Denoising Diffusion Implicit Models

Jiaming Song, Chenlin Meng Strfano Ermon

摘要

扩散模型是一类以高斯噪声为扰动源的生成模型，迄今为止该模型已被广泛应用于各种生成建模任务。去噪扩散概率模型因其需要多次迭代来产生高质量的样本，生成过程包含数千个步骤，耗时耗力的缺点在生成模型中尤为突出。该论文基于去噪扩散概率模型的基础理念，在保证生成样本质量之上，进一步改进去噪扩散概率模型，有效减少模型采样时间，是并广泛应用到现在的扩散模型中。基于此论文以及开源代码复现了一个车的生成模型项目，并在多次调整参数后生成样本的效果表现不错。

关键词：扩散模型；深度学习；图像生成

1　引言

生成模型是一类可以根据某些隐含参数随机生成观察结果的模型。近年来，扩散模型因其能力而成为一类日益崛起的生成模型。扩散模型是基于正向扩散阶段和反向扩散阶段两个阶段的深层生成模型。在前向扩散阶段，输入数据通过加入高斯噪声逐步被扰动。在相反的阶段，一个模型负责通过学习逐步逆转扩散过程来恢复原始输入数据。扩散模型因生成样本的质量和多样性而得到广泛赞赏，尽管它们有已知的计算负担，即由于采样过程中涉及的高步骤而速度较低。

本次复现的论文提出了一种更有效的迭代隐式概率模型（DDIM）[1]，通过一类非马尔可夫扩散过程来改进原有的去噪扩散概率模型（DDPM）[2]中特定的马尔可夫扩散过程的反向过程，从而产生能够快速产生高质量样本的隐式模型。

2　相关工作

2.1　去噪扩散概率模型的正向过程

给定一个初始数据分布 $X_0 \sim q(X)$，不断向该分布中添加高斯噪声，一共加 T 次，所添加噪声的均值是由预先确定的超参数 β_t 所确定的，方差是由 β_t 和当前 t 时刻的数据 X_t 所决定的，此过程是一个马尔可夫链。要求随着 t 的不断增大最终到达 T 时刻时，所得的数据分布 X_T 最终变成一个各项独立的高斯分布，在每个时间步内，$q(X_t|X_{t-1})$ 可以表示为，给定 X_{t-1}，服从均值为 $\sqrt{1-\beta_t}X_{t-1}$，方差为 $\beta_t I$ 的正态分布，其中 I 是单位矩阵。

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

因为是马尔可夫链，所以联合分布就是：

$$q(X_{1:T}|X_0) = q(X_1, X_2, ..., X_T|X_0) = \prod_{t=1}^{T} q(X_t|X_{t-1}) \tag{2}$$

每步采样过程重参数化可得到 $X_t = \sqrt{\alpha_t}X_{t-1} + \sqrt{1-\alpha_t}Z_{t-1}$，每步噪声都是马尔可夫的连乘结果，

即 $\overline{\alpha}_t = \prod_{i=1}^{T} \alpha_i$，于是可推出：

$$\begin{aligned} X_t &= \sqrt{\alpha_t} X_{t-1} + \sqrt{1-\alpha_t} Z_{t-1} = \sqrt{\alpha_t}(\sqrt{\alpha_{t-1}} X_{t-2} + \sqrt{1-\alpha_{t-1}} Z_{t-2}) + \sqrt{1-\alpha_t} Z_{t-1} \\ &= \sqrt{\alpha_t \alpha_{t-1}} X_{t-2} + \sqrt{\alpha_t - \alpha_t \alpha_{t-1}} Z_{t-2} + \sqrt{1-\alpha_t} Z_{t-1} \end{aligned} \tag{3}$$

对于两个正态分布 $X \sim N(\mu_1, \sigma_1^2)$ 和 $X \sim N(\mu_2, \sigma_2^2)$，线性叠加后的分布 aX+bY 的均值是 $a\mu_1 + b\mu_2$，方差是 $a^2\sigma_1^2 + b^2\sigma_2^2$。因为 Z_{t-2} 和 Z_{t-1} 都服从标准正态分布，所以可合并得到：

$$X_t = \sqrt{\alpha_t \alpha_{t-1}} X_{t-2} + \sqrt{1-\alpha_t \alpha_{t-1}} Z_{t-2} = \ldots = \sqrt{\overline{\alpha}_t} X_0 + \sqrt{1-\overline{\alpha}_t} Z_t \quad Z_t \sim N(0,1) \tag{4}$$

再由上述公式可得到：

$$q(X_t|X_0) = N(X_t; \sqrt{\overline{\alpha}_t} X_0, (\sqrt{1-\overline{\alpha}_t})^2 I) \tag{5}$$

