湖南大学

HUNAN UNIVERSITY

《人工智能》
课程实验报告

题　　目：	diffusion扩散模型学习报告
姓　　名：	杨铭
学　　号：	202001130329
专　　业：	计科2002
上课时间：	2023-秋
授课教师：	许莹

2023 年 05月 22 日

目录

一、实验目的

通过使用深度学习工具与框架，进一步了解深度学习主要步骤和代码实现。并通过结合应用背景，将深度学习框架用到实际生活中。

- 了解深度学习的基本原理
- 能够使用深度学习开源工具
- 应用深度学习算法求解实际问题

二、实验要求

- 解释深度学习的原理；
- 采用深度学习框架完成课程综合实验，并对实验结果分析；
- 回答思考题；

三、实验平台和工具

平台	说明
Huawei Cloud （ModelArts组件）	服务器提供GPU和高频CPU
Personnal Computer	实验代码书写
Jupyter Notebook	用于运行和调试并展现diffusion生成结果
晟腾910服务器	程序运行

表1

四、实验内容和步骤

（一）项目实战介绍

1. Diffusion概述

伴随着人工智能在图像生成，文本生成以及多模态生成等领域的技术的不断积累，如：生成对抗网络（GAN）、变为分自动编码器（VAE）、正态流模型、自回归模型（AR）等。

本实验将目光投注到近年来最为火热的模型之一——扩散模型（Diffusion Model）

扩散模型的成功并不是横空出世的，起始早在15年就有人提出了类似的思想，最终在20年有了我们熟知的“Denoising Diffusion Probabilistic Models”。

本项目的目的是阅读这一模型框架的论文，了解基本的原理思想，结合课程学习的深度神经网络，观察并实践论文代码的实现，进一步理解深度神经网络在扩散模型，乃至图像生成领域的广泛应用。

2. Diffusion应用

Diffusion最大的应用就是可以进行AI作画，通过扩散的效果实现，多幅图片融合的效果，达到根据图片生成图片、文字生成图片的效果。下图显示的是[Stable Diffusion Online (stablediffusionweb.com)](stablediffusionweb.com)根据关键词生成的图像。

上图是一个文本生成图像的人工智能网站的演示效果。如图，当输入计算机数额相关的文字，生成了以上的图片。图像生成对于计算机视觉有着重要的应用。

3. 模型训练和采样的算法流程

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

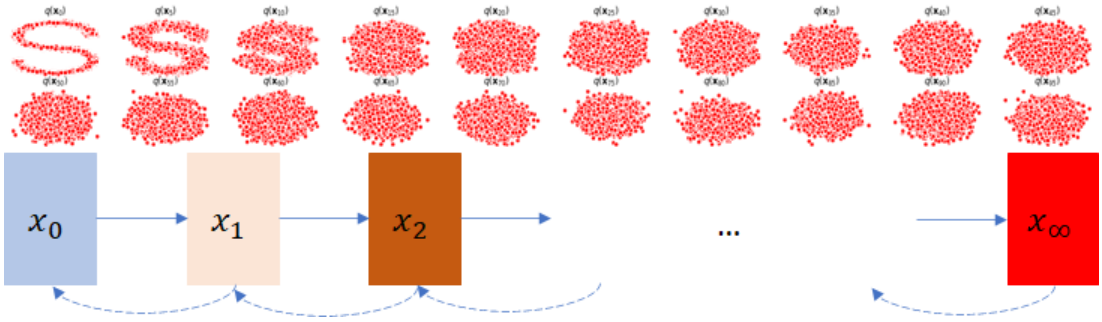
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

上图是论文的伪代码，扩散模型一共分为两个过程；Training即正向扩散过程、Sampling即反向采样过程。上片段是两个过程的伪代码。

（二）正向Training

1. 前向噪声扩散公式推导

diffusion模型的前向古城是向原始图片中加入高斯噪声，直到最后的图像趋于高斯分布。所以噪声占图像的比值会越来越大。

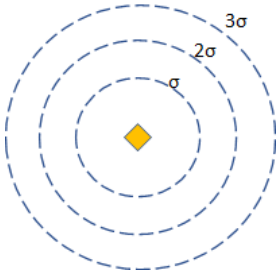


前向过程是不停加噪的过程，加入的噪声随着事件步增加增多，根据马尔可夫定理，加噪后这一时刻与前一时刻的相关性最高也与要加噪的噪音有关（是上一时刻的映像大还是要加的噪声影响大，当前向时刻越往后，噪音的权重就越来越大了，因为刚开始加的噪声有了效果，之后加入的噪声越来越多）。在论文中前想过成下一状态各点的分布被看成是只和前一时刻的状态相关的，这是一个马尔可夫的过程。

