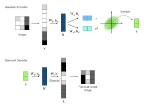
理解变分自编码器（VAE）

知 zhuanlan.zhihu.com/p/519448634



Intro

变分自编码器（Variational AutoEncoder，VAE）是深度学习中常用的无监督学习方法，可以用来做数据生成，表征学习，维度压缩等一系列应用。由于架构上的相似性，VAE常常和自编码器（AutoEncoder，AE）联系在一起，但是VAE名字中的变分二字又彰显了它的不俗之处。所以通过这篇文章，我们就来一探究竟，以图更好地理解变分自编码器。

我们会首先介绍它提出的背景，通过推导理解变分方法在VAE中的作用和意义，比较VAE和AE之间的异同之处。最后，我们还会用pytorch来实现一个简单的VAE，并且在MNIST数据集上做演示，感兴趣的同学可以查看发布在GitHub上的Notebook，或者直接在 Colab 上运行查看结果（记得开GPU加速）。

PART I： 概念与推导

这一部分我们从统计视角出发，对VAE做一些基本的数学推导，这有助于我们理解VAE到底是怎么和变分方法扯上关系的，它和AutoEncoder的区别和联系又在哪里。

问题背景

首先我们假设有一个由 N 个独立同分布的数据点组成的数据集 $X=\{x^{(i)}\}_{i=1}^N$ ，其中的每一个数据点的生成遵循如下生成式过程：

1. 从某个先验分布 $p_\theta(z)$ 中采样得到隐变量 $z^{(i)}$；
2. 从条件分布 $p_\theta(x|z=z^{(i)})$ 中采样得到数据点 $x^{(i)}$。

其中，先验分布 $p_\theta(z)$ 和条件似然函数 $p_\theta(x|z)$ 都是参数 θ 的函数（这个函数可以是神经网络，也可以是任何其他数学模型）。此外，隐变量 $z^{(i)}$ 的取值也是不可观测的，只有数据点 $x^{(i)}$ 可观测。

基于上述背景，我们关注的问题有以下三个：

1. **参数估计**问题：即在仅知道数据集$X=\{x^{(i)}\}_{i=1}^N$ 的前提下，如何对上述的生成模型 θ 做参数估计
2. **后验推断**问题：即给定一组参数 θ后，在给定某个数据点 $x^{(i)}$ 后，计算生成这个数据点的隐变量的概率 $p_\theta(z|x)$
3. **边际分布推断**问题：即给定一组参数 θ后，直接计算边际分布 $p_\theta(x^{(i)})$

以上三个问题在深度学习领域都是很有应用价值的课题：解决了参数估计问题，我们就能全面刻画数据的生成过程，就能做图像生成了；解决了后验推断问题，我们就能对数据点 x 做表征学习，或者维度压缩；解决了边际分布推断问题，边际分布 $p(x)$ 就能适用于任何需要对数据 x做先验假设的场景。

到底如何一次性解决上述三个问题呢？我们会层层深入，讨论VAE到底作出了什么贡献。

变分方法

变分方法是我们要重点介绍的内容，它在推导VAE的关键步骤。上面提到的三大问题，我们先关注“后验推断“问题，它是引入变分的关键。通过贝叶斯公式，后验概率可以拆分为（推导省去参数 θ ）：

$$p(z|x^{(i)}) = \frac{p(z,x^{(i)})}{p(x^{(i)})}$$

$$= \frac{p(x=x^{(i)}|z=z^{(i)})p(z=z^{(i)})}{\int_{z^{(i)}} p(x=x^{(i)}|z=z^{(i)})p(z=z^{(i)})dz^{(i)}}$$

$$= \frac{p(x^{(i)}|z)p(z)}{\int_z p(x^{(i)}|z)p(z)dz} \quad \text{simplifythenotation}$$

我们先假设参数 θ 已知，那么先验分布 $p_\theta(z)$ 和条件似然函数 $p_\theta(x^{(i)}|z)$ 就都是已知的。理论上来说，只要我们把分母里的积分项 $\int_z p_\theta(x^{(i)}|z)p(z)dz$ 计算出来，那整个后验分布 $p(z|x^{(i)})$ 就可以求了，后验推断问题也就解决了。但是，现实很骨感，在没有对 $p_\theta(z)$ 和 $p_\theta(x^{(i)}|z)$ 作任何简化假设的前提下，这个积分基本上是没有解析解的。你想硬着头皮解，那么基本意味着你要穷举隐变量 z 的所有可能取值，假设 z 有 k 个维度，每个维度采样 n 个取值，那么这个穷举过程的复杂度就是 $O(n^k)$。