所以 X_t 在 X_0 条件下的分布是均值为 $\sqrt{\overline{\alpha}_t} X_0$，方差为 $1-\overline{\alpha}_t$ 的正态分布。

2.2 逆扩散过程

逆扩散过程是从高斯噪声中一步步恢复原始分布，与正向扩散过程类似，同样假设每一步都是一个高斯分布 $q(X_{t-1}|X_t)$。逆扩散过程仍然是一个马尔可夫链，训练一个网络 $p_\theta(X_{t-1}|X_t)$ 来近似得到 $q(X_{t-1}|X_t)$。假设有一个网络输入是 X_t 和 t，能估计出条件概率 $p_\theta(X_{t-1}|X_t)$，也就是均值为 $\mu_\theta(X_t, t)$，方差为 $\sum_\theta^2(X_t, t)$ 的高斯分布：

$$p_\theta(X_{t-1}|X_t) = N\left(X_{t-1}; \mu_\theta(X_t, t), \sum_\theta^2(X_t, t)\right) \tag{6}$$

条件概率的方差直接取值 β_t，则只需要通过网络去预测均值。所以上述公式则简化为：

$$p_\theta(X_{t-1}|X_t) = N(X_{t-1}; \mu_\theta(X_t, t), \beta_t) \tag{7}$$

$$p_\theta(X_{0:T}) = p_\theta(X_0, X_1, \ldots, X_T) = \prod_{t=1}^{T} p_\theta(X_{t-1}|X_t) \tag{8}$$

3 本文方法

3.1 非马尔可夫正向过程

首先考虑推理分布表示为：

$$q_\sigma(X_{1:T}|X_0) := q_\sigma(X_T|X_0) \prod_{t=2}^{T} q_\sigma(X_{t-1}|X_t, X_0) \tag{9}$$

其中 $q(X_t|X_0) = N(\sqrt{\overline{\alpha}_t} X_0, (\sqrt{1-\overline{\alpha}_t})^2 I)$ 服从高斯分布，取均值函数进行排序保证该式对所有 t 都成立。因此论文定义了一个联合推断分布与边缘分布相匹配，正向过程从贝叶斯规则推导出：

$$q_\sigma(X_t|X_{t-1}) = \frac{q(X_{t-1}|X_t) q(X_t|X_0)}{q(X_{t-1}|X_0)} \tag{10}$$

每个 X_t 都可以依赖于 X_{t-1} 和 X_0，因此正向过程不再是马尔可夫链过程。

q_σ 中 σ 的值用来控制正向过程的随机程度。当其趋近于 0，给定 t 时间的 X_t 和 X_0，那么 X_{t-1} 即为已知的。

3.2 逆向过程

每个逆向过程 $p_\theta^{(t)}(X_{t-1}|X_t)$ 都利用了正向过程结果中的 $q_\sigma(X_{t-1}|X_t, X_0)$。给定一个噪声 $X_t = \sqrt{\overline{\alpha}_t} X_0 + \sqrt{1-\overline{\alpha}_t}\varepsilon$，用神经网络 $\varepsilon_\theta^{(t)}(\overline{\alpha}_t)$ 从 X_t 预测出 ε_t:

$$\mathrm{f}_\theta^{(\mathrm{t})}(\overline{\alpha}_\mathrm{t}) := (\overline{\alpha}_\mathrm{t} - \sqrt{1-\overline{\alpha}_\mathrm{t}}\varepsilon_\theta^{(\mathrm{t})}(\overline{\alpha}_\mathrm{t}))/\sqrt{\overline{\alpha}_\mathrm{t}} \tag{11}$$

由此逆向过程可以定义为：

$$\mathrm{p}_\theta^{(\mathrm{t})}(\mathrm{X}_{\mathrm{t}-1}|\mathrm{X}_\mathrm{t}) = \begin{cases} \mathrm{N}(\mathrm{f}_\theta^{(1)}(\mathrm{X}_1), \sigma_1^2\mathrm{I}), & \mathrm{t}=1 \\ \mathrm{q}_\sigma(\mathrm{X}_{\mathrm{t}-1}|\mathrm{X}_\mathrm{t}, \mathrm{f}_\theta^{(\mathrm{t})}(\mathrm{X}_\mathrm{t})), & otherwise \end{cases} \tag{12}$$

