生成模型

Author: Peter Han

生成模型概述（通俗解释）

生成的核心是生成抽象化的内容，利用已有的内容生成没有的/现实未发生的内容。这个过程类似于人类发挥想象力的过程。

生成模型的应用场景非常广泛，可以应用于艺术表达，如画的生成、电影（视频）的生成，音乐生成（借助提取的风格进行生成）等。

生成模型定义与思路

生成模型：给定训练集，产生与训练集同分布的新样本。

希望学到一个模型$p_{model}(x)$，其与训练样本的分布$p_{data}(x)$相近。

无监督学习里的一个核心问题——**密度估计问题**

几种典型思路：

- 显式的密度估计：显式的定义并求解分布$p_{model}(x)$。
- 隐式的密度估计：学习一个模型$p_{model}(x)$，而无需显式地定义它。

条件概率、联合概率的贝叶斯公式[3]

条件概率：

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

联合概率：

$$P(A,B) = P(A|B) \times P(B)$$

举例说明联合概率分布：打靶时命中的坐标(x, y)的概率分布就是联合概率分布（涉及两个随机变量），其他同样类比。

联合分布的边缘分布：

- $P\{X = x_i\} = \sum_j P\{X = x_i, Y = y_j\} = \sum_j p_{ij} = p_i$
- $P\{X = x_j\} = \sum_i P\{X = x_i, Y = y_j\} = \sum_i p_{ij} = p_j$

极大似然估计[5]

e.g. 在一个未知的袋子里摸球，有若干红色和蓝色的球。此概率服从二项分布：

X	红色	蓝色
P	θ	1-θ

由于不知道袋子中究竟有多少个球以及每个颜色的球有多少个，所以无法对参数θ进行计算，也不能计算出摸到哪种颜色的球的概率是多少。于是，假设有一个测试人员对袋内球进行有放回的抽取，进行了100次随机测验之后，统计得出：有30次摸到的是红球，有70次摸到的是蓝球。

从测试结果推测，红色：蓝色=3：7，进而求得θ=0.3。

需要注意的是，极大似然估计中采样需满足一个重要的假设，就是所有的采样都是独立同分布的。

PixelRNN与PixelCNN

显式的密度模型

利用链式准则将图像x的生成概率转变为每个像素生成概率的乘积：

$p(x) = \prod_{i=1}^{n} p(x_i|x_1, \ldots, x_{i-1})$

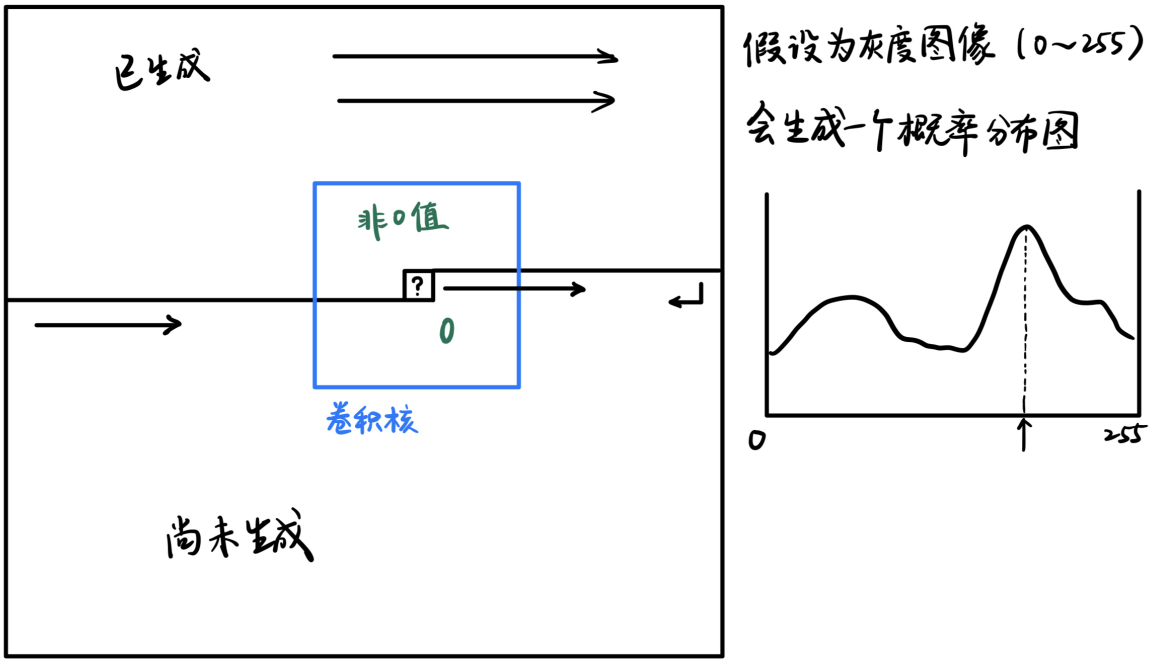
其中，$p(x)$是图像x的似然，x_i是给定已经生成的像素的前提下生成第i个像素的概率。

（描述图像的生成过程：先生成图像中的第一个点，再根据第一个点生成第二个点，再根据第一个点和第二个点生成第三个点，以此类推）

似然：密度函数

这个分布很复杂，但是可以使用神经网络来建模。

PixelCNN:



PixelRNN与PixelCNN的优缺点：

- 优点：
  - 似然函数可以精确计算
  - 利用似然函数的值可以有效地评估模型性能
- 缺点：
  - 序列产生速度慢

信息量、香农熵、交叉熵、KL散度[1]

信息量(Amount of Information)：对于一个事件，如果小概率事件发生，则事件带来了很大的信息量；如果大概率事件发生，信息量很小；对于独立的事件来说，它们的信息量可以相加。

e.g. 如果说有人中了彩票，则信息量很大；如果说没人中彩票，则信息量很小。

信息量定义：

$I(x) = \log_2\left(\frac{1}{p(x)}\right) = -\log_2\left(p(x)\right)$

e.g. 抛硬币 h代表正面向上 t代表反面向上

均匀硬币：

$p(h) = 0.5\ I_p(h) = \log_2 \frac{1}{0.5} = 1$

$p(t) = 0.5\ I_p(t) = \log_2 \frac{1}{0.5} = 1$

不均匀硬币：

$q(h) = 0.2\ I_p(h) = \log_2 \frac{1}{0.2} = 2.32$

$q(t) = 0.8\ I_p(t) = \log_2 \frac{1}{0.8} = 0.32$

香农熵：一个概率分布所带有的平均信息量。这里的熵可以描述概率分布的不确定性。

香农熵定义：离散概率分布中每一个事件发生的概率乘信息量并求和

$H(p) = \sum p_i I_i^p = \sum p_i \log_2 \frac{1}{p_i} = -\sum p_i \log_2 p_i$

连续概率分布的情况-把求和变成积分

e.g. 均匀硬币

$H(p) = p(h) \times \log_2(\frac{1}{p(h)}) + p(t) \times log_2(\frac{1}{p(t)}) = 0.5 \times 1 + 0.5 \times 1 = 1$

e.g. 不均匀硬币（正：反=2：8）

$H(q) = q(h) \times \log_2 \frac{1}{q(h)} + q(t) \times \log_2 \frac{1}{q(t)} = 0.2 \times 2.32 + 0.8 \times 0.32 = 0.72$

对于概率分布来说：

- 概率密度函数均匀→随机变量不确定性更高→熵更大
- 概率密度函数聚拢→随机变量更确定→熵更小

交叉熵：给定估计概率q，对真实概率分布p的平均信息量的估计。

公式：$H(p,q) = \sum p_i I_i^q = \sum p_i \log_2 \frac{1}{q_i} = -\sum p_i \log_2\left(q_i\right)$

e.g. 一个硬币的ground truth的正反面概率分别为$p(h) = 0.5$，$p(t) = 0.5$。它的概率估计值$q(h) = 0.2$，$q(t) = 0.8$。

则$H(p,q) = p(h) \times \log_2 \frac{1}{q(h)} + p(t) \times \log_2 \frac{1}{q(t)} = 0.5 \times 2.32 + 0.5 \times 0.32 = 1.32$

若$q(h) = 0.4$，$q(t) = 0.6$，则$H(p,q) = 1.03$，熵值比上述情况小。

KL散度(Kullback-Leibler Divergence)：量化地衡量两个概率分布的区别的函数。两个分布越接近，则KL散度值越小。

定义：$KL_{\text{散度}} = \text{交叉熵} - \text{熵}$

$D(p||q) = H(p,q) - H(p) = \sum p_i I_i^q - \sum p_i I_i^p = \sum p_i \log_2 \frac{1}{q_i} - \sum p_i \log_2 \frac{1}{p_i} = \sum p_i \log_2 \frac{p_i}{q_i}$

一般的正态分布和标准正态分布的KL散度：

$KL(N(\mu,\sigma^2)||N(0,1)) = \frac{1}{2}(-\log\sigma^2 + \mu^2 + \sigma^2 - 1)$

KL散度的性质：

- $D(p||q) \geq 0$，仅两个分布相同时，值为0。
- $D(p||q) \neq D(q||p)$
- （梯度）$\nabla_\theta D(p||q_\theta) = \nabla_\theta H(p,q_\theta) - \nabla_\theta H(p) = \nabla_\theta H(p,q_\theta)$