当然也有人用MCMC来做积分项的估计，虽然这个方案做采样估计很精准，但是费时费力，很难适用于大数据场景。所以一般更常见的方案是采用变分方法（variational method），它可以绕过对积分项的求解，通过把统计推断问题转化成参数优化问题来实现“降维打击”。

首先变分方法会设置一个新的参数化分布 $q_\phi(z|x^{(i)})$，它的参数是 ϕ，我们把它称作“识别模型”（原文记作recognition model）。变分方法的核心思想是：直接让“识别模型”去拟合后验分布 $p_\theta(z|x^{(i)})$，只要近似到位，那么采用 $q_\phi(z|x^{(i)})$ 作为后验推断的结果就行了。如何做近似呢？很简单，直接最小化 $q_\phi(z|x^{(i)})$ 和 $p_\theta(z|x^{(i)})$ 两者间的KL散度即可。

就这样，变分方法把原来的统计推断问题转化成了优化问题：

$$\phi^*, \theta^* = \operatorname{argmin}_{\phi,\theta} KL(q_\phi(z|x^{(i)}) || p_\theta(z|x^{(i)}))$$

在上面这个优化式子当中，参数 ϕ 和 θ 是联合优化的。下面我们进一步推导：

$$
\begin{aligned}
KL(q(z|x^{(i)})||p(z|x^{(i)})) &= \int_z q(z|x^{(i)}) \log\frac{q(z|x^{(i)})}{p(z|x^{(i)})}dz \\
&= E_q[\log q(z|x^{(i)})] - E_q[\log p(z|x^{(i)})] && \text{rewrite as the form of expectation} \\
&= E_q[\log q(z|x^{(i)})] - E_q[\log p(x^{(i)},z)] + E_q[\log p(x^{(i)})] && p(z|x^{(i)}) = \frac{p(x^{(i)},z)}{p(x^{(i)})} \\
&= E_q[\log q(z|x^{(i)})] - E_q[\log p(x^{(i)},z)] + \log p(x^{(i)}) && p(x^{(i)}) \text{ is irrelevant of } q \\
&= -\text{ELBO} + \log p(x^{(i)})
\end{aligned}
$$

上式中 $\log p(x^{(i)})$ 是常数项，在优化过程中可以直接忽略，而ELBO（evidence lower bound）可以重写成以下形式：

$$
\begin{aligned}
\text{ELBO} &= -E_q[\log q(z|x^{(i)})] + E_q[\log p(x^{(i)},z)] \\
&= -E_q[\log q(z|x^{(i)})] + E_q[\log p(z)] + E_q[\log p(x^{(i)}|z)] \\
&= -KL(q(z|x^{(i)})||p(z)) + E_q[\log p(x^{(i)}|z)]
\end{aligned}
$$

最终，优化问题可以写成下列形式：

$$
\phi^*, \theta^* = \text{argmax}_{\phi,\theta} - KL(q_\phi(z|x^{(i)})||p_\theta(z)) + E_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)}|z)]
$$

The Learning Algorithm

上一小节告诉我们，使用变分方法可以把后验推断问题转化成参数优化问题。只要解决了这个优化问题，第一节中提到的“后验推断”问题就可以引刃而解，而且由于求解得到了 θ，那么“参数估计”问题也就解决了。同时当后验分布 $p(z|x^{(i)})$ 已知后，边际分布通过贝叶斯公式也可以立马得到，“边际分布推断”问题也变得可解。所以解决上述的优化问题，是一举三得的。

那么如何解决这个优化问题呢？而这个问题在深度学习中，是最不成问题的问题，直接上随机梯度下降！

在这里我们把原本需要最大化的ELBO写成大家相对熟悉的可最小化的损失函数形式：

$$L(\phi,\theta,x^{(i)}) = KL(q_\phi(z|x^{(i)})||p_\theta(z)) - \frac{1}{L}\sum_{l=1}^{L}[\log p_\theta(x^{(i)}|z^{(i,l)})]$$

除了正负号区别以外，最引人瞩目的是原来的期望项 $E_q[\log p(x^{(i)}|z)]$ 发生了变化：我们对期望采用了MCMC采样做近似。样本 $z^{(i,l)}$ 采样自分布 $q_\phi(z|x^{(i)})$ ，通过对样本求平均来代替期望。