3.3 损失函数定义

与其他生成模型类似，最小化在真实数据期望下，模型预测分布的负对数似然[3]。最小化预测 $\mathrm{p}_{\mathrm{data}} = \mathrm{q}(\mathrm{X}_0)$ 和 $p_\theta(X_0)$ 的交叉熵：

$$L = E_{x_0 \sim p_\theta(X_0)}\left[-\log\left(p_\theta\left(X_0\right)\right)\right] \tag{13}$$

对 $p_\theta(X_0)$ 推导可得：

$$L = E_{q(X_{0:T})}\left[\log\left(\frac{q\left(X_{1:T}\left|X_0\right.\right)}{p_\theta\left(X_{1:T}\right)}\right)\right] \tag{14}$$

最小化 L 可以转而最小化其上界 L_{VLB}:

$$L_{VLB}E_{q(X_{0:T})}\left[\log\left(\frac{q\left(X_{1:T}\left|X_0\right.\right)}{p_\theta\left(X_{1:T}\right)}\right)\right] \geqslant E_{q(X_0)}\left[-\log\left(p_\theta\left(X_0\right)\right)\right] \tag{15}$$

经过配分后可以将 L_{VLB} 分为 L_T、L_{t-1} 和 L_0 三部分，分别为：

$$\begin{gathered} L_\mathrm{T} : \mathrm{D_K}\left(\mathrm{q}\left(\mathrm{X_T}\left|\mathrm{X}_0\right.\right)||\mathrm{p}_\theta\left(\mathrm{X_T}\right)\right) \\ L_{t-1} : \sum_{t=1}^{T} D_{KL}\left(q\left(X_{t-1}\left|X_t\right., X_0\right)||p_\theta\left(X_{t-1}\left|X_t\right.\right)\right) \\ L_0 : -\log\left(\mathrm{p}_\theta\left(\mathrm{X}_0\left|\mathrm{X}_1\right.\right)\right) \end{gathered} \tag{16}$$

可发现 L_{VLB} 是由一个熵（L_0），和多个 KL 散度（$L_T, \mathrm{t} \in \{1,2,3,\ldots,\mathrm{T}\}$）构成。因此最小化 L_{VLB} 只与 $L_T(\mathrm{t} \in \{1,2,3,\ldots,\mathrm{T}\})$ 有关。如果前向和逆向过程为高斯分布，可知方差都是常数，因此优化目标就是优化两个分布均值的二范数。

4 复现细节

4.1 与已有开源代码对比

本次实验结合论文在 Github 上的开源模型代码和去噪扩散概率模型论文所提供的开源模型代码，基于 StanfordCars 数据集，实现扩散模型对车图像的生成。个人实现的主要工作包括使用非马尔可夫过程取代了 DDPM 中使用的马尔可夫正向过程，使模型采样过程更快的修改优化并获得好的训练结果。如图 1 是复现模型实现流程，首先是完成数据集的处理，为保证数据量的充分利用，将数据进行拼接用作训练；接着是训练模型的实现，包括数据预处理，数据封装，损失函数，学习率等参数的定义，保存验证效果最好的模型进行测试。

复现模型的训练部分的实现过程为：对 torchvision 中 StanfordCars 数据集的封装并返回图像。训练的 250 轮 epoch，学习率经多次调整后为 0.00008,batchsize 为 32。

代码实现：模型部分

```python
class SimpleUnet(nn.Module):
    # Unet架构简化版本
    def __init__(self):
        super().__init__()
        image_channels = 3
        down_channels = (64, 128, 256, 512, 1024)
        up_channels = (1024, 512, 256, 128, 64)
        out_dim = 1
        time_emb_dim = 32

        # 时间嵌入
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU()
        )
        # 初始预估
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)
        # 下采样
        self.downs = nn.ModuleList(
            [Block(down_channels[i], down_channels[i + 1], time_emb_dim) for i in range(len(down_channels) - 1)])
        # 上采样
        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)])

        self.output = nn.Conv2d(up_channels[-1], 3, out_dim)

    def forward(self, x, timestep):

        t = self.time_mlp(timestep)

        x = self.conv0(x)

        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()

            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)


model = SimpleUnet()
```

图 1: U-Net 网络

```python
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, ):
        # 第一次卷积
        h = self.bnorm1(self.relu(self.conv1(x)))
        # 时间嵌入
        time_emb = self.relu(self.time_mlp(t))
        # 扩展到最后2个维度
        time_emb = time_emb[(...,) + (None,) * 2]
        # 添加时间通道
        h = h + time_emb
        # 第二次卷积
        h = self.bnorm2(self.relu(self.conv2(h)))
        # 上采样或者下采样
        return self.transform(h)
```