总结：

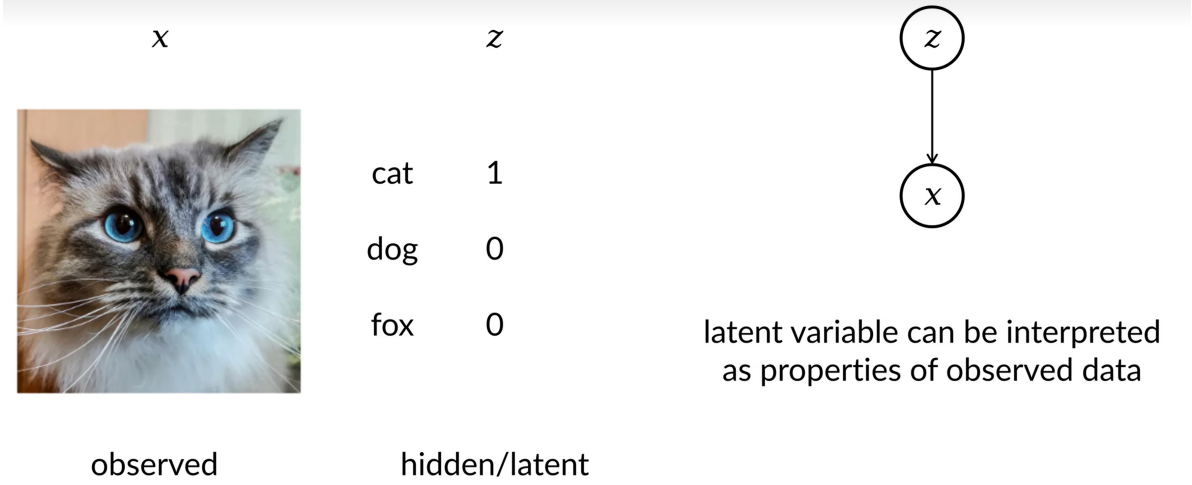
- 事件的信息量和事件发生的概率呈反比。
- 熵表述了一个概率分布的平均信息量。
- 交叉熵描述了从估计概率分布的角度对真实概率分布平均信息量的估计值。
- KL散度定量描述了两个概率分布的区别。
- KL散度对于概率模型而言是一个至关重要的概念，对推导模型的损失函数，比如交叉熵损失函数，有着重要的意义。

变分推理[2]

隐变量图模型：

x是隐变量，z是观测变量。

e.g.



变分推理的思想：我们通常感兴趣的是从观察到的数据中获得见解，从观察到的数据推断潜在变量的条件概率分布。

$x \sim p(x), z \sim p(z|x)$

变分推理希望通过x去推理z的分布。

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

（第一步变换是贝叶斯公式，第二步是$p(x)$在z上的积分）

由于$p(x)$难以解析计算，因此需要求解其近似解，用一系列的简单分布来近似这一复杂分布。这就是变分推断。

Variational Inference: Example (1)

$$z \backsim p(z) = \begin{cases} e^{-z}, z \geq 0 \\ 0 \quad , z < 0 \end{cases} = e^{-z} I(z \geq 0)$$

$$x \backsim p(x|z) = N(x, \mu = z, \sigma = 1) = \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(x-z)^2)}$$

$$p(x,z) = p(x|z)p(z) = \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(x-z)^2)} e^{-z} I(z \geq 0) \quad \text{(joint distribution)}$$

$$p(x) = \int_0^\infty p(x,z)\, dz = \underbrace{\int_0^\infty e^{-z} \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(x-z)^2)}\, dz}_{\text{the integral has no closed-form solution}}$$

thus the posterior has no closed-form solution

$p(x,z)$是联合概率分布，$p(x)$是边缘概率分布。

无法求解$p(x)$的解析解。

Variational Inference: Example (2)

$$p(z|x) \sim p(x,z) = p(x|z)p(z) = \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(x-z)^2)} e^{-z} I(z \geq 0)$$

$$= \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}z^2 + (x-1)z - \frac{1}{2}x^2)} I(z \geq 0)$$

$$= \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(z-(x-1))^2 + \frac{1}{2}(x-1)^2 - \frac{1}{2}x^2)} I(z \geq 0)$$

$$= \frac{1}{\sqrt{2\pi}} e^{(\frac{1}{2}(x-1)^2 - \frac{1}{2}x^2)} e^{(-\frac{1}{2}(z-(x-1))^2)} I(z \geq 0)$$

$$\sim \frac{1}{\sqrt{2\pi}} e^{(-\frac{1}{2}(z-(x-1))^2)} I(z \geq 0)$$

posterior proportional to a gaussian curve for *z* larger or equal to zero

z的后验概率正比于x, z联合分布。联合分布在$z \geq 0$的部分呈高斯分布。因此，z的后验分布正比于该正态分布。

Approximate Distribution: ELBO

$q_\theta(z) \approx p(z|x)$

$$D(q_\theta(z)||p(z|x)) = E_{z\sim q}[\log\frac{q_\theta(z)}{p(z|x)}] = E_{z\sim q}[\log q_\theta(z) - \log p(z|x)]$$
$$= E_{z\sim q}[\log q_\theta(z) - \log\frac{p(z,x)}{p(x)}]$$
$$= E_{z\sim q}[\log q_\theta(z) - \log p(z,x)] + \log p(x)$$

evidence lower bound (elbo) 证据下界

KL散度≥0

$$\log p(x) = E_{z\sim q}[\log p(z,x) - \log q_\theta(z)] + D(q_\theta(z)||p(z|x))$$

$$\log p(x) \geq E_{z\sim q}[\log p(z,x) - \log q_\theta(z)] \equiv \mathcal{L}_q$$

objective: minimizing kl divergence ⟶ maximizing elbo (variational optimization: optimizing over functions)

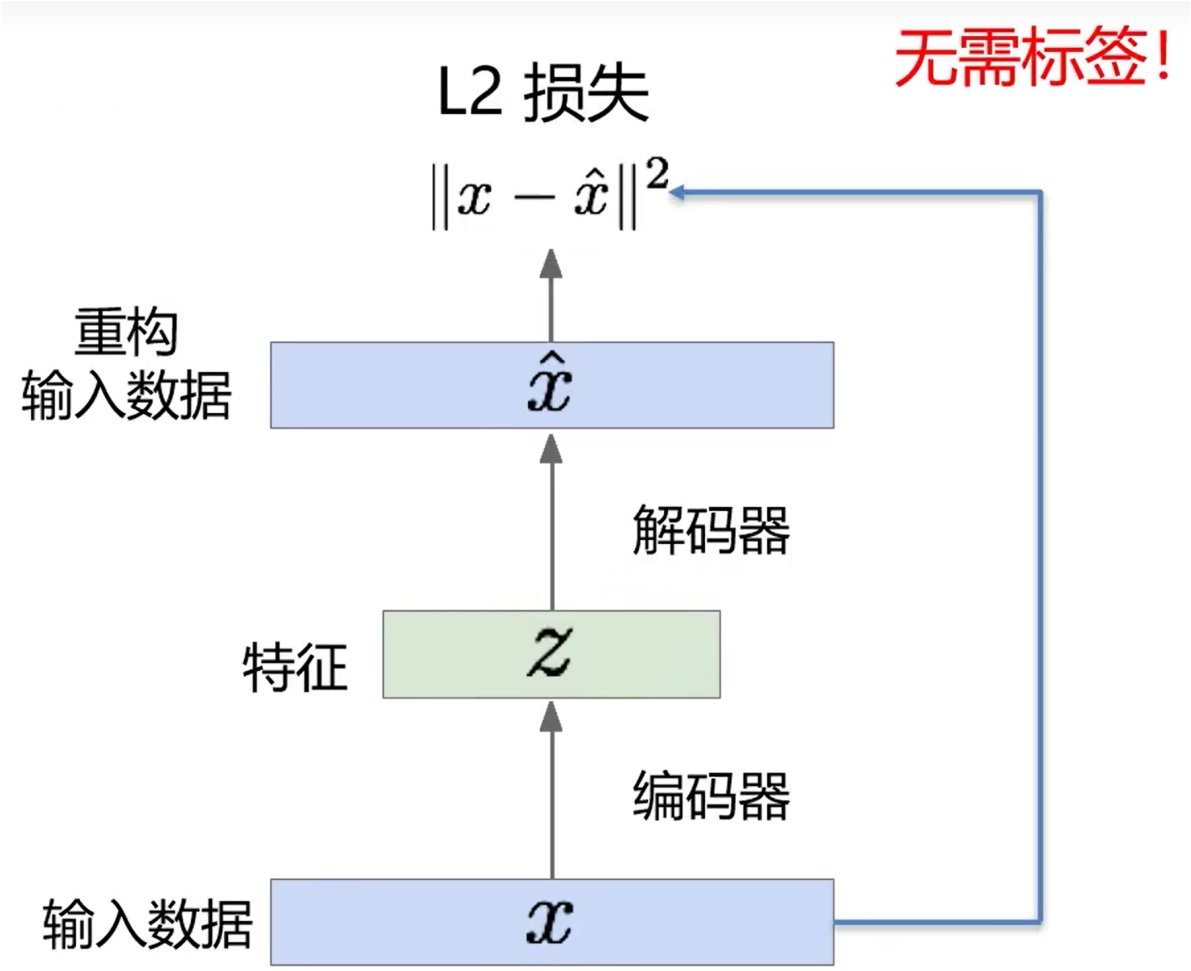
总结：

- 一个推理问题可以被构建为通过被观测变量推理相关隐变量后验概率密度分布的隐变量图模型，而这些隐变量通常可以被理解为被观测变量的一些属性，它们可以带来一些对被观测变量的见解和认知。
- 由于边缘概率通常包含针对隐变量的积分，所以真实的后验概率密度分布通常非常难以计算。
- 变分推理使用一组简单的、可以参数化的分布来近似真实的后验概率密度分布，从而将推理这一难以计算的问题变成一个可以计算的优化问题。
- KL散度衡量了近似概率分布和尝试逼近的真实后验概率分布之间的差异。
- 最小化KL散度等同于最大化证据下界(elbo)的过程。
- 证据下界是一个负数，而且它在随后的算法推导中非常重要。

变分自编码器(VAE)[4]