而在实践中，一般不对 $q_\phi(z|x^{(i)})$ 直接作采样，采用 **reparameterization trick** 来简化操作，我们设 $z^{(i,l)} = g_\phi(\epsilon^{(i;l)};x^{(i)})$ ，其中 g_ϕ 是一个拟合函数（e.g. 神经网络），而噪声 $\epsilon^{(i;l)}$ 可以通过采样得到，一般直接采样自简单的标准正态分布。

采用reparameterization trick有两大好处：

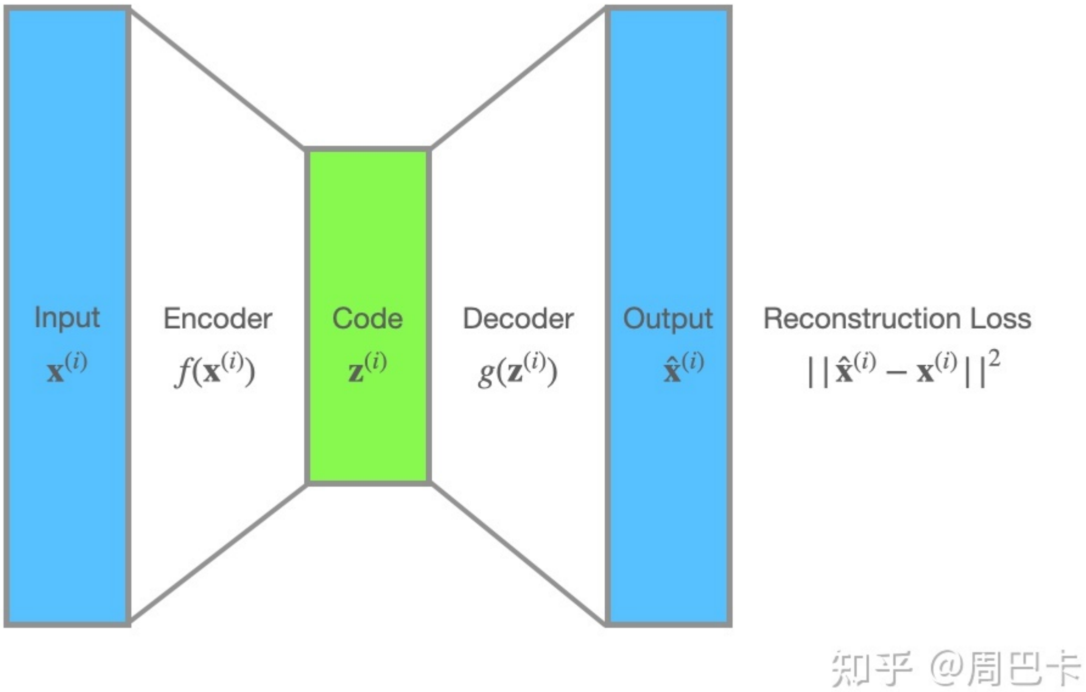
- 由于分布 $q_\phi(z|x^{(i)})$ 可能是一个比较复杂的函数，直接采样操作费时费力，而且采样方差可能很大，不利于收敛，通过reparameterization可以简化操作，提高效率，提高数值上的稳定性；
- 假设我们不考虑采样难度，直接对 $q_\phi(z|x^{(i)})$ 采样，那么梯度反向传播的时候，损失函数中的 $\frac{1}{L}\sum_{l=1}^{L}[\log p_\theta(x^{(i)}|z^{(i,l)})]$ 是没法对 ϕ 求导的，这样损失函数 $L(\phi,\theta,x^{(i)})$ 只能通过KL散度的梯度对 ϕ 做优化，这和我们做联合参数优化的意图是违背的。所以使用reparameterization trick 让 $z^{(i,l)} = g_\phi(\epsilon^{(i;l)};x^{(i)})$ ，实际上是让参数 θ,ϕ 可以同时得到期望项和KL散度项的反传梯度进行优化，让模型学得更好。

总结一下，整个VAE 的 Learning 过程可以用如下步骤表示：

1. 从数据集中得到minibatch，batch_size 为 M；
2. 计算minibatch loss $\frac{1}{M}\sum_{i=1}^{M}L(\phi,\theta,x^{(i)})$ ；
3. 反向传播计算梯度 $\frac{1}{M}\sum_{i=1}^{M}\nabla_{\phi,\theta}L(\phi,\theta,x^{(i)})$ ；
4. 梯度更新到参数 ϕ,θ；
5. 重复前4个步骤直至收敛。