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

这个过程中，随着t增大，图像越来越接近纯噪声。当T趋向于∞，图像是完全噪声。β（<1）超参数是认为确定的一组线性插值，用于确定q的表达式，由于T趋于无穷，图像趋于噪声，因此β应该越来越大。

我们以一个二维图像为例，看看加噪的过程：



每一个上衣状态的噪点可以看成是以原来位置为均值，以σ为标准差，每一次递推，点都以正态分布的概率分布向周围跃迁。因此随着时间推进，所有点组成的整体将在时间趋于无穷大的时候趋向于整体正态分布的混乱状态。

2. 参数重整化（VAE）

参数重整化技术在很多工作中都有使用。如果我们要从一个分布中随机采样（高斯分布）的样本，这个过程无法反向传递梯度（离散）。但是这个通过高斯噪声采样得到的过程却是可微的（因此可以传递梯度），这是由于每一个噪点相比于采样，足够小。通常的做法是把随机性通过一个独立的随机变量（ε）引导过去。

3. 利用VAE，任意时刻的x_t可以由x_0和β表示

要得到某个状态x_t，如果利用公式1的马尔可夫递推式，每次计算某个状态都需要$O(t)$的时间复杂度，这个过程是十分复杂度。使用VAE可以将这个问题优化成$O(1)$的过程。下面是公式的推导：

令 $\alpha_t = 1 - \beta_t$

$$
\begin{aligned}
x_t &= \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t} z_1 \quad where \quad N(z_i; 0, I) \\
&= \sqrt{\alpha_t}(\sqrt{\alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_{t-1}} z_2) + \sqrt{1-\alpha_t} z_1 \\
&= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + (\sqrt{\alpha_t(1-\alpha_{t-1})} z_2 + \sqrt{1-\alpha_t} z_1) \\
&= \ldots = \sqrt{\prod_{i=1}^{T} \alpha_i} x_0 + \sqrt{1 - \prod_{i=1}^{T} \alpha_i} \prod_{i=1}^{T} z_i
\end{aligned} \tag{2}
$$

其中两个更好下的和α相关的系数可以利用前缀和的思想计算出来并存放在数组中。因此要求同时刻的图像，只需要在0好图像上叠加t个系数倍的标准正态分布即可。

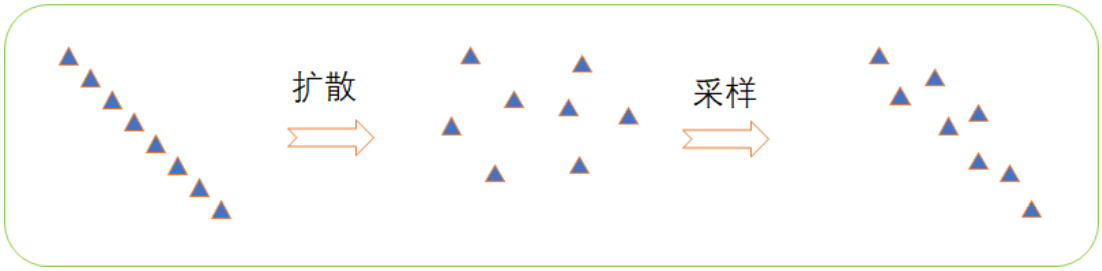
公式2中第三行到第四行可以利用正态分布叠加公式：

$$
\begin{aligned}
&若 z_1 \sim N(\mu_1, \sigma_1^2), z_2 \sim N(\mu_2, \sigma_2^2) \\
&则 z_3 \sim N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2) \\
&令 z_1 \sim N(0, (\sqrt{\alpha_t(1-\alpha_{t-1})})^2), z_2 \sim N(0, (\sqrt{1-\alpha_{t-1}})^2) \\
&故 z_3 \sim N(0, (\sqrt{1-\alpha_t \alpha_{t-1}})^2)
\end{aligned}
$$

（三）反向Sampling

1. 反向传播要做什么？

如果说前向过程forward是加噪过程，那么逆向过程（reverse）就是diffusion的去噪过程。实际上可以理解成，随机抽取一些时刻，对部分点进行同样正态分布的移动。但是不移动的点就是扩散后的。所以整个图像最终仍然有原来的形状，但是部分点是混乱的，所以呈现扩散。