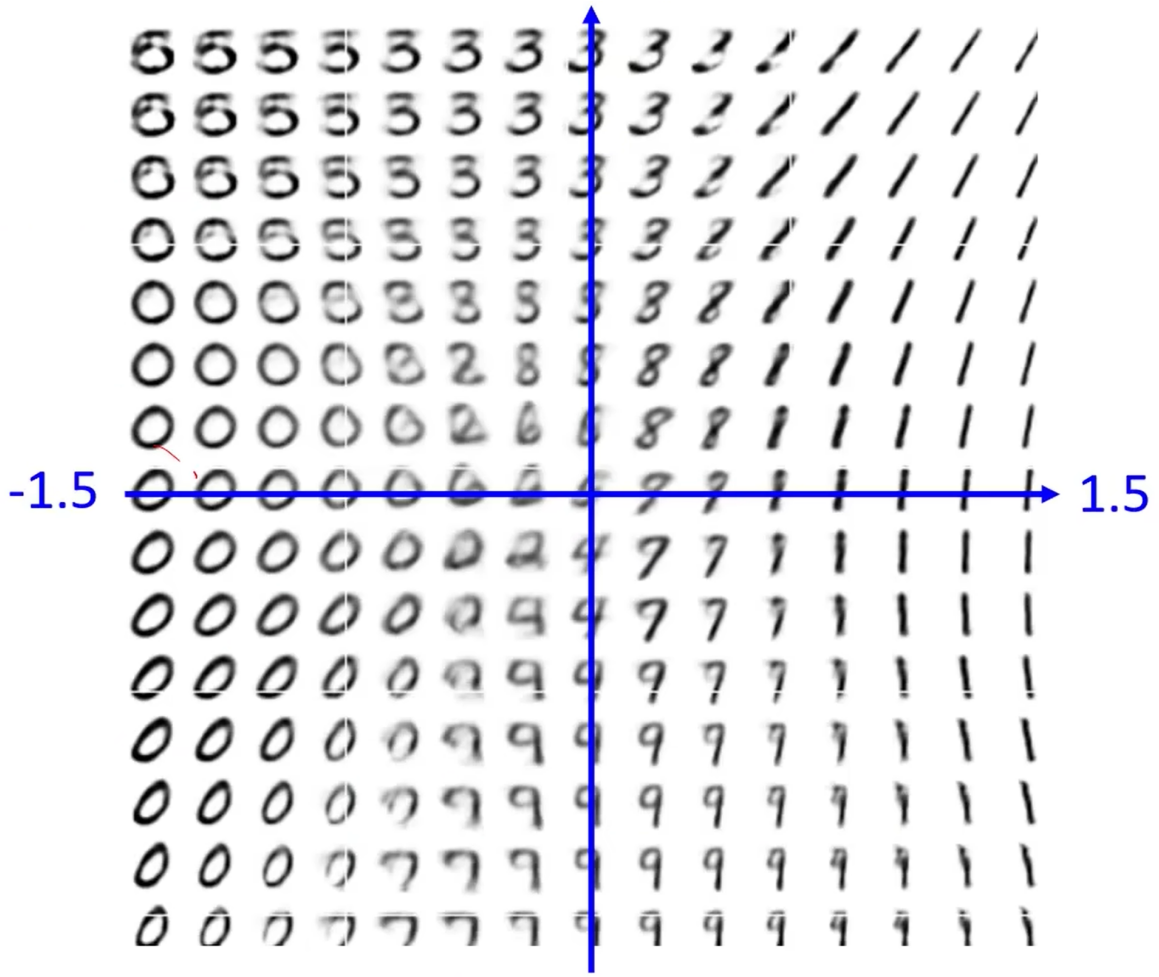
自编码器：无监督的特征学习，其目标是利用无标签数据找到一个有效的低维特征提取器。

AE:



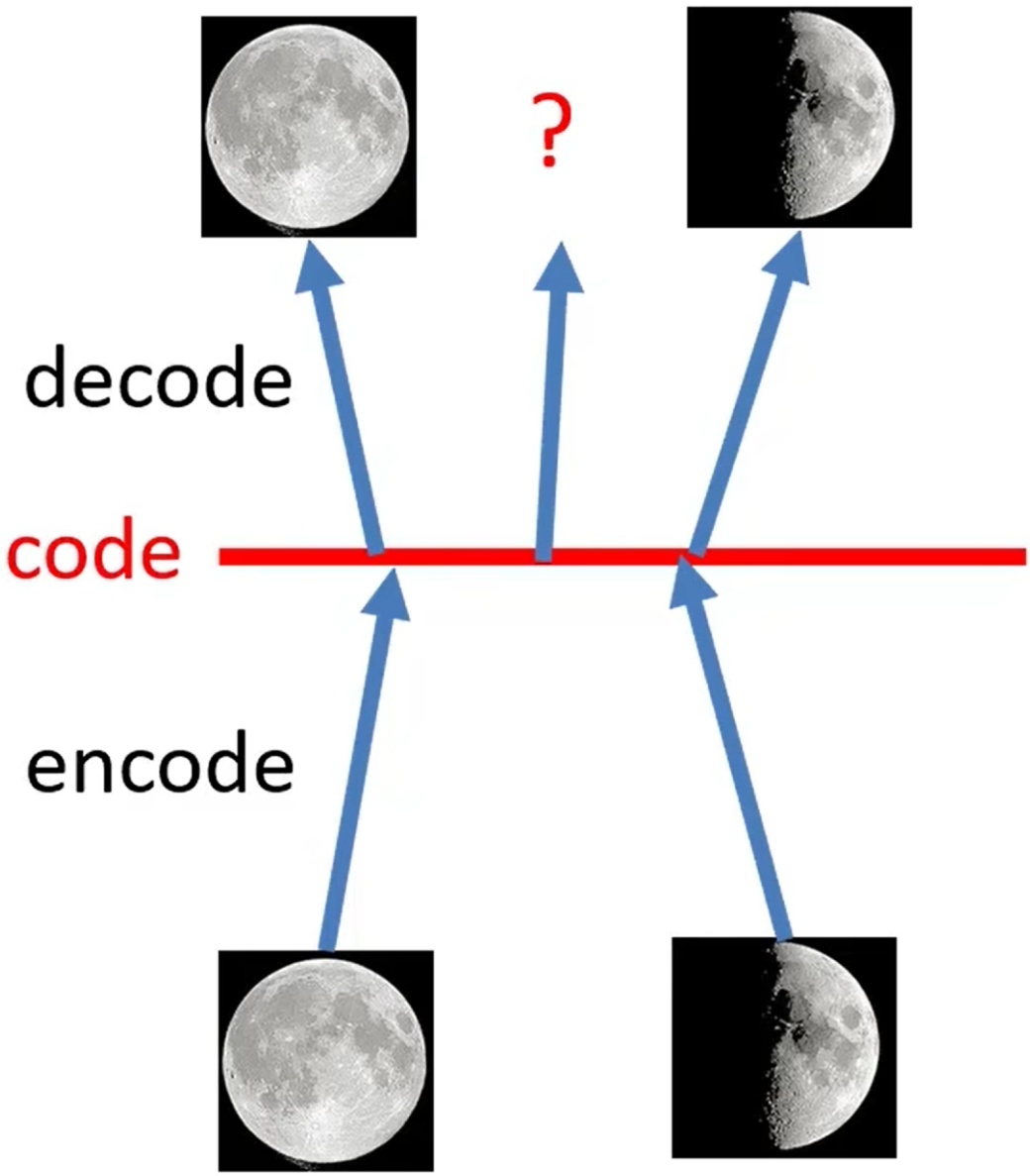
相当于一个网络，网络中间有编码器和解码器两个层。

可以用码空间生成数据：



为什么要在AE的基础上加上V？(Why VAE?)

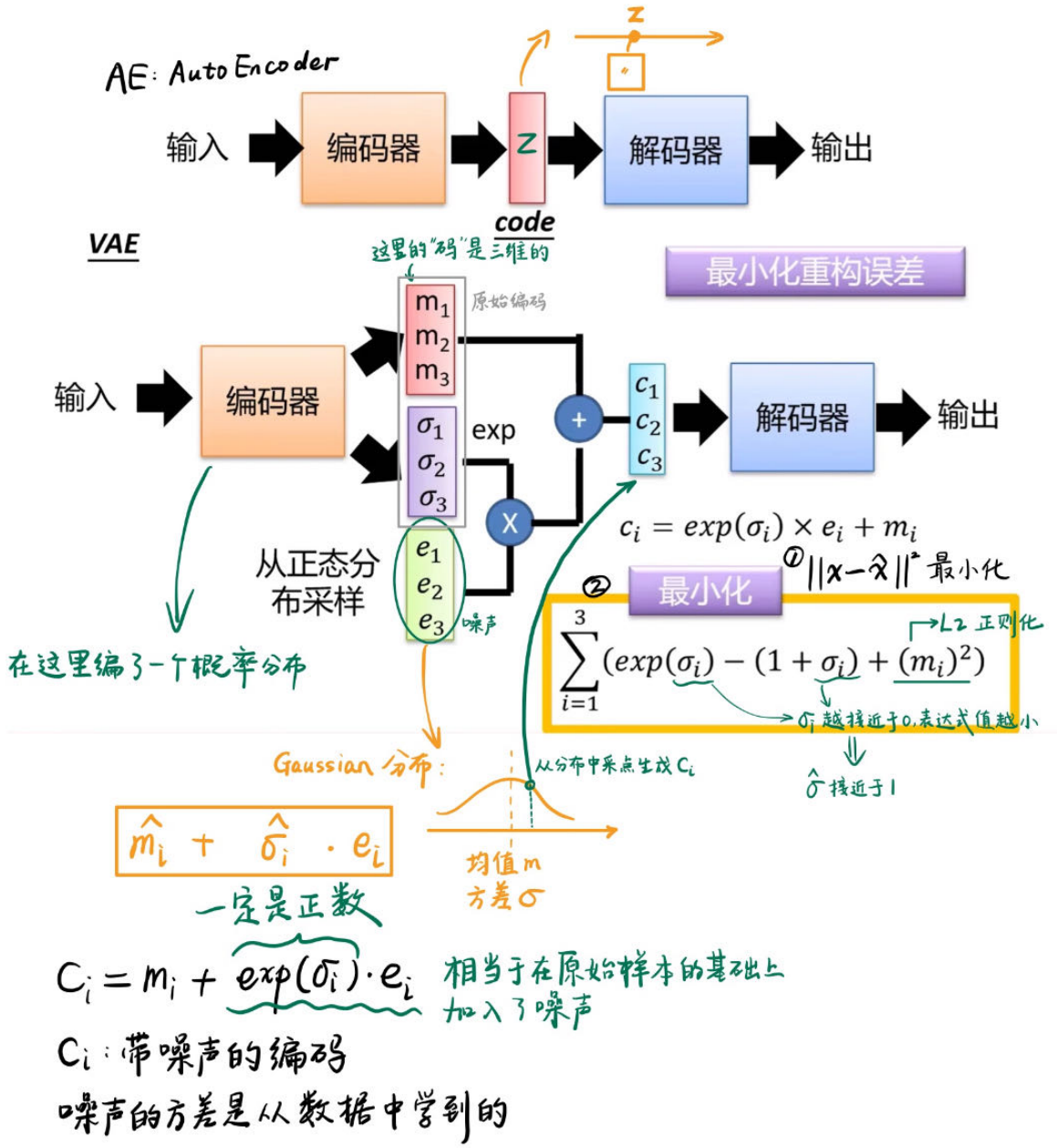
通俗理解：AutoEncoder中，对于离散输入的encode，decode后的输出也是离散的。也就是说，无法生成出来介于两者之间的图像。



