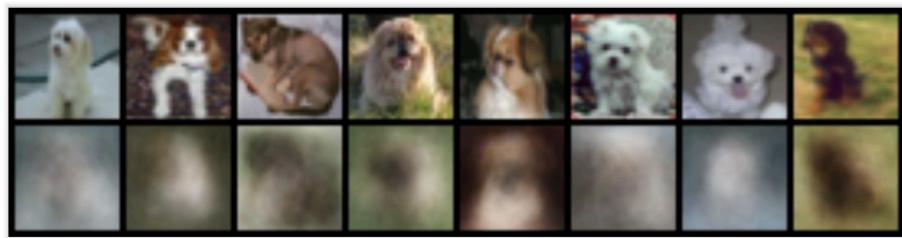
说明神经网络确实按照我们的目标函数在训练。

生成图片的效果：



- 使用线性全连接层会比使用CNN的网络训练得快。但得到的图片会更模糊，效果不如卷积层好。

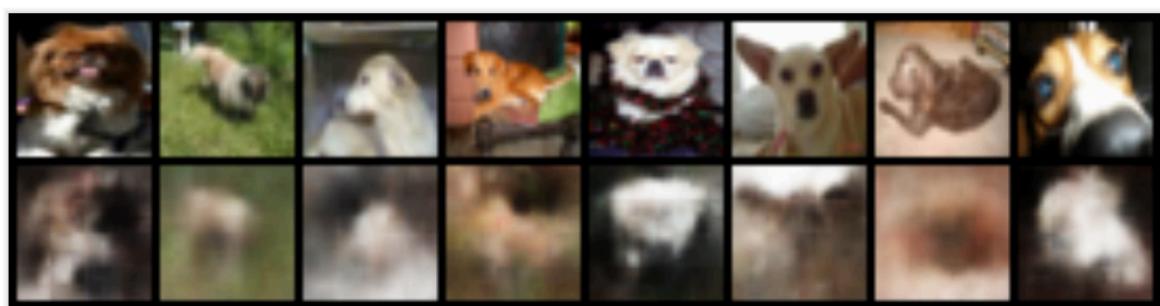
100个epoch的Linear网络重构效果：



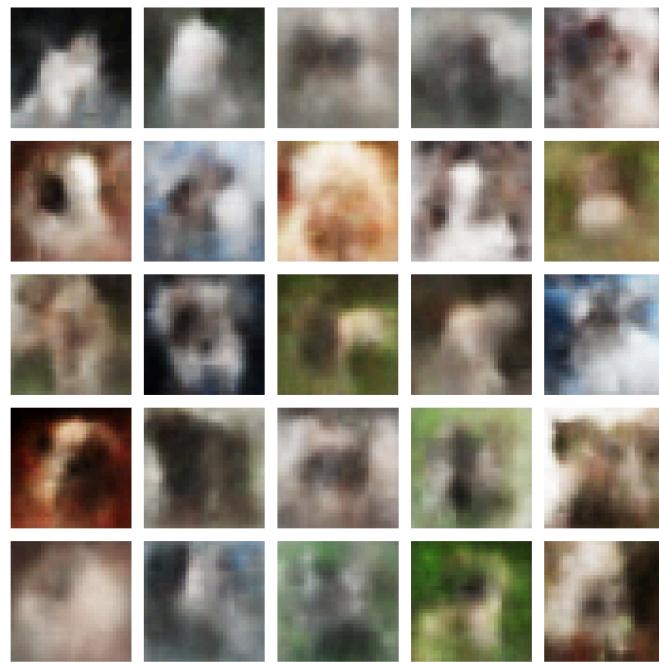
生成效果：



100个epoch的Convolution网络重构效果：



生成效果：

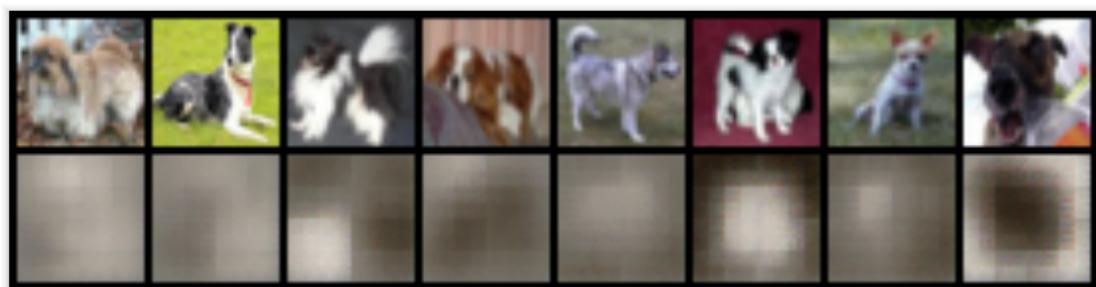


CNN用在VAE里，能更好地提取图像特征。

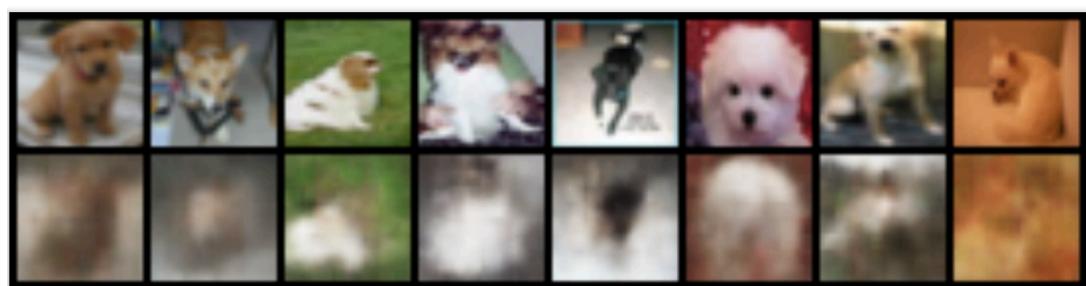
- 根据我们逐步打印的重构图像，可以看到VAE的训练效果：

使用CNN的VAE经过300个epochs的效果：

epoch1：这个时候网络还没有学到太多的信息



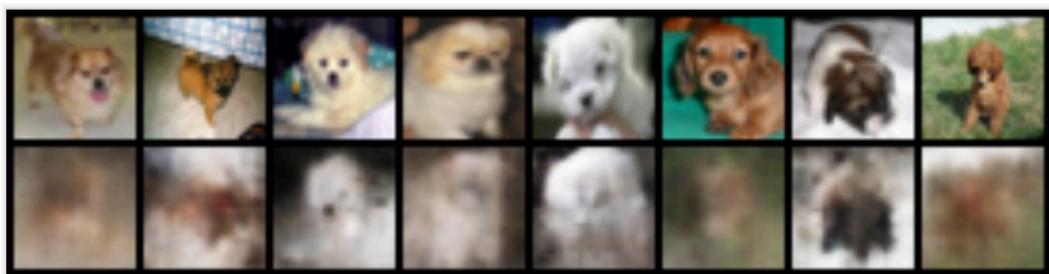
epoch101：已经能学到比较模糊的图像信息了



epoch203:



epoch343:



虽然总是很模糊，但可以看到逐渐变好的重构效果。

(考虑到最后Loss函数还是在慢慢下降的，所以再继续训练更多的epochs可能可以得到更好的重构、生成图像效果。)

- 无论怎么训练，VAE生成的图像总是很模糊的，一方面可能是我的VAE模型实际搭建实现的问题，另一方面就是VAE本身的特点。查找资料后得知，应该有大致如下的原因：
 - 根据VAE的推导，我们总是在用高斯分布去近似实际的后验分布，但实际的分布很可能更复杂多样，不管怎么近似，高斯分布总是有一定的差距的；
 - 在优化目标中，要同时考虑KL散度和重构误差，于是为了KL散度的优化，就可能牺牲了一部分降低重构误差的能力；
 - decoder的结构并不复杂（比GAN的decoder结构简单），因此不会生成太清晰的图像；
 - VAE通常使用二元交叉熵损失（BCELoss）或均方误差损失（MSELoss）作为重构误差，而这些损失函数在高分辨率图像重构时表现有限，因为它们无法捕捉图像中的高频细节；