但是加噪之后的图像已经变成噪音图了，我们无法轻易推导出反向递推式$q(x_{t-1}|x_t)$。因此这里我们使用深度神经网络去预测这样一个逆向的分布p_θ

$$
\begin{aligned}
p_\theta(X_{0:T}) &= p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \\
p_\theta(x_{t-1}|x_t) &= N(x_{t-1};\mu_\theta(x_t,t),\Sigma_\theta(x_t,t))
\end{aligned}
\tag{3}
$$

可以证明逆向过程也可以看成一个正态的马尔可夫模型，只是我们并不知道这个正态分布的均值和方差是多少，我们用后一个状态和时间的函数来表示它，希望通过深度学习网络来学习得到这两个参数μ_θ和Σ_θ。

虽然我们无法得到$q(x_{t-1}|x_t)$，但如果直到x_0，是可以通过贝叶斯公式得到$q_(x_{t-1}|x_t,x_0)$的：

$$
\begin{aligned}
q(x_{t-1}|x_t,x_0) &= q(x_t|x_{t-1},x_0)\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)} \\
&\propto \exp\left(-\frac{1}{2}\left(\frac{(x_t-\sqrt{\alpha_t}x_{t-1})^2}{\beta_t}+\frac{(x_{t-1}-\sqrt{\bar{\alpha}_{t-1}}x_0)^2}{1-\bar{a}_{t-1}}-\frac{(x_t-\sqrt{\bar{\alpha}_t}x_0)^2}{1-\bar{a}_t}\right)\right) \\
&= \exp \\
&\left(-\frac{1}{2}\Big(\underbrace{(\frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}})x_{t-1}^2}_{x_{t-1}\text{方差}}-\underbrace{(\frac{2\sqrt{\alpha_t}}{\beta_t}x_t+\frac{2\sqrt{\bar{a}_{t-1}}}{1-\bar{\alpha}_{t-1}}x_0)x_{t-1}}_{x_{t-1}\text{均值}}+\underbrace{C(x_t,x_0)}_{\text{与}x_{t-1}\text{无关}}\Big)\right).
\end{aligned}
$$

上公式第一行将逆向公式利用贝叶斯公式展开成$(x_{t-1},x_0)\rightarrow x_t$的正向过程。公式第二行利用高斯函数的密度表达式代入。为了简化计算，忽略系数，只看指数部分，对于指数部分可以分解为三个部分：

- x_{t-1}的方差部分
- x_{t-1}的均值部分
- 与x_{t-1}无关的常数部分

再经过一系列的整理（略），可以得到其中一个参数的值：

$$\mu_\theta(x_t,t)=\frac{1}{\sqrt{\alpha_t}}(x_t-\frac{\beta_t}{\sqrt{1-\prod_{i=1}^{T}\alpha_i}}z_\theta(x_t,t)) \tag{4}$$

第二个参数Σ_θ在不同的模型中有不同的处理办法，在DDPM中使用β_t直接表示，在GLIDE中根据网络来预测。在这里的实现中，对于Σ_θ的处理我们次啊用和DDPM的方式来简单替代。

2. Diffusion训练

在知道反向传播要做什么之后，就弄清楚diffusion的反向推断过程了。如何训练diffusion模型得到靠谱的μ_θ和Σ_θ是这个过程的主要目的。因此我们需要定义学习过程的损失函数。我们的目的是最大化模型预测分布的对数似然，即优化在$x_0\sim q(x_0)$下的$p_\theta(x_0)$交叉熵：

$$Loss=E_{q(x_0)}[-logp_\theta(x_0)] \tag{5}$$

利用公式7的变分下限（VLB）来优化负对数似然。由于KL散度非负，可以得到如下不等式：

$$-logp_\theta(x_0)\leq -logp_\theta(x_0)+D_{KL}(q(x_{1:T}|x_0)||p_\theta(x_{1:T}|x_0)) \tag{6}$$

因此，要使得损失函数最小，只要不等式右边最小。利用然后对不等式两边取期望，再利用Fubini定理，可以进一步化简，这里略。最终得到，损失函数可以通过用：

$$L_t^{simple}=E_{x_0,t}[||\overline{z}_t-z_\theta(\sqrt{\overline{\alpha}_t}x_0+\sqrt{1-\overline{\alpha}_t}\overline{z}_t,t)||^2] \tag{7}$$

其实我们最终只要使得：

$$||\overline{z}_t-z_\theta(\sqrt{\overline{\alpha}_t}x_0+\sqrt{1-\overline{\alpha}_t}\overline{z}_t,t)||^2 \tag{8}$$