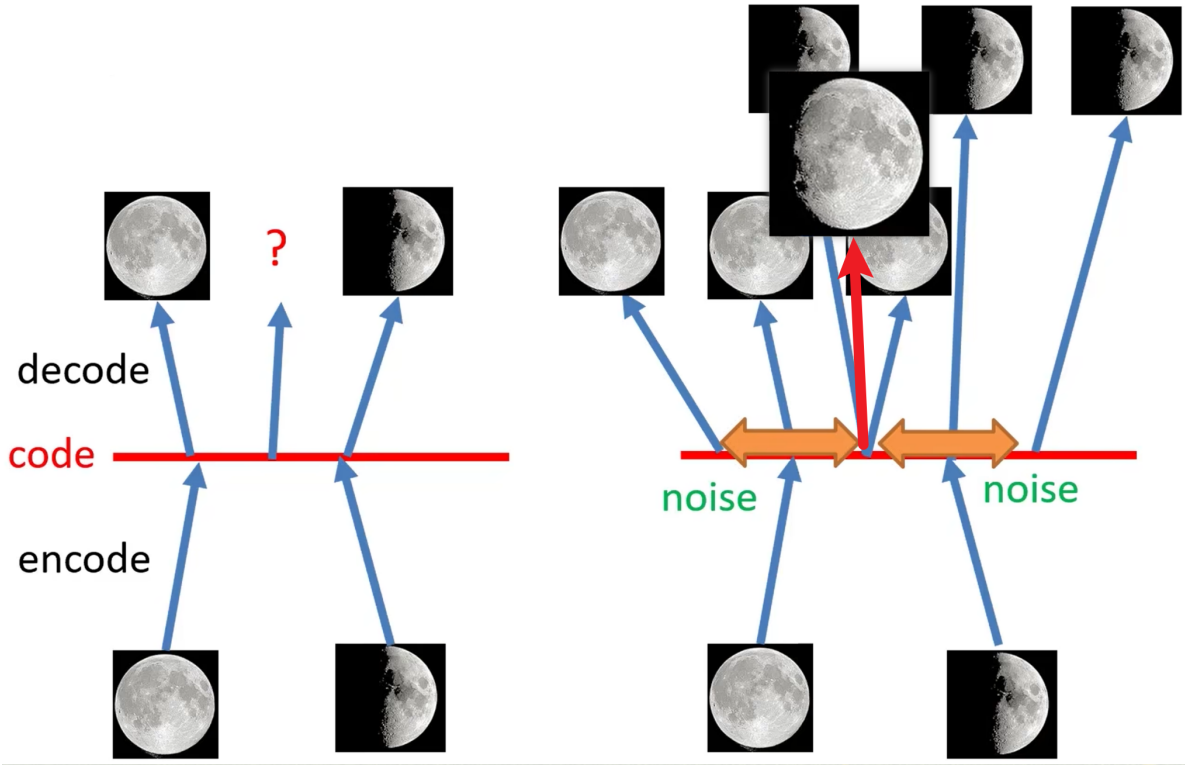
上图中，解码器无法生成一个介于满月和半个月亮之间的月亮图像。

VAE相较于AE，引入了“变分”，能用 正态分布 来替代原先的离散值，这样可以生成中间的图像。

变分自编码器：


AE: Auto Encoder
输入
编码器
z
解码器
输出
code
VAE
这里的"码"是三维的
最小化重构误差
原始编码
输入
编码器
exp
解码器
输出
从正态分布采样
噪声
在这里编了一个概率分布
$c_i = exp(\sigma_i) \times e_i + m_i$
① $||x-\hat{x}||^2$ 最小化
② 最小化
$\sum_{i=1}^{3}(exp(\sigma_i) - (1+\sigma_i) + (m_i)^2)$
L2 正则化
σ_i 越接近于0，表达式值越小
$\hat{\sigma}$ 接近于1
Gaussian 分布：
从分布中采点生成 c_i
$\hat{m}_i + \hat{\sigma}_i \cdot e_i$
均值 m
方差 σ
一定是正数
$C_i = m_i + exp(\sigma_i) \cdot e_i$
相当于在原始样本的基础上加入了噪声
C_i：带噪声的编码
噪声的方差是从数据中学到的

结果：



VAE代码实现[8]

1. 环境准备：Python编译器（PyCharm或Jupyter Notebook或Google Colab）、Pytorch、SciPy。

```
import torch
from torch import nn
import torch.nn.functional as F
from torch import optim
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
import matplotlib.pyplot as plt
```

2. 搭建VAE模型：

```
latent_dim = 2
input_dim = 28 * 28
inter_dim = 256

class VAE(nn.Module):
    def __init__(self, input_dim=input_dim, inter_dim=inter_dim,
latent_dim=latent_dim):
        super(VAE, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(input_dim, inter_dim),
            nn.ReLU(),
            nn.Linear(inter_dim, latent_dim * 2),
        )

        self.decoder =  nn.Sequential(
            nn.Linear(latent_dim, inter_dim),
```

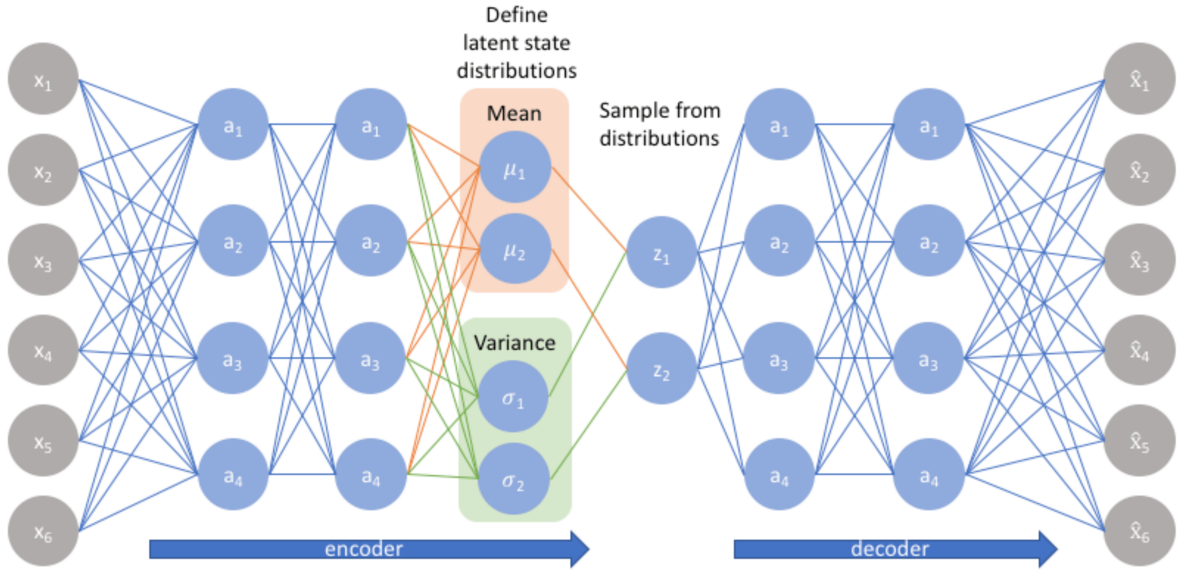
```
            nn.ReLU(),
            nn.Linear(inter_dim, input_dim),
            nn.Sigmoid(),
        )

    def reparameterize(self, mu, logvar):
        epsilon = torch.randn_like(mu)
        return mu + epsilon * torch.exp(logvar / 2)

    def forward(self, x):
        org_size = x.size() # x.size()返回包含x每一维大小的元组
        batch = org_size[0] # 提取x.size()第一维的大小
        x = x.view(batch, -1) # 改变x的形状，将张量展平（成一维），-1表示张量大小自动推断

        h = self.encoder(x)
        mu, logvar = h.chunk(2, dim=1) # 将张量h按维度dim=1切分成n=2个子张量
        z = self.reparameterize(mu, logvar)
        recon_x = self.decoder(z).view(size=org_size) # 使输入输出的维度保持一致

        return recon_x, mu, logvar
```

定义VAE类，实现初始化、重参数化、前向传播三个函数（继承自nn.Module）。

①初始化：

先初始化父类。

再定义编码器和解码器。（编码器和解码器的结构可以参考下图）



其中编码器的末尾不应添加ReLU层。若添加ReLU层，则解码器会因信息不足而崩溃。

这里的latent_dim后面乘2是为了输出均值和对数方差两个部分。

②重参数化：

为了解决生成模型中**采样操作不可微**以及无法通过反向传播来更新梯度的问题，"Reparameterization trick"提出将随机采样操作从网络中移动到一个确定性函数中。这个确定性函数通常是一个线性变换，将从标准高斯分布（均值为0，方差为1）中采样的随机噪声与潜在变量的均值和标准差相结合。这个确定性函数是可微分的，因此梯度可以在这个过程中传播。

`torch.rand_like()`返回一个张量，该张量由区间[0, 1)上均匀分布的随机数填充。(Returns a tensor with the same size as that is filled with random numbers from a uniform distribution on the interval input [0, 1))

e.g.

```
import torch
x = torch.randn(2, 3) # 是randn不是rand n可以理解为normal distribution
# 这里的输入x仅为生成y提供size的约束
# y服从均值为0方差为1的正态分布
y = torch.randn_like(x) # 是randn_like不是rand_like
print("x:")
print(x)
print("y:")
print(y)
```