图 2: 方法函数

```python
# 将训练集和测试集一起调用，作为数据集
def load_transformed_dataset():
    data_transforms = [
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda t: (t * 2) - 1)
    ]
    data_transform = transforms.Compose(data_transforms)

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

    test = torchvision.datasets.StanfordCars(root="/pubdata/huocc/", download=True, transform=data_transform,
                                             split='test')

    return torch.utils.data.ConcatDataset([train, test])
```

图 3: 数据预处理

在 ddpm 一文中采用 PixelCNN++ 网络结构，通过短接构成类似于 Unet 的网络结构[4]，在复现网络中参考并设计了本次实验的网络结构。

在 train 函数中，循环调用 epoch，并在每一次 step 为 0 时输出上轮训练的结果。结果包括当前

epoch 和损失值以及生成图像的成像图。每个 epoch 循环结束都会做一次存储。

```python
for epoch in range(epochs):
    # print(dataloader)
    for step, batch in enumerate(dataloader):
        # exit()
        optimizer.zero_grad()
        # out = model(input)
        torch.cuda.manual_seed(2022)
        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
        loss = get_loss(model, batch[0], t)
        loss.backward()
        optimizer.step()
        if step == 0:
            print(f" Epoch {epoch}  |  step {step:03d}  Loss:{loss.item()}")
            image = batch[0]
            show_tensor_image(image)
            sample_plot_image()
    save_states_path = 'weights/{}.pth'.format(str(epoch + 1))
    states = {
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'optimizer': optimizer.state_dict(),
    }
    torch.save(states, save_states_path)
```

图 4: train

```python
@torch.no_grad()
def sample_plot_image():
    # 样本噪声
    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)
        if i % stepsize == 0:
            plt.subplot(1, num_images, int(i / stepsize + 1))
            show_tensor_image(img.detach().cpu())
    plt.show()
```

图 5: 样本输出

```python
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)
```

图 6: 损失函数

```python
# 采样
@torch.no_grad()
def sample_timestep(x, t):
    """
    调用模型里预测图像中的噪声，并返回去噪后的图像。
    如果还没有进入最后一步，则对该图像施加噪音。
    """
    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)
    # 调用模型
    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:
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise
```

图 7: 采样函数

4.2 实验环境搭建

使用 pycharm2022 远程连接服务器，1080tiGPU 训练

4.3 通过随机梯度朗之万动力学实现扩散模型

复现代码参考 Github 的开源代码，通过简单的瑞士卷点集的扩散和复原过程，模拟从动力学的角度实现扩散模型如何在噪声中逐步生成图像[5]。为保证实验结果在预期范围内，通过使用指数移动平均而对模型的参数做平均，增加模型鲁棒性。以下为模型设计的实现效果。

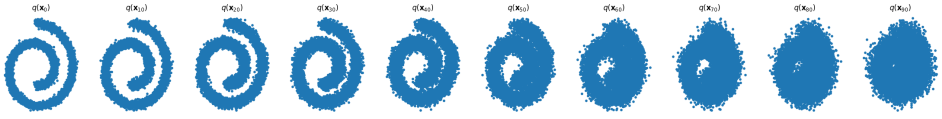

图 8: 前向扩散过程

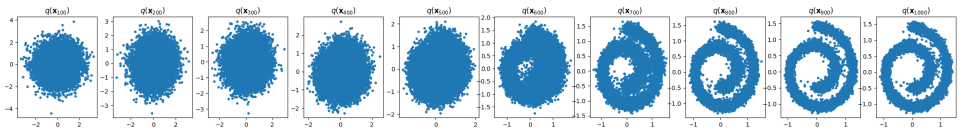

图 9: 逆向生成过程

5 实验结果分析

扩散模型受热力学启发，通过反转逐渐的噪声过程来学习生成数据。分为扩散过程（forward/diffusion process）和逆扩散过程（reverse process）。扩散过程 $(X_0 - X_T)$ 逐步对图像加噪声，这一逐步过程可以认为是参数化的马尔可夫过程。逆扩散过程（$X_T - X_0$）从噪声中反向推导，逐渐消除噪声以逆转生成图像。正向过程是个固定的过程，逐渐向图像中添加高斯噪声，直到最终得到纯噪声，逆向过程通过一个可学习的 p_θ 逐步降噪恢复得到图像。

扩散模型的构建首先需要描述一个逐步将数据转化为噪声的过程，然后训练一个神经网络，逐步学习将这个过程转化为噪声。这些步骤中的每一步都包含一个有噪声的输入，通过填充一些被噪声所掩盖的信息，使它的噪声稍微小一些，如果从纯噪声开始，并且这样做足够多次，就可以从这种方法生成数据。[6]