最小就行了。所以训练过程可以总结为：

1. 获取输入x_0，从1...T中随机采样一个t
2. 从标准高斯分布中采样一个随机噪声$\overline{z}_t\sim N(0,I)$
3. 损失函数可以看成是最小化$||\overline{z}_t-z_\theta(\sqrt{\overline{\alpha}_t}x_0+\sqrt{1-\overline{\alpha}_t}\overline{z}_t,t)||^2$

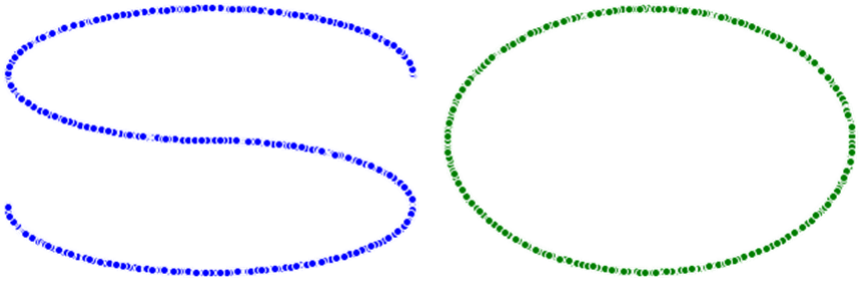
（四）代码实现

1. 数据生成

为了展示Diffusion带来的图像融合现象。我们生成两种不同的图像，一个是s_curve图形，还有一个是圆圈。

```
s_curve,_ = make_s_curve(10**4, noise=0.001)
circles,_ = make_circles(10**4, noise=0.001, factor=0.99999)
```

生成图像如下：



在后续的实验中，我们将S图形加噪之后转化为正态图形，然后对S图形的部分点进行替换，但是仍然保持图形是一个噪声图。当完成去噪后，图形变成 $\oint$ 这样一个闭合曲线积分符号。需要注意的是，我们的正向过程是对S进行加噪的，但是加入O图形后去噪。O图形不会不仅不会被当成噪音，并且它和S一样都得到了扩散过程。我们理解这个过程为，保留S的特征，加入融合图像的特征。在现实中就是通过这种方式利用diffusion进行AI绘画，将个输入图像融合并进行绘画。

2. 确定超参数

根据上面的公式推导，我们需要的超参数包括：α 序列、β 序列、$\overline{\alpha}$序列 和 $\sqrt{\overline{\alpha}}$序列、$\sqrt{1-\overline{\alpha}}$序列，当然为了高效计算损失函数，我们还需要序列 $log(\overline{\alpha})$ 和 $log(1-\overline{\alpha})$序列。

下面是计算超参数的代码。

```
# 确定超参数的值（T = 100，共100个采样时刻）
num_steps = 100  #可由beta值估算

# 利用sigmoid函数生成beta，按照时间从小到大变化
betas = torch.linspace(-6,6,num_steps)
betas = torch.sigmoid(betas)*(0.5e-2 - 1e-5)+1e-5

# 计算alpha、alpha_prod、alpha_prod_previous、alpha_bar_sqrt等变量的值
```

```
alphas = 1-betas
# alpha连乘
alphas_prod = torch.cumprod(alphas,0)
# 从第一项开始，第0项另乘1
alphas_prod_p =
torch.cat([torch.tensor([1]).float(),alphas_prod[:-1]],0)
# alphas_prod开根号
alphas_bar_sqrt = torch.sqrt(alphas_prod)
# 之后公式中要用的
one_minus_alphas_bar_log = torch.log(1 - alphas_prod)
one_minus_alphas_bar_sqrt = torch.sqrt(1 - alphas_prod)

# 大小都一样，常数不需要训练（超参数）
assert alphas.shape==alphas_prod.shape==alphas_prod_p.shape==
alphas_bar_sqrt.shape==one_minus_alphas_bar_log.shape
==one_minus_alphas_bar_sqrt.shape
print("all the same shape",betas.shape)
```

生成的向量维数等于规定的正向过程周期数T，表示这些草参数是t=1..T的不同时间段的超参数序列。

3. 正向传播forwarding过程

正如原理部分所说的，如果按照

$$q(x_t|x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{9}$$

的马尔可夫递推式生成状态，将会是一个非常低效的过程，但是通过VAE参数重整化技术，可以得到t时刻的x_t的通项表达式。

$$x_t = q_x(x_0, t) = \sqrt{\prod_{i=1}^{T}\alpha_i}x_0 + \sqrt{1-\prod_{i=1}^{T}\alpha_i}\prod_{i=1}^{T}z_i \tag{10}$$