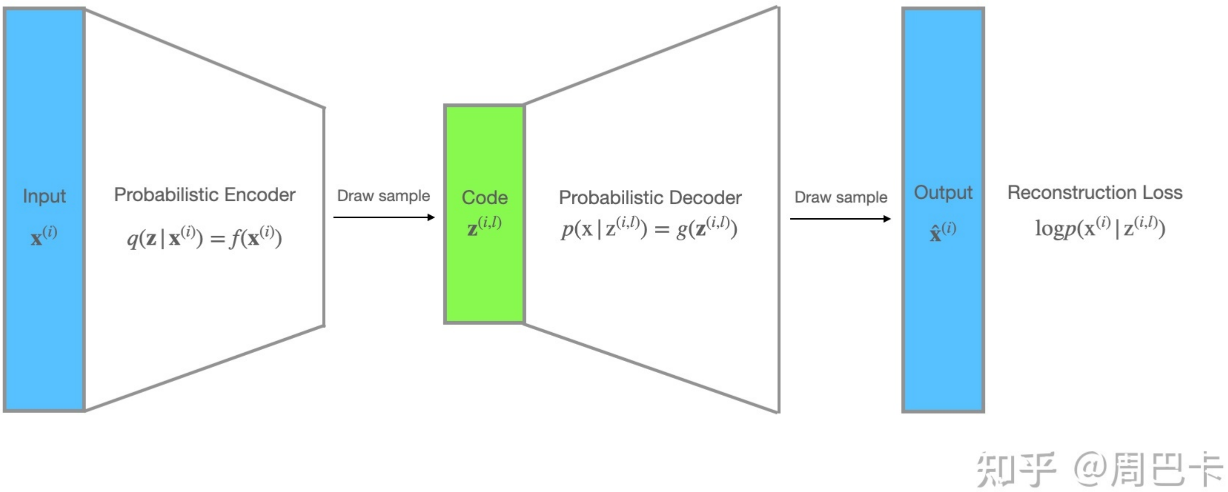
VAE vs. AE

通过前面复杂的数学推导，我们描述了VAE要解决的三大问题，介绍了变分方法在VAE中起到的作用，还描述了VAE模型的学习方法。这也基本讲清楚了变分自编码器这个学名中“变分”二字的由来，那么这一小节我们来讲剩下的“自编码器”部分。



Auto Encoder Arch

首先我们回顾一下自编码器的结构：AutoEncoder的结构很简单，每一个数据点 $x^{(i)}$ 都会经过Encoder网络，随后得到表征 $z^{(i)}$（也叫code），而Decoder网络会接受 $z^{(i)}$ 作为输入，最终输出的是重构数据 $\hat{x}^{(i)}$，重构损失通常用平方误差 $||\hat{x}^{(i)} - x^{(i)}||^2$。



现在我们用AutoEncoder的观点来审视刚才的推导出来的损失函数（negative ELBO）和其中涉及到两个模型 $q_{\phi}(z|x)$ 和 $p_{\theta}(x|z)$：

$$L(\phi,\theta,x^{(i)})=KL(q_\phi(z|x^{(i)})||p_\theta(z))-\frac{1}{L}\sum_{l=1}^{L}[\log p_\theta(x^{(i)}|z^{(i,l)})]$$

仔细观察“识别模型” $q_\phi(z|x)$ 的形式，实际上就是以 x 为输入 z 为输出的Encoder，只不过相比于AE中的Encoder，它输出的不是确定的值，而是 z 所有可能取值的分布。同样的，似然函数 $p_\theta(x|z)$ 可以对应Decoder，对于给定的一个隐变量 z，它给出的是样本点 x 所有可能取值的分布。

反观损失里面的期望项 $-E_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)}|z)]\sim-\frac{1}{L}\sum_{l=1}^{L}[\log p_\theta(x^{(i)}|z^{(i,l)})]$，其本质就是Negative Log Likelihood损失，一般用作输出是连续分布的模型的损失函数，这一项可以对应为AE里的重构损失。当然除了重构损失以外，VAE还更加高级地引入了KL散度项作为正则，它的目的是为了尽可能让“识别模型”的输出分布 $q_\phi(z|x^{(i)})$ 和先验分布 $p_\theta(z)$ 接近。