打印结果：

```
x:
tensor([[-1.2325,  1.2024, -1.3687],
        [-0.9878, -0.3169,  2.3081]])
y:
tensor([[-0.4256, -0.7590, -0.2116],
        [ 1.0796, -0.0953,  0.0863]])
```

改为rand和rand_like：

```
import torch
x = torch.rand(5, 5)
# 这里的输入x仅为生成y提供size的约束
y = torch.rand_like(x) # 符合[0, 1)的均匀分布
print("x:")
print(x)
print("y:")
print(y)
```

打印结果：

```
x:
tensor([[0.7323, 0.4171, 0.7637, 0.5724, 0.1118],
        [0.3072, 0.5862, 0.1472, 0.3808, 0.7808],
        [0.6639, 0.3512, 0.4014, 0.3718, 0.4768],
        [0.9470, 0.6729, 0.2839, 0.8006, 0.4525],
        [0.2911, 0.9403, 0.4398, 0.6744, 0.6521]])
y:
tensor([[0.8802, 0.3079, 0.8996, 0.8264, 0.2596],
        [0.2414, 0.7230, 0.6033, 0.4801, 0.2473],
        [0.6322, 0.6492, 0.4419, 0.5045, 0.4613],
        [0.8297, 0.3991, 0.8906, 0.7500, 0.2619],
        [0.1669, 0.9790, 0.8143, 0.3800, 0.7385]])
```

所以代码中的`epsilon`表示一个与μ形状相同的、符合标准正态分布的噪声。返回值表示将符合标准正态分布的噪声调整到以μ为均值、以$e^{\frac{\log \sigma^2}{2}}$（即$\sigma$）为标准差的正态分布。

③前向传播：

x.view(batch, -1)将输入数据展平为二维向量，batch是批量大小。

h.chunk(2, dim=1)将h分割成均值和对数方差。

调用self.reparameterize(mu, logvar)生成潜变量z。

3. 定义重构损失和KL损失：

```
kl_loss = lambda mu, logvar: -0.5 * torch.sum(1 + logvar - mu.pow(2) -
logvar.exp())
recon_loss = lambda recon_x, x: F.binary_cross_entropy(recon_x, x,
size_average=False) # size_average: 控制是否对每个样本的损失进行平均
```

这里的KL散度符合：$KL(N(\mu, \sigma^2)||N(0,1)) = \frac{1}{2}(-\log\sigma^2 + \mu^2 + \sigma^2 - 1)$

因为MNIST是黑白二值图像, 所以的Decoder就可以用Sigmoid后的值当做灰度, 重构损失直接就用**BCE**了, 用MSE做重构损失亦可。但如果是三通道图像或者是灰度图像, 还是必须使用MSE做重构损失。

4. 训练前的准备工作

```
epochs = 100
batch_size = 128

transform = transforms.Compose([transforms.ToTensor()])
data_train = MNIST('MNIST_DATA/', train=True, download=False,
transform=transform) # 路径可自行更改
data_valid = MNIST('MNIST_DATA/', train=False, download=False,
transform=transform)

train_loader = DataLoader(data_train, batch_size=batch_size, shuffle=True,
num_workers=0)
test_loader = DataLoader(data_valid, batch_size=batch_size, shuffle=False,
num_workers=0)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = VAE(input_dim, inter_dim, latent_dim)
model.to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

5. 训练

```
best_loss = 1e9
best_epoch = 0

# 分别用来存储每个epoch的验证损失和训练损失
valid_losses = []
train_losses = []

for epoch in range(epochs):
    print(f"Epoch {epoch}:")
    model.train()
    train_loss = 0. # 累加每个batch的训练损失
    train_num = len(train_loader.dataset)

    for idx, (x, _) in enumerate(train_loader):
        batch = x.size(0) # 相当于获取batch_size
```

```
        x = x.to(device)
        recon_x, mu, logvar = model(x)
        recon = recon_loss(recon_x, x)
        kl = kl_loss(mu, logvar)

        loss = recon + kl
        train_loss += loss.item() # .item(): 用于将仅含一个元素的Tensor转换为其Python
数值类型
        loss = loss / batch

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if idx % 100 == 0:
            print(f"Training loss {loss: .3f} \t Recon {recon / batch: .3f} \t
KL {kl / batch: .3f} in Step {idx}")

    train_losses.append(train_loss / train_num)

    valid_loss = 0.
    valid_recon = 0.
    valid_kl = 0.
    valid_num = len(test_loader.dataset)
    model.eval()
    with torch.no_grad():
        for idx, (x, _) in enumerate(test_loader):
            x = x.to(device)
            recon_x, mu, logvar = model(x)
            recon = recon_loss(recon_x, x)
            kl = kl_loss(mu, logvar)
            loss = recon + kl
            valid_loss += loss.item()
            valid_kl += kl.item()
            valid_recon += recon.item()

        valid_losses.append(valid_loss / valid_num)

        print(f"Valid loss {valid_loss / valid_num: .3f} \t Recon {valid_recon /
valid_num: .3f} \t KL {valid_kl / valid_num: .3f} in epoch {epoch}")

        if valid_loss < best_loss: # 更新模型
            best_loss = valid_loss
            best_epoch = epoch

            # 保存模型
            torch.save(model.state_dict(), 'best_model_mnist')
            print("Model saved")
```

理解`for idx, (x, _) in enumerate(train_loader):`：

```python
import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader

train_dataset = torchvision.datasets.MNIST(root='...', train=True,
transform=transforms.ToTensor())

train_loader = DataLoader(dataset=train_dataset, batch_size=4, shuffle=True)

for idx, (x, y) in enumerate(train_loader):
    print(idx)
    print(x)
    print(y)
```

运行结果：

```
0
tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]]])
tensor([3, 6, 4, 1])
1
tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
```

可以看出，enumerate为train_loader中的每个batch提供了编号，所以idx打印出来就是当前batch的编号。对于每个batch内部，x是表示每张图片内像素信息的张量，y对应的是图像的标签。x和y可以分别理解为input和target。

```python
import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader

train_dataset = torchvision.datasets.MNIST(root='./minst', train=True,
transform=transforms.ToTensor())

train_loader = DataLoader(dataset=train_dataset, batch_size=4, shuffle=True)

print(len(train_dataset))
print(len(train_loader))
print(len(train_loader.dataset))
```

运行结果：

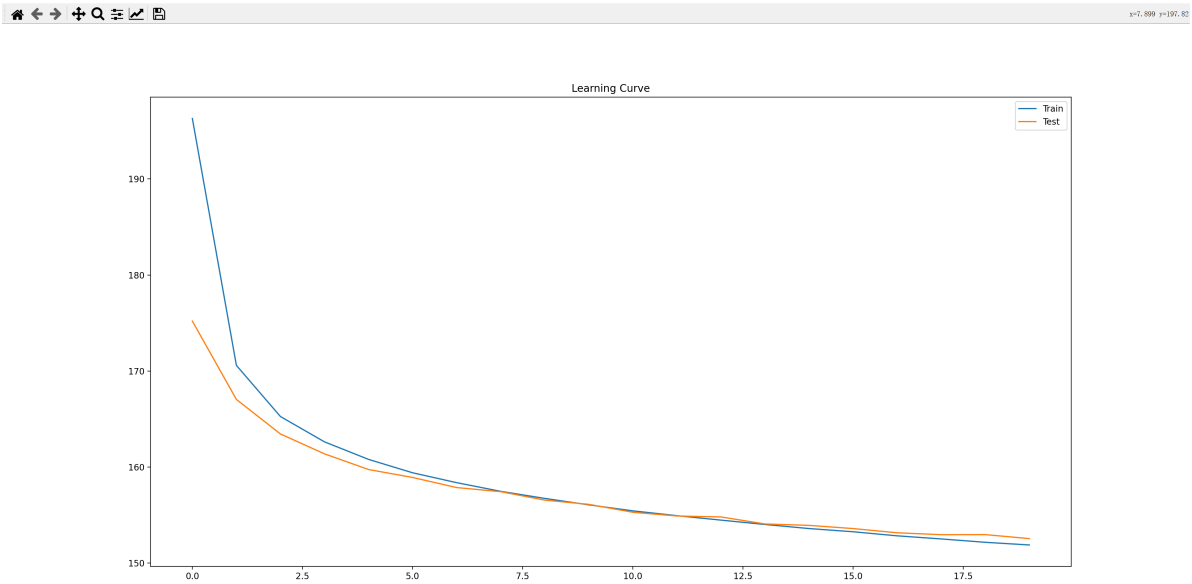
```
60000
15000
60000
```

train_dataset和train_loader.dataset表述的含义是相同的。

6. 绘制损失曲线（使用matplotlib）

```
plt.plot(train_losses, label='Train')
plt.plot(valid_losses, label='Valid')
plt.legend()
plt.title('Learning Curve');
```

epoch = 20:



7. 可视化生成结果

```
import numpy as np
from scipy.stats import norm

state = torch.load('best_model_mnist')
model = VAE()
model.load_state_dict(state)


n = 20
digit_size = 28

grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))


model.eval()
figure = np.zeros((digit_size * n, digit_size * n)) # 创建“画布”，小图像是28*28的，整体是由20*20个小图像组成
for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        t = [xi, yi] # t是当前点在潜在空间中的坐标
```

```
        z_sampled = torch.FloatTensor(t) # 将坐标转换为张量
        with torch.no_grad(): # 禁用梯度计算
            decode = model.decoder(z_sampled) # 通过解码器生成图像
            digit = decode.view((digit_size, digit_size)) # 调整生成的图像为28*28像
素
            figure[
                i * digit_size: (i + 1) * digit_size,
                j * digit_size: (j + 1) * digit_size
            ] = digit # 将生成的图像放到网格中的对应位置上