根据公式写出代码：

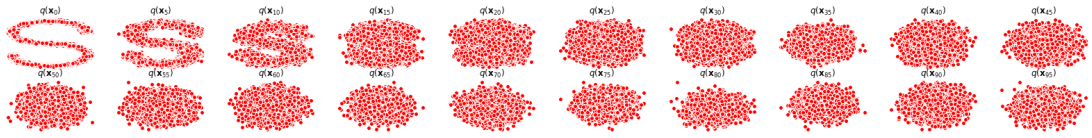
```
# 计算任意时刻的x采样值，基于x_0和重参数化
def q_x(x_0, t):
    """可以基于x[0]得到任意时刻t的x[t]"""
    # 生成正态分布采样
    noise = torch.randn_like(x_0)
    # 得到均值方差
    alphas_t = alphas_bar_sqrt[t]
    alphas_1_m_t = one_minus_alphas_bar_sqrt[t]
    # 利用参数重整化，根据x0 求 xt
    return (alphas_t * x_0 + alphas_1_m_t * noise)
```

在实现实验过程中，我定义T = 100个时钟周期。整个正向传播过程代码如下

```
# 演示加噪过程，加噪100步情况，展示20个
num_shows = 20
fig,axs = plt.subplots(2,10,figsize=(28,3)) # 画一个2x10的图
plt.rc('text',color='black')

# 共有10000个点，每个点包含两个坐标
# 生成100步以内每隔5步加噪声后的图像，扩散过程散点图演示——基于x0生成条件分布采样得到xt
for i in range(num_shows):
    j = i//10
    k = i%10
    q_i = q_x(dataset,torch.tensor([i*num_steps//num_shows]))
    # 生成t时刻的采样数据
    axs[j,k].scatter(q_i[:,0],q_i[:,1],
                     color='red',edgecolor='white')
    axs[j,k].set_axis_off()
    axs[j,k].set_title(
        '$q(\mathbf{x}_{'+str(i*num_steps//num_shows)+'})$')
plt.show()
```

这里加噪100步，每隔10步进行1次展示，加噪过程如下：



我们对S图形加噪，最终图形趋向于完全正态分布的图像。

4. 定义反向神经网络

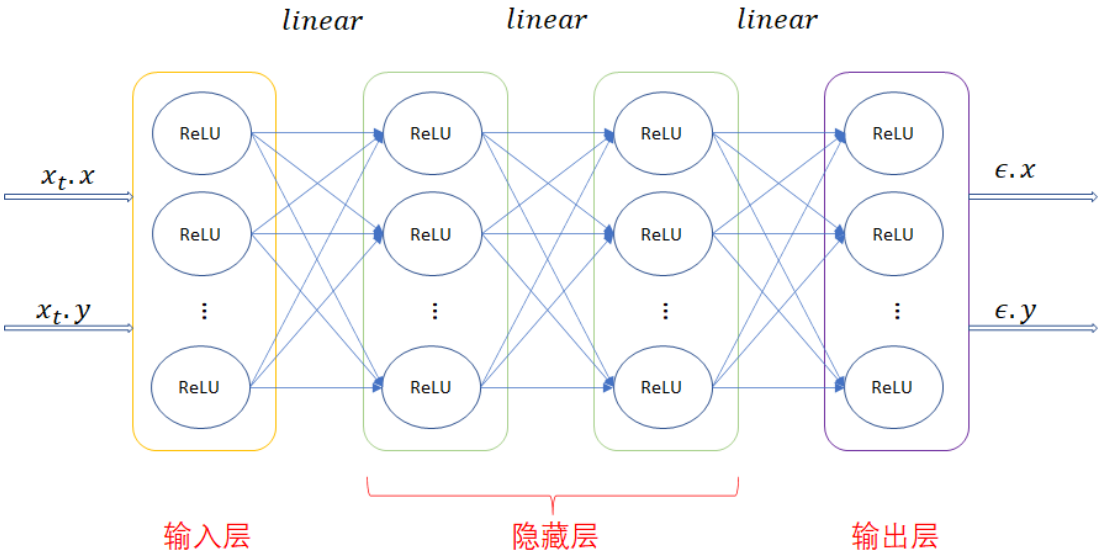
利用pytorch带的多层感知机模型（MLP），可以定义一个简单的深度神经网络。为了简单起见，仅定义一个输入层、一个输出层、和两个隐藏层。每层采用128个神经元，作为批处理单元。

```
# 自定义神经网络（多层感知机模型）
class MLPDiffusion(nn.Module):
    def __init__(self, n_steps, num_units=128):
        super(MLPDiffusion, self).__init__()

        # ⭐含义如下图！！！
        self.linears = nn.ModuleList(
            [
                nn.Linear(2, num_units),
                nn.ReLU(),
                nn.Linear(num_units, num_units),
```

```
                nn.ReLU(),
                nn.Linear(num_units, num_units),
                nn.ReLU(),
                nn.Linear(num_units, 2),
            ]
        )
        self.step_embeddings = nn.ModuleList( # 转换特征维度，构造层次全连接结构
            [
                nn.Embedding(n_steps, num_units),
                nn.Embedding(n_steps, num_units),
                nn.Embedding(n_steps, num_units),
            ]
        )
```