总结

我们总结一下第一部分几个要点：

1. VAE的提出帮助解决了含有隐变量的生成式模型下的三大问题：参数估计，后验推断，边际推断；
2. 通过变分方法，我们可以构造出损失函数为Negative ELBO的参数优化问题，并通过reparameterization trick 和 随机梯度下降进行学习；
3. 变分方法引入的“识别模型” $q(z|x)$ 和原有的条件似然函数 $p(x|z)$ 恰好可以对照AutoEncoder结构，解释为Probabilistic Encoder和Probabilistic Decoder，而原本的损失函数可解释为参数正则项和重构损失的组合。

PART II：VAE 实践 - MNIST

第一部分我们对VAE的理论基础和导出过程做了全面介绍，有了一定的理论基础，第二部分我们聚焦于实战，着手自己实现一个VAE，并且在MNIST手写数字数据集上做一些简单的实验。

MNIST VAE

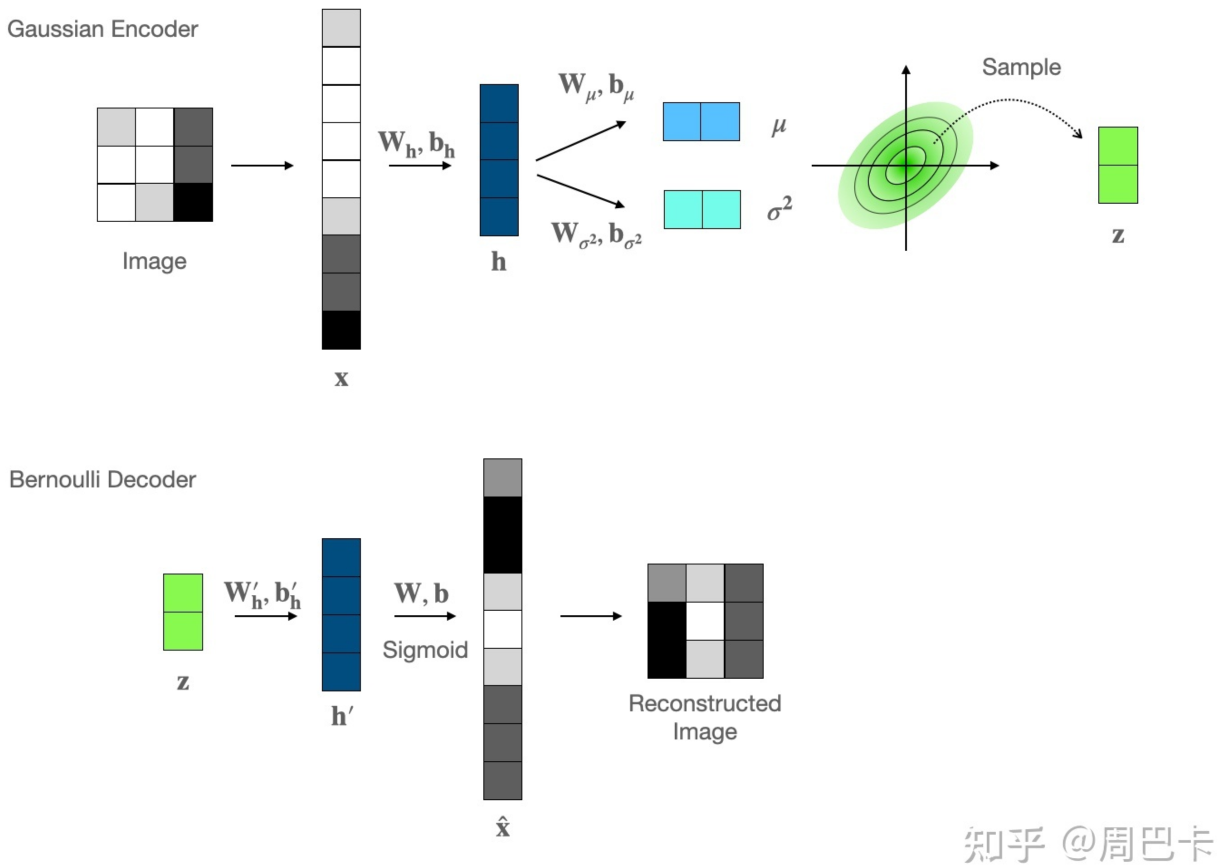
我们针对MNIST数据集专门设计了专属的VAE，它由一个高斯分布编码器和伯努利分布解码器组成。