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap="Greys_r")
plt.xticks([])
plt.yticks([])
plt.axis('off');
```

norm.ppf(): 给定一个概率值（累积概率），返回该概率值在标准正态分布中对应的分位数。它可以用来找出在标准正态分布中，有多少比例的数据点落在某个值以下。如果给定累积概率0.95，norm.ppf()将返回在正态分布中，使得95%的数据点小于或等于这个值的分位数。

np.linspace(0.05, 0.95, n): 生成n个在[0.05, 0.95]区间均匀分布的点。其通过norm.ppf()转换为标准正态分布的值。

e.g.

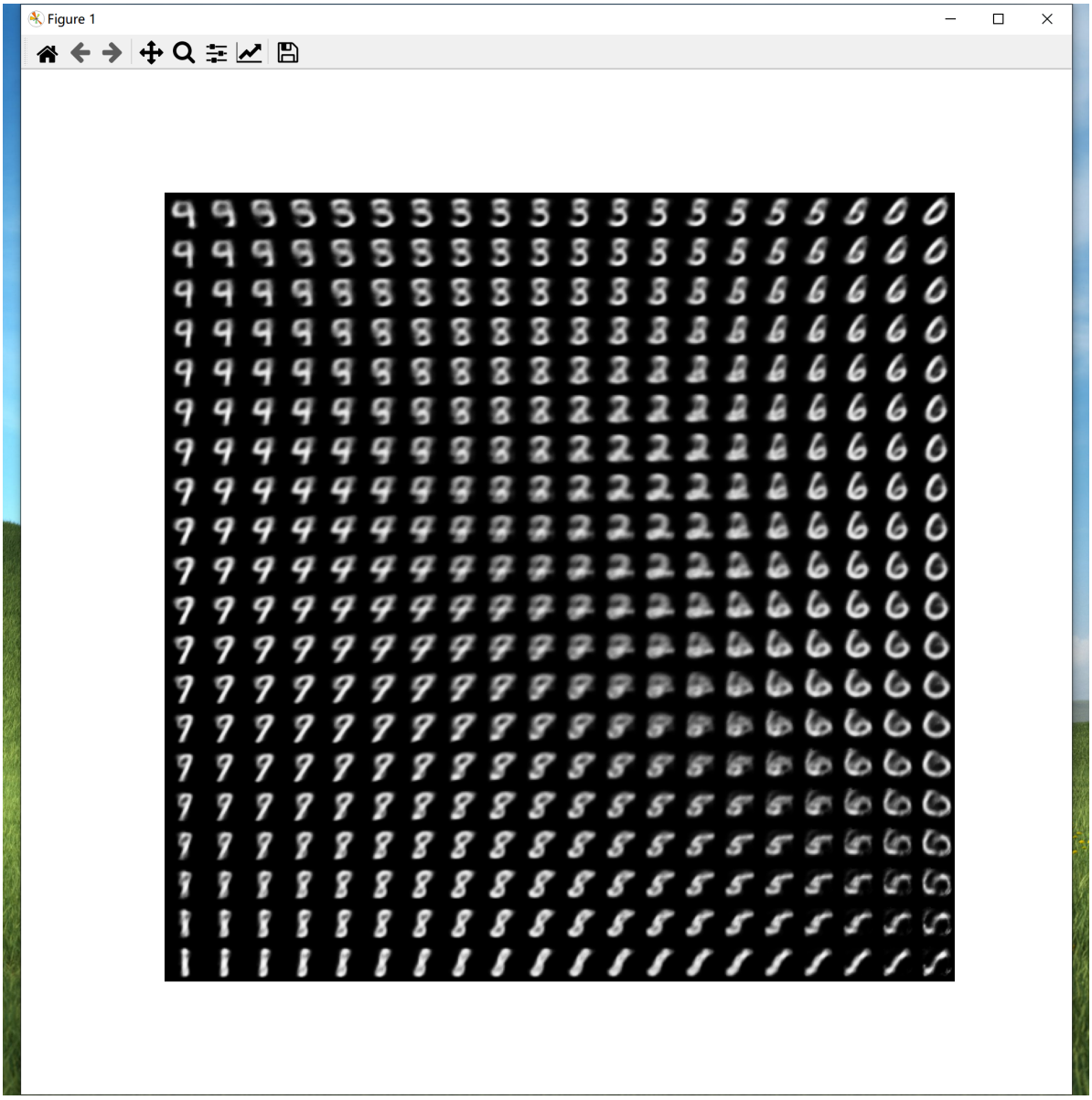
```
from scipy.stats import norm

p = 0.95
quantile = norm.ppf(p)

print(quantile)
print(np.linspace(0.05, 0.95, 20))
print(norm.ppf(np.linspace(0.05, 0.95, 20)))
```

运行结果：