定义输入参数以线性加权的方式存在。同时激活函数采用常用的ReLU函数。输入的是下一时刻的所有点的坐标向量，输出的是预测从t-1到t时刻产生的噪声的反向量。



因此计算输出的前向传播（不同于整个diffusion的前向传播）就是：

```
# 前向传播
    def forward(self, x, t):
        #         x = x_0
        for idx, embedding_layer in
        enumerate(self.step_embeddings):
            t_embedding = embedding_layer(t)
            x = self.linears[2 * idx](x) # linear 线性加权
            x += t_embedding
            x = self.linears[2 * idx + 1](x) # 调用激活函数
        x = self.linears[-1](x) # 最后一层得出的结果
        return x
```

embedding的作用是防止梯度爆炸或者梯度缺失。

5. 定义损失函数

损失函数可以用$||\overline{z}_t - z_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\overline{z}_t, t)||^2$进行似然估计。因此我们将损失函数diffusion_loss_fn定义成上面的表达式的期望：

```
def diffusion_loss_fn(model, x_0, alphas_bar_sqrt,
one_minus_alphas_bar_sqrt, n_steps):
    """ 对任意时刻t进行采样计算loss """
    batch_size = x_0.shape[0] # cuda batch

    # 对一个batchsize样本生成随机的时刻t，t变得随机分散一些，一个
batch size里面覆盖更多的t
    t = torch.randint(0, n_steps, size=(batch_size // 2,))
    t = torch.cat([t, n_steps - 1 - t], dim=0) # t的形状（bz）
    t = t.unsqueeze(-1) # t的形状（bz,1）

    # x0的系数，根号下(alpha_bar_t)
    a = alphas_bar_sqrt[t]

    # eps的系数,根号下(1-alpha_bar_t)
    aml = one_minus_alphas_bar_sqrt[t]

    # 生成随机噪音eps
    e = torch.randn_like(x_0)

    # 构造模型的输入
    x = x_0 * a + e * aml

    # 送入模型，得到t时刻的随机噪声预测值
    output = model(x, t.squeeze(-1))

    # 与真实噪声一起计算误差，求平均值
    return (e - output).square().mean()
```

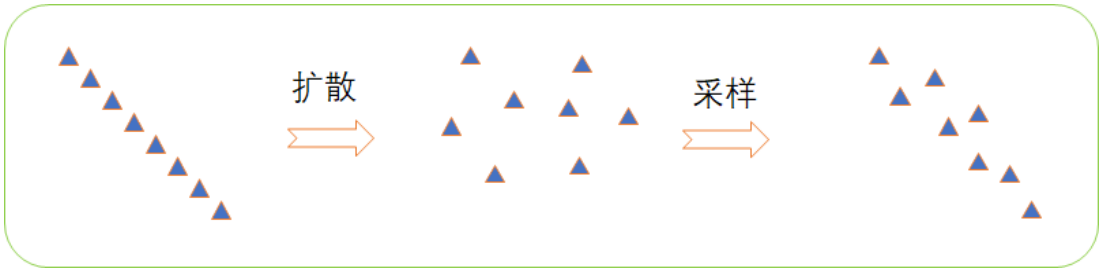
注意最后需要计算的是生成的随机噪声$\bar{z}_t$和模型输出的噪声z_θ之间的误差平方平均数。