高斯分布编码器：这里我们选用了多元高斯分布作为编码器的输出分布，它的均值和方差是都是向量，可以由一个小神经网络来建模。编码器的参数 ϕ 包括了：$W_h,b_h,W_\mu,b_\mu,W_{\sigma^2},b_{\sigma^2}$。下面的式子就刻画了高斯分布编码器的架构：

$$\begin{aligned}
h &= tanh(W_hx+b_h)\\
\mu &= W_\mu h+b_\mu\\
\sigma^2 &= exp(W_{\sigma^2}h+b_{\sigma^2})\\
q(z|x) &= N(z;\mu,\sigma^2 I)
\end{aligned}$$

伯努利分布解码器：由于MNIST数据集的样本是黑白图片，本质就是每个像素取值都是0~1之间的矩阵，那么使用伯努利分布作为解码器的输出是非常合适的。f_σ 是sigmoid激活函数，解码器参数 θ 包括了：$W_h^{'},b_h^{'},W,b$。

$$
\begin{aligned}
h' &= tanh(W_h' z + b_h') \\
p(x|z) &= f_\sigma(Wh' + b)
\end{aligned}
$$



Vanilla MNIST VAE

损失函数：VAE的损失函数 $L(\phi, \theta, x^{(i)})$ 由KL散度项和期望项共同构成。我们先来求KL散度项 $KL(q_\phi(z|x^{(i)})||p_\theta(z))$，假设先验分布 $p_\theta(z) = N(0, I)$。“识别模型”为多元高斯分布 $q_\phi(z|x) = N(z; \mu, \sigma^2 I)$，其中 $\mu \in R^J, \sigma^2 \in R_+^J$，$\mu_j, \sigma_j^2$ 为向量中第 j 个分量。两个高斯分布求KL散度，可以直接写出：

$$
KL(q_\phi(z|x^{(i)})||p_\theta(z)) = \frac{1}{2}\sum_{j=1}^{J}(1 + 2\log\sigma_j - \mu_j^2 - \sigma_j^2)
$$

而对于期望项，则取采样数量 $L = 1$，通过MCMC采样，期望项代表的重构损失就变成了 $\log p(x^{(i)}|z^{(i,1)})$，其中code的采样点通过reparameterization trick得到：$z^{(i,1)} = \mu + \sigma \odot \epsilon$，噪声 $\epsilon \sim N(0, I), \epsilon \in R^J$，$\mu, \sigma^2$ 是高斯分布编码器中的均值和方差。

两项合并即可得到VAE需要优化的损失函数：

$$L(\phi, \theta, x^{(i)}) = \frac{1}{2}\sum_{j=1}^{J}(1 + 2\log\sigma_j - \mu_j^2 - \sigma_j^2) - \log p(x^{(i)} \mid z^{(i,1)}) z^{(i,1)} = \mu + \sigma \odot \epsilon \epsilon \sim N(0, I)$$

代码实现

```
class GaussianEncoder(torch.nn.Module):
    """
    modeling the prob q(z|x), where z, x are n-d, m-d vectors
    """
    def __init__(self, dim_z, dim_x, dim_hidden):
        """
        dim_z: the dimensionality of the latent variable
        dim_x: the dimensionality of the input variable
        dim_hidden: the dimensionality of the hidden vector h
        """
        super(GaussianEncoder, self).__init__()
        self.hidden_layer = torch.nn.Sequential(
                                torch.nn.Linear(dim_x, dim_hidden),
                                torch.nn.Tanh()
                                )

        # transform hidden vector to gaussian mean
        self.mean_transform_layer = torch.nn.Linear(dim_hidden, dim_z)

        # transform hidden vector to gaussian variance
        self.var_transform_layer = torch.nn.Linear(dim_hidden, dim_z)

    def get_mean_and_var(self, x):
        """
        :param x: the condition part, [batch, m]
        :return: (mean, variance) [batch, n], [batch, n]
        """

        h = self.hidden_layer(x)  # [batch, h]
        return (self.mean_transform_layer(h),
                torch.exp(self.var_transform_layer(h)))

    def forward(self, z, x):
        """
        give the log prob of p(z|x)
        :param z: [batch, n]
        :param x: [batch, m]
        :return: [batch, ]
        """
        dim_z = z.shape[1]
        mean, var = self.get_mean_and_var(x)  # [batch, n], [batch, n]

        # inversed covariance mat, [b, n, n]
        inv_covar = torch.einsum('bi, ij -> bij',
                                 1 / var,
                                 torch.eye(dim_a))
        # gaussian pdf
        exponent = - 1 / 2 * torch.einsum('bi, bi -> b',
                                          torch.einsum('bi, bij->bj',
                                                       z - mean,
                                                       inv_covar),
                                          z - mean)  # [b,]

        return - dim_z / 2 * torch.log(torch.tensor(2 * torch.pi)) \
               - 1 / 2 * torch.sum(torch.log(var), dim=1) + exponent


class BernoulliDecoder(torch.nn.Module):
    """
    The decoder modeling likelihood p(x|z),
    suitable for binary-valued data, or the real-value between 0 and 1
    """
    def __init__(self, dim_latent, dim_input, dim_hidden):
        super(BernoulliDecoder, self).__init__()
        self.layer = torch.nn.Sequential(
            torch.nn.Linear(dim_latent, dim_hidden),
            torch.nn.Tanh(),
```

```
            torch.nn.Linear(dim_hidden, dim_input),
            torch.nn.Sigmoid()
        )

    def forward(self, x, z):
        """
        evaluate the log - prob of p(x|z)
        :param x: [batch, n]
        :param z: the given latent variables, [b, m]
        :return: [batch, ]
        """
        y = self.layer(z)  # [b, n]
        return torch.sum(x * torch.log(y) + (1 - x) * torch.log(1 - y),
                         dim=1)

    def generate(self, z):
        """
        generate data points given the latent variables, i.e. draw x ~ p(x|z)
        :param z: the given latent variables, [batch, m]
        :return: generated data points, [batch, n]
        """
        with torch.no_grad():
            # [batch, n]
            y = self.layer(z)
            return torch.where(torch.rand(y.shape) > y, 0., 1.)

    def prob(self, z):
        """
        evaluate the conditional probability
        :param z: the given latent variables, [batch, m]
        :return: [batch, n], 0 <= elem <= 1
        """
        with torch.no_grad():
            return self.layer(z)
```