 - 使用的神经网络结构还不够好；
 - 隐变量维度较低，无法捕捉图像的细节信息
- 针对上面的原因2，我尝试简单修改VAE-LOSS函数如下（降低KL-散度项的占比，让VAE更接近AE）：

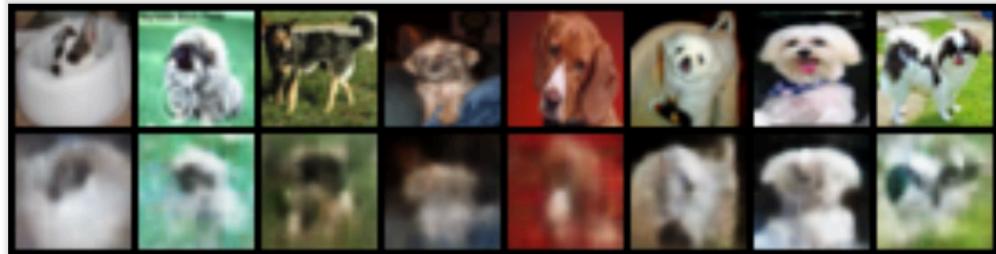
```

def VAE_loss(x, recon_x, mean, log_var):
    """
    loss function to be minimized, loss = -ELBO
    :param x: input sample x
    :param recon_x: reconstructed x
    :param mean:
    :param log_var:
    :return: loss
    """

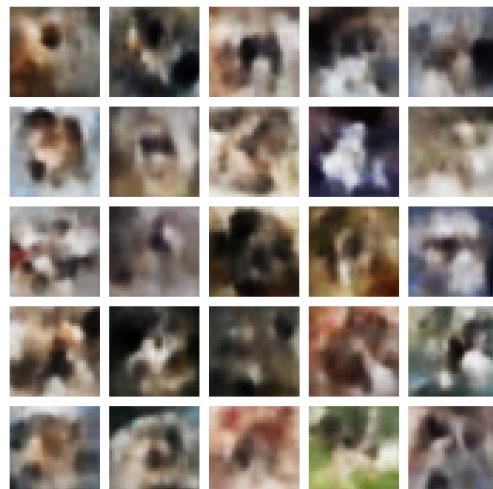
    # BCELoss函数的前 1 个参数是预测值，后 1 个参数是样本真实值
    # reduction不能改成 mean，否则反而降低了这个recon_loss在计算公式中所占的权重，最后会导致特征丢失
    recon_loss = nn.BCELoss(reduction='sum')(recon_x, x)
    # use ".pow(2)" , 按元素求幂
    kl_div = -0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp())
    loss = 3 * recon_loss + kl_div
    return loss

```

此时的重构效果更好了一点：



但生成的效果并未改进，看上去生成的图片的多样性降低了（因为降低了KL散度的影响），而且**尤其是图片内部的连续程度降低了（内部结构没那么光滑）**：



原本的VAE的生成效果：



结论

VAE 是一种基于概率统计理论的深度生成模型，经CIFAR-10数据集训练后，能生成较模糊的相似图片（原因分析见上面实验结果部分）。如果还需要提高生成图片的质量，应该还要和GAN、Diffusion模型相结合。