```
1.6448536269514722
[0.05       0.09736842 0.14473684 0.19210526 0.23947368 0.28684211
 0.33421053 0.38157895 0.42894737 0.47631579 0.52368421 0.57105263
 0.61842105 0.66578947 0.71315789 0.76052632 0.80789474 0.85526316
 0.90263158 0.95      ]
[-1.64485363 -1.29669299 -1.05927692 -0.87016448 -0.7079966  -0.56263389
 -0.42831603 -0.30133652 -0.17905472 -0.05940243  0.05940243  0.17905472
  0.30133652  0.42831603  0.56263389  0.7079966   0.87016448  1.05927692
  1.29669299  1.64485363]
```

表示在标准正态分布中，大约95%的数据点小于或等于1.645（近似值，下同），大约9.737%的数据点小于或等于-1.297。

生成图像结果：



生成对抗网络(GAN)

- 判别器 (θ_d) 希望**最大化目标函数**使得D(x)接近于1（真实样本），而 D(G(z)) 接近于0 （假样本）
- 生成器 (θ_g) 希望**最小化目标函数**使得 D(G(z)) 尽量接近于1 ，即希望判别器认为生成器产生的图像 G(z) 为真实图片。

打分值介于0到1之间 { 0：假样本 / 1：真实样本

Minimax 目标函数:

$$\min_{\theta_g}\max_{\theta_d}\left[\mathbb{E}_{x\sim p_{data}}\log D_{\theta_d}(x)+\mathbb{E}_{z\sim p(z)}\log(1-D_{\theta_d}(G_{\theta_g}(z)))\right]$$

判别器对真实样本打分　判别器对生成样本G(z)打分　真实数据　"噪声"

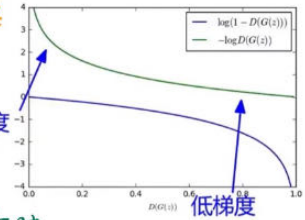
交替完成:

1. **Gradient ascent** on discriminator 优化判别器

$$\max_{\theta_d}\left[\mathbb{E}_{x\sim p_{data}}\log D_{\theta_d}(x)+\mathbb{E}_{z\sim p(z)}\log(1-D_{\theta_d}(G_{\theta_g}(z)))\right]$$

2. **Instead: Gradient ascent** on generator, different objective 优化生成器

$$\max_{\theta_g}\mathbb{E}_{z\sim p(z)}\log(D_{\theta_d}(G_{\theta_g}(z)))$$

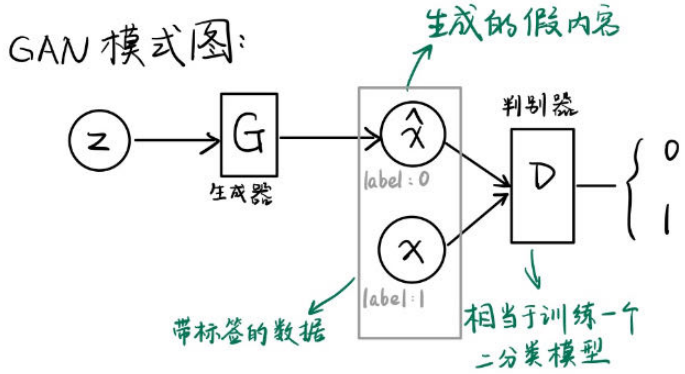


原本是: $\min_{\theta_g} \mathbb{E}_{z\sim p(z)} \log(1-D_{\theta_d}(G_{\theta_g}(z)))$ 但带来的问题是: 高梯度

真实中很好用，大家几乎都这么用!

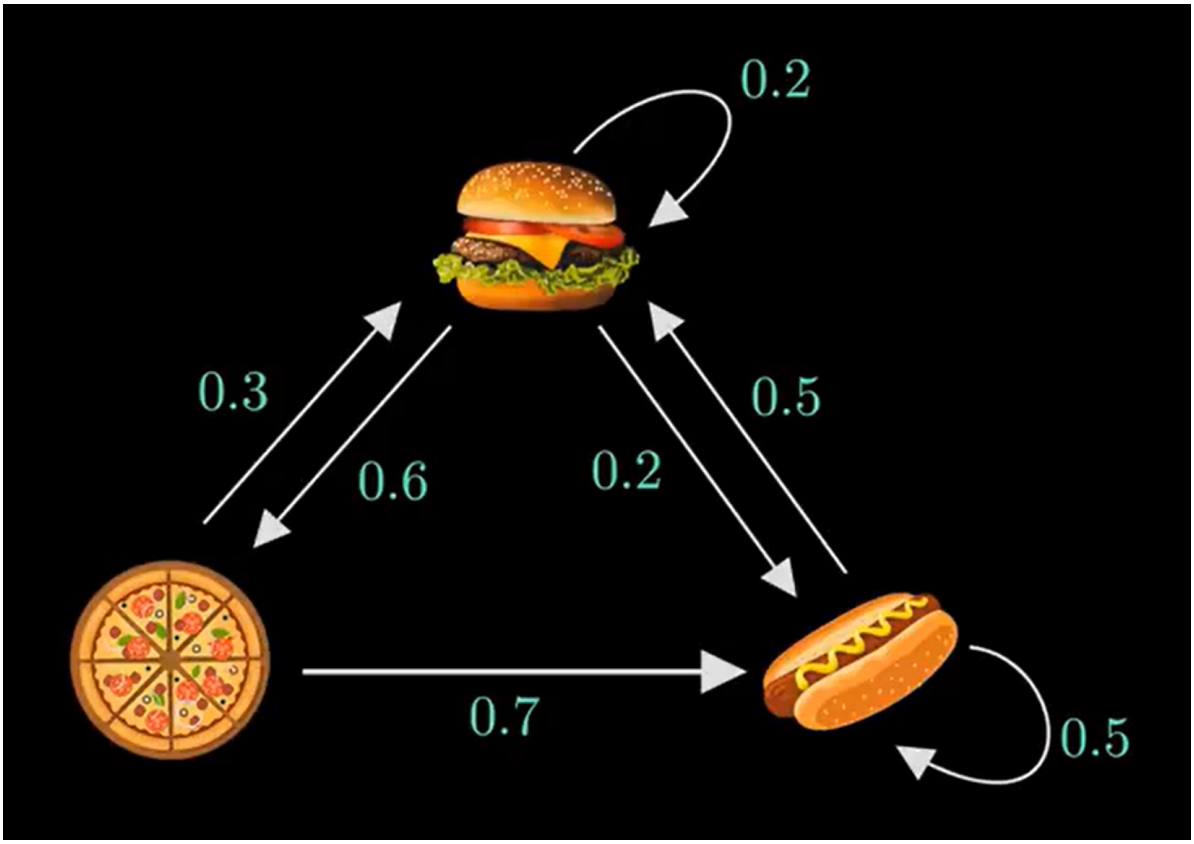
前期梯度小，后期梯度大 → 学习速度慢 / 无需大幅更新时，低梯度更合适

低梯度



马尔可夫链[6]

e.g. 一个餐厅每天有可能供应三种食物：汉堡、披萨、热狗。箭头A→B表示在前一天供应A时，第二天供应B的概率。



这里对箭头上的概率可以表示如下：

$P(X_{n+1} = x|X_n = x_n)$

马尔可夫链的精髓——马尔可夫性质：可以忽略前面除$n-1$步以外的步骤对结果的影响。

e.g. $P(X_4 = \text{热狗}|X_3 = \text{披萨}) = 0.7$，与$X_1$和$X_2$的取值无关。

经过∞天后，餐厅供应各种食物的概率会趋近于“稳态”。

求解“稳态”：

①写邻接矩阵：

$$A = \begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{bmatrix}$$

②构建π向量（代表状态的概率）：

（假设开始时，处于披萨日）

$$\pi_0 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$\pi_0 A = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.3 & 0 & 0.7 \end{bmatrix} = \pi_1$$

继续将π_1, π_2等带入。此时，如果存在一个稳态，那么在某个点后，输出的行向量应该与输入的行向量完全相同。用π表示。

$$\pi A = \pi$$

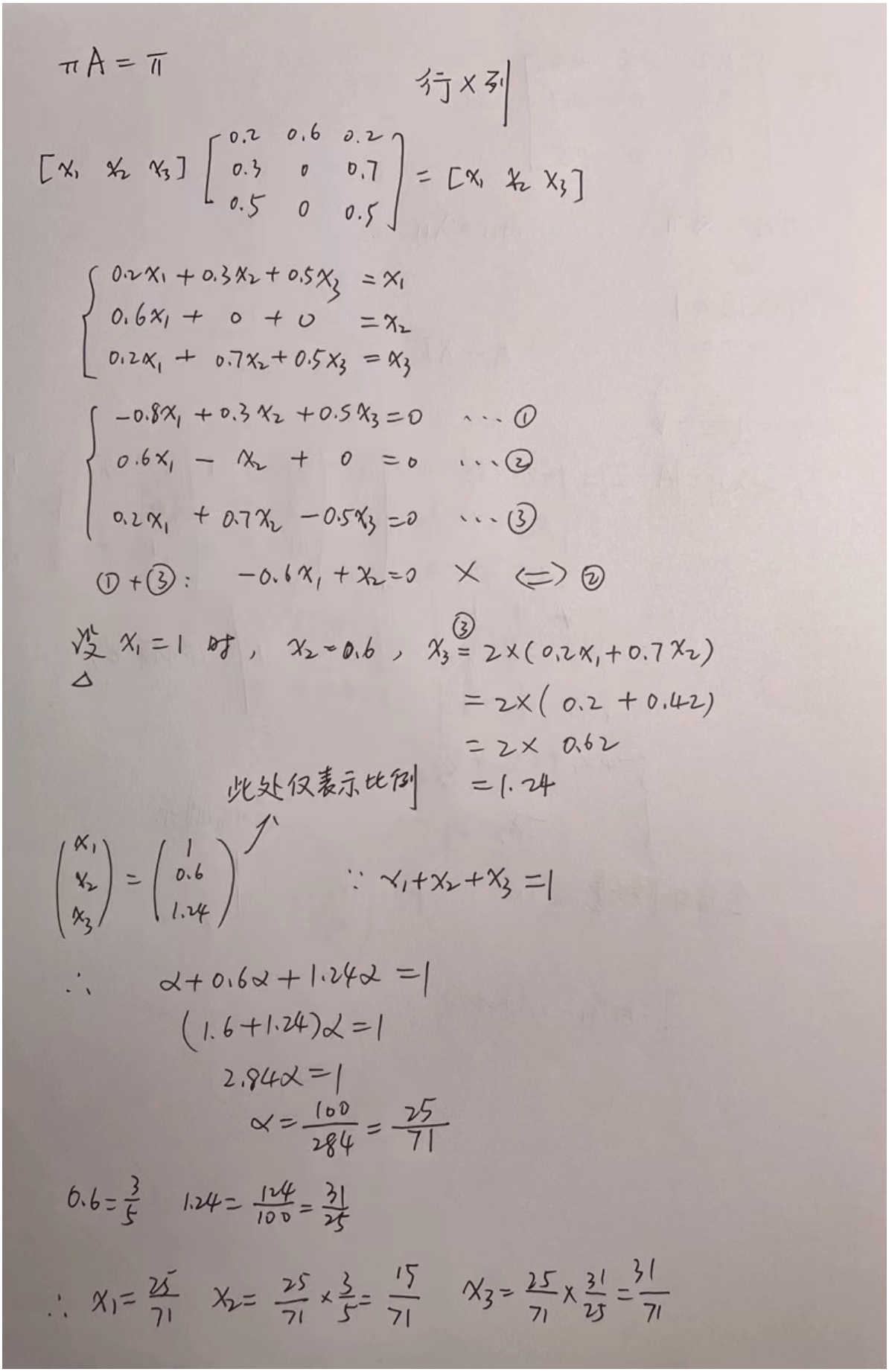
相当于π是矩阵A的左特征向量，特征值等于1。（参考$Av = \lambda v$理解）

同时，要满足$\sum \pi[i] = 1$

③求解：

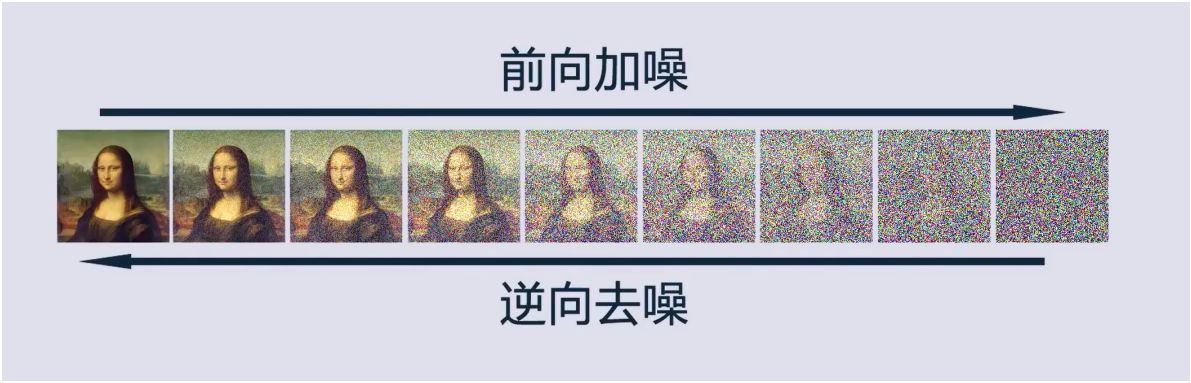
$\pi = \left[\frac{25}{71} \quad \frac{15}{71} \quad \frac{31}{71} \right]$

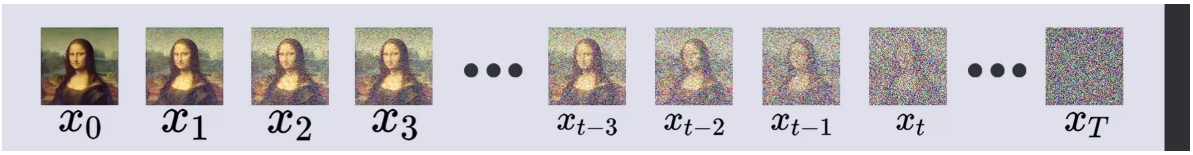
计算过程：



④判断是否有多个稳态：看是否存在不止一个特征值等于1的特征向量。

扩散模型(Diffusion Model)[7]





前向过程

后一时刻图像与前一时刻图像的关系：$x_t = \sqrt{1-\alpha_t} \times \epsilon_t + \sqrt{\alpha_t} \times x_{t-1}$

$x_{t-1} = \sqrt{1-\alpha_{t-1}} \times \epsilon_{t-1} + \sqrt{\alpha_{t-1}} \times x_{t-2}$

x_{t-2}与x_t的关系：$x_t = \sqrt{a_t(1-a_{t-1})} \times \epsilon_{t-1} + \sqrt{1-a_t} \times \epsilon_t + \sqrt{a_t a_{t-1}} \times x_{t-2}$

叠加概率分布（正态分布）：$N(\mu_1, \sigma_1^2) + N(\mu_2, \sigma_2^2) = N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$

因为ϵ_{t-1}和ϵ_t均为服从标准正态分布的随机数，所以$\sqrt{a_t(1-a_{t-1})} \times \epsilon_{t-1} \sim N(0, \alpha_t - \alpha_t\alpha_{t-1})$，$\sqrt{1-a_t} \times \epsilon_t \sim N(0, 1-\alpha_t)$，二者叠加后，服从分布$N(0, 1-\alpha_t\alpha_{t-1})$。因此，$x_t$可表示为：

$x_t = \sqrt{1-\alpha_t\alpha_{t-1}} \times \epsilon + \sqrt{\alpha_t\alpha_{t-1}} \times x_{t-2}$（使用**“重参数化技巧”**）

以此类推，可以得到x_t与x_{t-k}的关系式：

$x_t = \sqrt{1-\alpha_t\alpha_{t-1}\alpha_{t-2}\dots\alpha_{t-(k-2)}\alpha_{t-(k-1)}}\epsilon + \sqrt{\alpha_t\alpha_{t-1}\alpha_{t-2}\dots\alpha_{t-(k-2)}\alpha_{t-(k-1)}}x_{t-k}$

当$k=0$时，可以得到x_t和x_0的关系式：

$x_t = \sqrt{1-\alpha_t\alpha_{t-1}\alpha_{t-2}\dots\alpha_2\alpha_1} \times \epsilon + \sqrt{\alpha_t\alpha_{t-1}\alpha_{t-2}\dots\alpha_2\alpha_1} \times x_0$

令$\bar{\alpha_t} = \alpha_t\alpha_{t-1}\alpha_{t-2}\dots\alpha_2\alpha_1$，则：

$x_t = \sqrt{1-\bar{\alpha_t}} \times \epsilon + \sqrt{\bar{\alpha_t}} \times x_0$

反向过程

由贝叶斯定理，$P(x_{t-1}|x_t) = \frac{P(x_t|x_{t-1})P(x_{t-1})}{P(x_t)} = \frac{P(x_t|x_{t-1},x_0)P(x_{t-1}|x_0)}{P(x_t|x_0)}$

由于$x_t = \sqrt{1-\alpha_t} \times \epsilon_t + \sqrt{\alpha_t} \times x_{t-1}$，所以$P(x_t|x_{t-1}, x_0) \sim N(\sqrt{\alpha_t}x_{t-1}, 1-\alpha_t)$。（$\epsilon_t \sim N(0,1)$）

由于$x_t = \sqrt{1-\bar{\alpha_t}} \times \epsilon + \sqrt{\bar{\alpha_t}} \times x_0$，所以$P(x_t|x_0) \sim N(\sqrt{\bar{\alpha_t}}x_0, 1-\bar{\alpha_t})$

$P(x_{t-1}|x_0) \sim N(\sqrt{\bar{\alpha_{t-1}}}x_0, 1-\bar{\alpha_{t-1}})$

将上述的$N(\mu, \sigma^2)$带入正态分布的概率密度函数$f(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$中：

$$P(x_{t-1}|x_t,x_0)=\frac{1}{\sqrt{2\pi}(\frac{\sqrt{1-\alpha_t}\sqrt{1-\bar{\alpha}_{t-1}}}{\sqrt{1-\bar{\alpha}_t}})}e^{\frac{[-\frac{(x_{t-1}-(\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t+\frac{\sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)}{1-\bar{\alpha}_t}x_0))}{1-\bar{\alpha}_t}]}{2(\frac{\sqrt{1-\alpha_t}\sqrt{1-\bar{\alpha}_{t-1}}}{\sqrt{1-\bar{\alpha}_t}})^2}}$$

可以得到：

$$P(x_{t-1}|x_t,x_0)\sim N(\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t+\frac{\sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)}{1-\bar{\alpha}_t}x_0,(\frac{\sqrt{1-\alpha_t}\sqrt{1-\bar{\alpha}_{t-1}}}{\sqrt{1-\bar{\alpha}_t}})^2)$$

由$x_t=\sqrt{1-\bar{\alpha}_t}\times\epsilon+\sqrt{\bar{\alpha}_t}\times x_0$，可以将$x_0$用含有$x_t$的式子进行表达。

所以可以通过训练神经网络来模拟这个过程。

扩散模型代码实现[9]