有了 Encoder 和 Decoder 代码以后，把它们组合起来，写出前向传播的过程，至此VAE的搭建就完成了。文章只列出了模型构建的代码，详细的训练和实验代码可以参见这里。

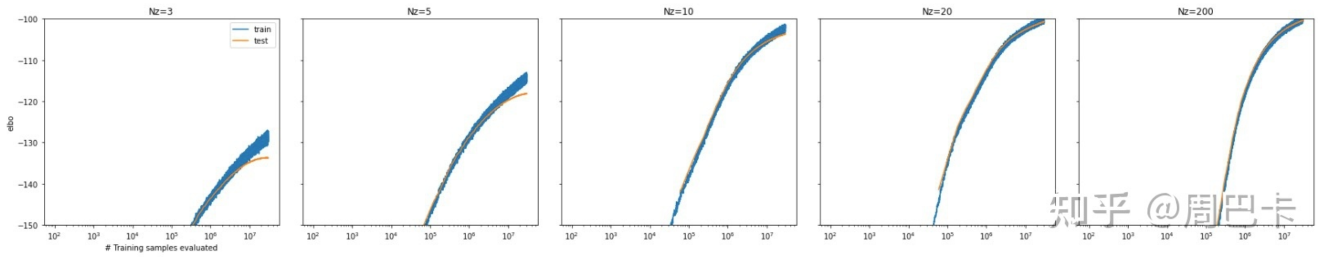
```
class VAEModel(torch.nn.Module):
    """
    the variational auto-encoder for MNIST data
    """
    def __init__(self, dim_latent, dim_input, dim_hidden):
        super(VAEModel, self).__init__()
        self.encoder = GaussianMLP(dim_latent, dim_input, dim_hidden)
        self.decoder = BernoulliDecoder(dim_latent, dim_input, dim_hidden)

    def compute_loss(self, data, reduction='mean'):
        if reduction == 'mean':
            return - torch.mean(self.forward(data))
        elif reduction == 'sum':
            return - torch.sum(self.forward(data))


    def forward(self, x) -> torch.Tensor:
        """
        corresponds to equation (10) in the original paper
        :return: the estimated ELBO value, i.e. the objective function
        """
        mean, var = self.encoder.get_mean_and_var(x)  # [b, n], [b, n]
        # draw a sample from q(z|x) by using reparameterization trick
        z = mean + torch.sqrt(var) * torch.randn(var.shape).to(x.device)
        # the KL divergence term plus the MC estimate of decoder
        return 1 / 2 * torch.sum(1 + torch.log(var) - mean ** 2 - var,
                                 dim=1) + \
                       self.decoder(x, z)
```