6. 反向采样过程

反向采样利用每一次预测的上一时刻图像，不断地推，最终生成x_0。根据原理部分的讲解，每一次采样，需要计算μ_θ和Σ_θ，并利用模型生成的噪声产生图像采样。

```
''' 采样逆向重构函数 '''
def p_sample(model, x, t, betas, one_minus_alphas_bar_sqrt):
    t = torch.tensor([t])
    coeff = betas[t] / one_minus_alphas_bar_sqrt[t]
    eps_theta = model(x, t)
    #得到均值
    mean = (1 / (1 - betas[t]).sqrt()) *
    (x - (coeff * eps_theta))
    z = torch.randn_like(x)
    sigma_t = betas[t].sqrt()
    #得到sample的分布
    sample = mean + sigma_t * z
    return (sample)
```

逆向过程实际上就是不断采样，并移动部分点，采样得到的点进行着近似可逆的移动，所以又回到了原来的形状，但是部分点没有被采样到，所以仍然保持扩散状态。



循环采样，定义为迭代采样num_epoch次：

```python
#从xt恢复x0
def p_sample_loop(model, shape, n_steps, betas,
one_minus_alphas_bar_sqrt):
    """从x[T]恢复x[T-1]、x[T-2]|...x[0]"""
    cur_x = torch.randn(shape)
    x_seq = [cur_x]
    for i in reversed(range(n_steps)): # 反向计算xt
        cur_x = p_sample(model, cur_x, i, betas,
                         one_minus_alphas_bar_sqrt)
        x_seq.append(cur_x)
    return x_seq
```

7. 训练过程

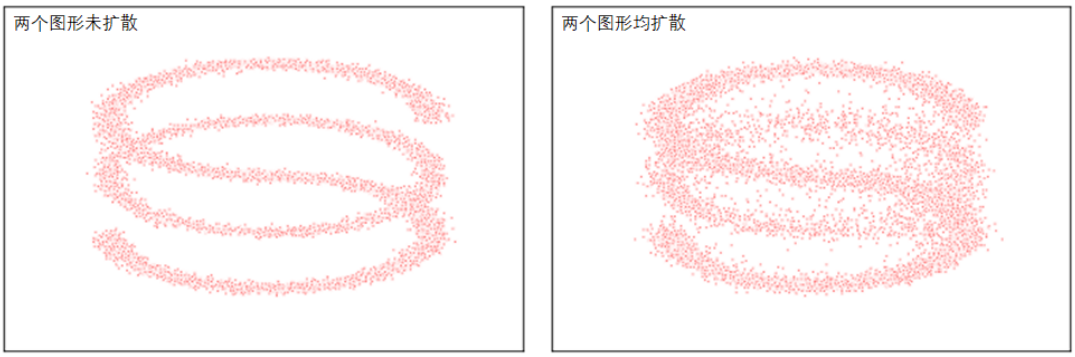
迭代num_epoch次，并对于每个批（将布片划分好几个批，更高效）进行处理，然后计算损失函数，并进行反向BP传播。同时每100个次迭代打印重建效果。

```python
# epoch遍历（重构周期数）
for t in range(num_epoch):
    # dataloader遍历
    for idx, batch_x in enumerate(dataloader):
        # enumerate为输入序列分配迭代次序
        # 得到loss
        loss = diffusion_loss_fn(model, batch_x,
        alphas_bar_sqrt,
        one_minus_alphas_bar_sqrt, num_steps)
        optimizer.zero_grad()
        loss.backward() # 反向传播
        # 梯度剪裁：保持稳定性，避免梯度爆炸
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.)
        optimizer.step()
    # 每100步打印重构效果
    if (t % 100 == 0):
        print(loss)
        # 根据参数采样一百个步骤的x，每隔十步画出来，
        # 迭代了2000个周期，逐渐更接近于原始
        x_seq = p_sample_loop(model, dataset.shape,
                  num_steps, betas, one_minus_alphas_bar_sqrt)
        fig, axs = plt.subplots(1, 10, figsize=(28, 3))
        for i in range(1, 11):
            cur_x = x_seq[i * 10].detach()
            # 画散点图
            axs[i - 1].scatter(cur_x[:, 0],
                  cur_x[:, 1], color='red', edgecolor='white')
            axs[i - 1].set_axis_off()
            axs[i - 1].set_title('$q(\mathbf{x}_{' +
```

```
                                        str(i * 10) + '})$')
plt.show()