```
import torch
import torchvision
import matplotlib.pyplot as plt

def show_images(datset, num_samples=20, cols=4):
    """ Plots some samples from the dataset """
    plt.figure(figsize=(15,15))
    for i, img in enumerate(data):
        if i == num_samples:
            break
        plt.subplot(int(num_samples/cols) + 1, cols, i + 1)
        plt.imshow(img[0])

data = torchvision.datasets.StanfordCars(root=".", download=True)
show_images(data)
```

```
import torch.nn.functional as F

# 返回一个一维的张量，这个张量包含了从start到end，分成steps个线段得到的向量。
# e.g. torch.linspace(3, 10, 5)
# tensor([3.0000, 4.7500, 6.5000, 8.2500, 10.0000])
def linear_beta_schedule(timesteps, start=0.0001, end=0.02):
    return torch.linspace(start, end, timesteps)

# *
def get_index_from_list(vals, t, x_shape):
    """
    Returns a specific index t of a passed list of values vals
    while considering the batch dimension.
    """
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())
    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

# 前向扩散
# 给定输入图像x_0和timestep t，返回图像的加噪版本
def forward_diffusion_sample(x_0, t, device="cpu"):
```

```
    """
    Takes an image and a timestep as input and
    returns the noisy version of it
    """
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t,
x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )
    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device),
noise.to(device) # “\”表示续行符


# Define beta schedule
T = 300
betas = linear_beta_schedule(timesteps=T)

# Pre-calculate different terms for closed form
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

```
from torchvision import transforms
from torch.utils.data import DataLoader
import numpy as np

IMG_SIZE = 64
BATCH_SIZE = 128

def load_transformed_dataset():
    data_transforms = [
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(), # Scales data into [0,1]
        transforms.Lambda(lambda t: (t * 2) - 1) # Scale between [-1, 1]
    ]
    data_transform = transforms.Compose(data_transforms)

    train = torchvision.datasets.StanfordCars(root=".", download=True,
                                        transform=data_transform)

    test = torchvision.datasets.StanfordCars(root=".", download=True,
                                        transform=data_transform, split='test')
    return torch.utils.data.ConcatDataset([train, test])
def show_tensor_image(image):
    reverse_transforms = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: t.permute(1, 2, 0)), # CHW to HWC
```

```
        transforms.Lambda(lambda t: t * 255.),
        transforms.Lambda(lambda t: t.numpy().astype(np.uint8)),
        transforms.ToPILImage(),
    ])

    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]
    plt.imshow(reverse_transforms(image))

data = load_transformed_dataset()
dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True,
drop_last=True)
```

```
# Simulate forward diffusion
image = next(iter(dataloader))[0]

plt.figure(figsize=(15,15))
plt.axis('off')
num_images = 10
stepsize = int(T/num_images)

for idx in range(0, T, stepsize):
    t = torch.Tensor([idx]).type(torch.int64)
    plt.subplot(1, num_images+1, int(idx/stepsize) + 1)
    img, noise = forward_diffusion_sample(image, t)
    show_tensor_image(img)
```

```
from torch import nn
import math


class Block(nn.Module):
    def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
        super().__init__()
        self.time_mlp =  nn.Linear(time_emb_dim, out_ch)
        if up:
            self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
        self.bnorm1 = nn.BatchNorm2d(out_ch)
        self.bnorm2 = nn.BatchNorm2d(out_ch)
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
```

```
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        return self.transform(h)


class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -
embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        # TODO: Double check the ordering here
        return embeddings


class SimpleUnet(nn.Module):
    """
    A simplified variant of the Unet architecture.
    """
    def __init__(self):
        super().__init__()
        image_channels = 3
        down_channels = (64, 128, 256, 512, 1024)
        up_channels = (1024, 512, 256, 128, 64)
        out_dim = 3
        time_emb_dim = 32

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],
\
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])
```

```
        # Edit: Corrected a bug found by Jakub C (see YouTube comment)
        self.output = nn.Conv2d(up_channels[-1], out_dim, 1)

    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)

model = SimpleUnet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
print(model)
```

```
def get_loss(model, x_0, t):
    x_noisy, noise = forward_diffusion_sample(x_0, t, device)
    noise_pred = model(x_noisy, t)
    return F.l1_loss(noise, noise_pred)
```

```
@torch.no_grad()
def sample_timestep(x, t):
    """
    Calls the model to predict the noise in the image and returns
    the denoised image.
    Applies noise to this image, if we are not in the last step yet.
    """
    betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x.shape
    )
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

    # Call model (current image - noise prediction)
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )
    posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)

    if t == 0:
        # As pointed out by Luis Pereira (see YouTube comment)
        # The t's are offset from the t's in the paper
```

```
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise

@torch.no_grad()
def sample_plot_image():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
        img = sample_timestep(img, t)
        # Edit: This is to maintain the natural range of the distribution
        img = torch.clamp(img, -1.0, 1.0)
        if i % stepsize == 0:
            plt.subplot(1, num_images, int(i/stepsize)+1)
            show_tensor_image(img.detach().cpu())
    plt.show()
```

```
from torch.optim import Adam

device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
optimizer = Adam(model.parameters(), lr=0.001)
epochs = 100 # Try more!

for epoch in range(epochs):
    for step, batch in enumerate(dataloader):
      optimizer.zero_grad()

      t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
      loss = get_loss(model, batch[0], t)
      loss.backward()
      optimizer.step()

      if epoch % 5 == 0 and step == 0:
        print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
        sample_plot_image()
```

Python类型注解

`->`符号用于表示函数的返回类型。

e.g.

```
def __init__(self, in_channels: int, out_channels: int, is_res: bool=False) ->
None:
    ...
```

`-> None` 表示这个 `__init__` 方法的返回类型是 `None`。

参考资料

[1] 【10分钟】了解香农熵，交叉熵和KL散度_哔哩哔哩_bilibili

[2] 【15分钟】了解变分推理_哔哩哔哩_bilibili

[3] 条件概率、联合概率和贝叶斯公式-CSDN博客

[4] 【生成模型VAE】十分钟带你了解变分自编码器及搭建VQ-VAE模型（Pytorch代码）！简单易懂！—GAN/机器学习/监督学习_哔哩哔哩_bilibili

[5] 一文彻底读懂【极大似然估计】-CSDN博客

[6] 超简单解释什么是马尔可夫链！_哔哩哔哩_bilibili

[7] 大白话AI | 图像生成模型DDPM | 扩散模型 | 生成模型 | 概率扩散去噪生成模型_哔哩哔哩_bilibili

[8] Pytorch实现: VAE | DaNing的博客 (adaning.github.io), VAE.ipynb - Colab (google.com)

[9] diffusion_model.ipynb - Colab (google.com)