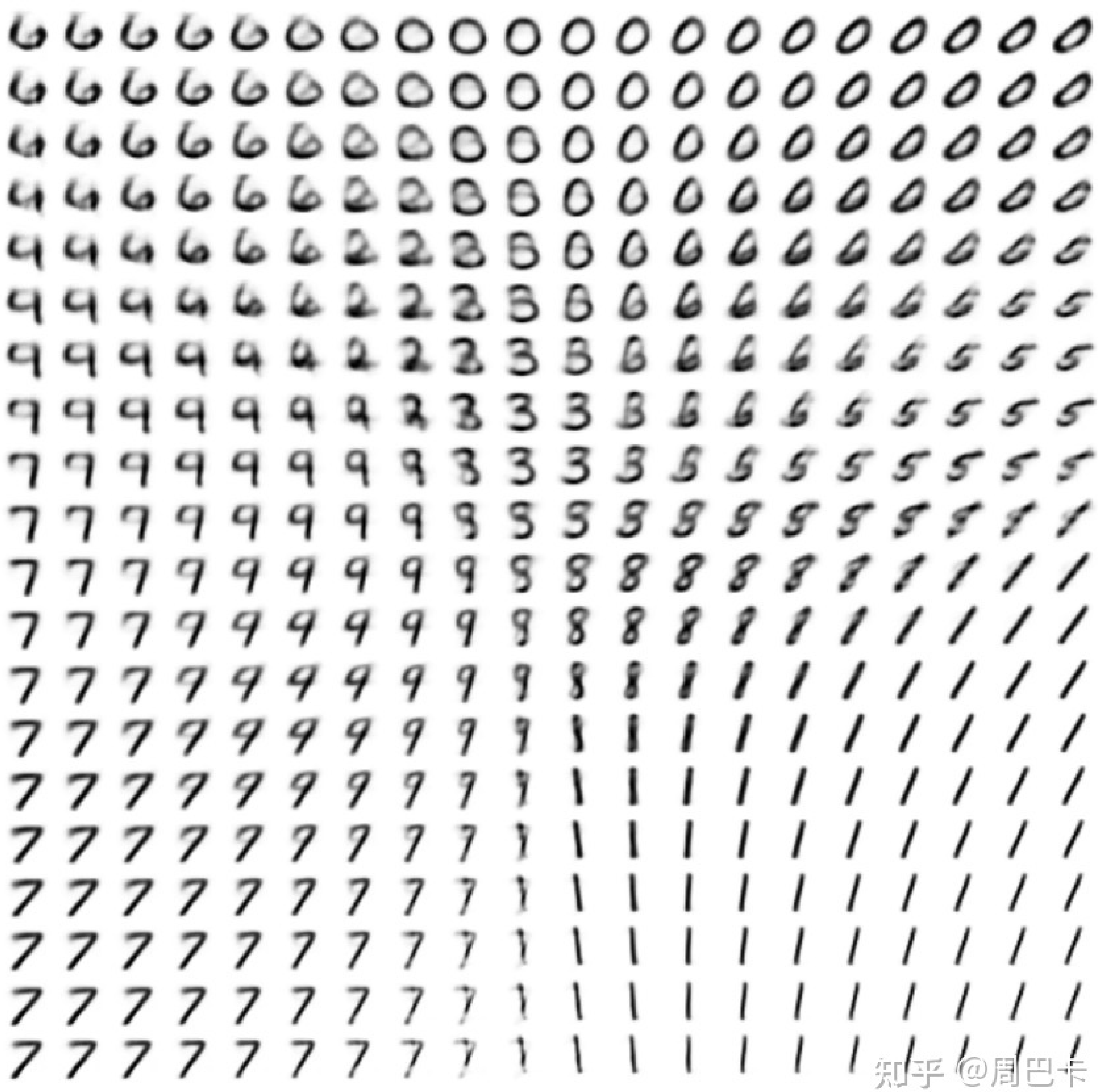
实验结果

我们首先研究一下latent variable的维度对学习性能的影响，我们把隐变量维度设置为3，5，10，20和200，分别观察了测试集上损失函数（最大化ELBO）学习曲线，可以看到维度越大，拟合性能越好。而当我们把维度设为200时，学习曲线上也没有呈现出明显的过拟合迹象，说明KL散度还是起到了一定的正则效果。



横轴为 log 模型训练样本数，纵轴为ELBO取值

此外，为了观察生成过程中隐变量 z 对所生成数据 x 的影响，我们还可视化了不同的二维隐变量所生成的图像集合。



实验代码在 colab notebook 上可以直接运行，欢迎大家去玩玩， Github上还有更多细节。