```

训练前后变化：



可以看到重建后，$发生扩散，但是只是对S图形进行加噪，将两个图形融合并共同扩散的工作是由神经网络实现的。

8. 将前向后向过程合并

为了更好观察这个过程，将前向后向过程中得到的所有图片的快照合成一个GIF图像，便于观察（详见工程项目文件中“diffusion.gif”）

```
imgs = []
for i in range(100):
    plt.clf()
    q_i = q_x(dataset, torch.tensor([i]))
    plt.scatter(q_i[:, 0], q_i[:, 1], color='red',
edgecolor='white', s=5)
    plt.axis('off')

    img_buf = io.BytesIO()
    plt.savefig(img_buf, format='png')
    img = Image.open(img_buf)
    imgs.append(img)
# plt.show()
reverse = []
for i in range(100):
    plt.clf()
    cur_x = x_seq[i].detach()
    plt.scatter(cur_x[:, 0], cur_x[:, 1], color='red',
edgecolor='white', s=5)
    plt.axis('off')

    img_buf = io.BytesIO()
    plt.savefig(img_buf, format='png')
```

```
    img = Image.open(img_buf)
    reverse.append(img)
# plt.show()
imgs = imgs + reverse
imgs[0].save("diffusion.gif",format='GIF',append_images=imgs,save_all=True,duration=100,loop=0)

```

五、思考题

深度学习算法参数的设置对算法性能的影响？

答：深度学习算法参数的设置对算法性能的影响是一个很复杂的问题，没有一个统一的答案。不同的参数可能会影响模型的收敛速度、稳定性、泛化能力等方面。一般来说，需要根据具体的问题、数据集、模型结构等因素来选择合适的参数，或者采用一些超参数优化方法来寻找最佳的参数组合。常见的超参数包括：

- 学习率：控制每次梯度下降的步长，过大可能导致不收敛，过小可能导致收敛速度慢。
- 迭代次数：控制模型训练的总轮数，过多可能导致过拟合，过少可能导致欠拟合。
- 隐层数目：控制模型的深度，影响模型的表达能力和复杂度。
- 隐层单元数：控制模型的宽度，影响模型的表达能力和复杂度。
- 激活函数：控制模型的非线性程度，影响模型的拟合能力和梯度传播。
- 批量大小：控制每次梯度下降使用的样本数，影响模型的收敛速度和稳定性。
- 优化器：控制模型的优化策略，影响模型的收敛速度和效果。
- 正则化：控制模型的惩罚项，影响模型的泛化能力和抗干扰能力。

以上是常见的参数以及他们会对算法照成什么影响。

六、实验总结

在本实验中，我根据论文“Diffusion Models: A Comprehensive Survey of Methods and Applications”[1]实现了Diffusion模型的代码，并在一些数据集上进行了简单的实验。Diffusion模型是一种强大的深度生成模型，它可以在多种应用领域中取得最先进的性能，包括图像合成、视频生成和分子设计等

[2]。Diffusion模型的基本思想是通过一个正向扩散过程将数据分布逐渐扰动为一个已知的先验分布，然后通过一个反向扩散过程学习恢复数据分布[3]。

在我的实验中，我使用了论文中提供的代码框架，并根据不同的数据集和任务调整了一些参数和设置。我主要关注了Diffusion模型在图像合成和分子设计两个应用领域的表现。我使用了sklearn生成的两个散点数据集作为输入，并使用了预训练好的分数函数网络作为反向扩散过程的核心组件。我按照论文中建议的步骤进行了采样，并观察了生成样本的质量和多样性。

我的实验结果表明，Diffusion模型可以生成高质量和高分辨率的图像和分子结构，与原始数据集具有很高的相似度和一致性。同时，Diffusion模型也可以生成一些新颖和有趣的样本，展示了其创造力和泛化能力。例如，我利用S型曲线和O型曲线合成了 $\oint$ 扩散符号。这些样本都是从标准高斯分布开始逐步恢复而来，说明了Diffusion模型可以有效地利用分数函数网络来逆转扩散过程，这个逆转过程不一定非得是原来方向的反方向。

通过这次实验，我对Diffusion模型有了更深入的理解和认识，也体会到了它在生成式建模领域的强大潜力和优势。我感到非常受用和开心，也很感谢论文作者提供了详尽的介绍和方便的代码。我希望未来能够继续学习和探索Diffusion模型的更多方法和应用。

更加重要的是对神经网络的正向传播、反向传播、神经网络的构建都有更深的理解。

七、参考

Ps：[5]是论文原文

[1] [2209.00796] Diffusion Models: A Comprehensive Survey of Methods and Applications (arxiv.org) ↩
[2] 轻松学习扩散模型（diffusion model），被巨怪踩过的脑袋也能懂——原理详解+pytorch代码详解（附全部代码） - 知乎 (zhihu.com) ↩
[3] 由浅入深了解Diffusion Model - 知乎 (zhihu.com) ↩