在设计 DDPM 的前期，由于给定的学习率较低，且模型训练耗时较长，模型在训练 50 个 epoch 后损失值已不发生较大改变，每次测试结果可见陷入局部最优解，如图所示。

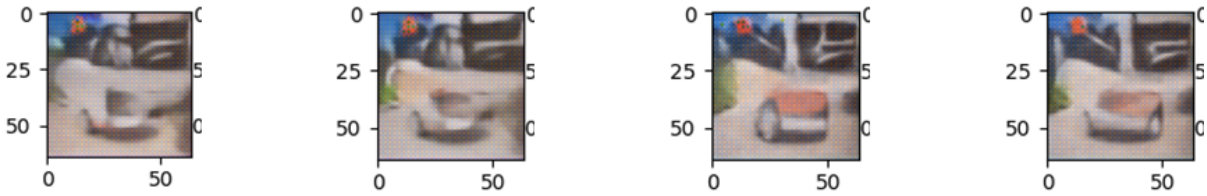

图 10: 局部最优实验结果

随着在几次对学习率的调整之后，模型已然能从噪声中生成较为不错的 64 像素大小的车的图像。以下是 6 张实验生成结果图。

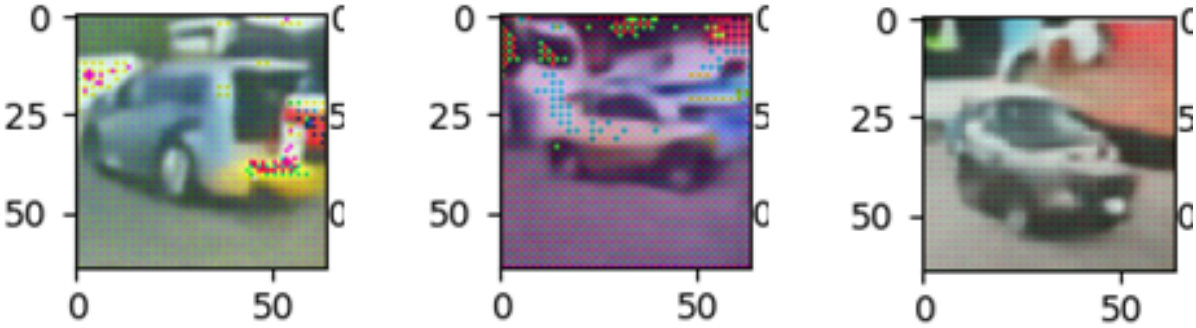

图 11: 实验结果示意图

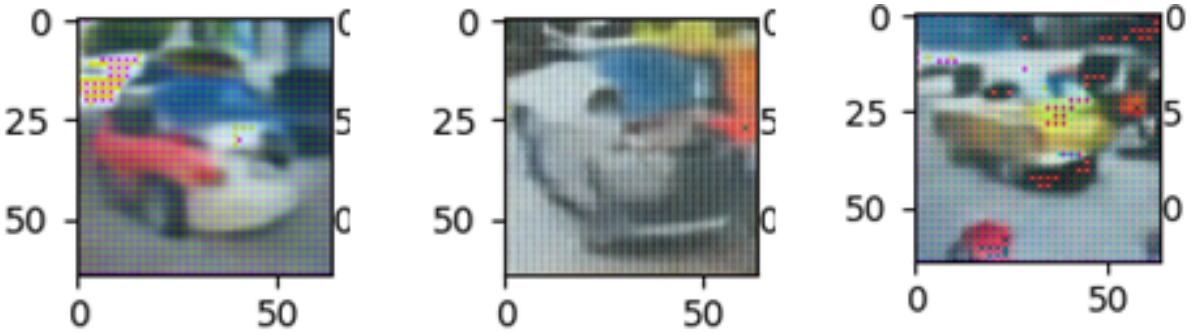

图 12: 实验结果示意图

DDIM 的加入使得原本的 DDPM 项目在同样的运行设备上，运行速度有了较大的提升，这种加速采样的巧妙已设计广泛应用于扩散模型中。

6 总结与展望

本次复现工作首先通过调研扩散模型领域的相关工作，深入浅出的理解了扩散模型的设计原理，了解到在对噪声的操作主要是通过预测网络一步步反推得到的。DDPM 一文作为将扩散模型引入深度学习的开山之作，其意义非同凡响。前向过程是一个固定的且可以用来作为模型学习的定向标记，但如若确定扩散步骤后仍重复计算前向噪声，势必会造成一定的资源浪费。训练过程中随机取来自真实未知且可能复杂的数据分布，也就是说在训练过程中 t 的取值是随机的，通过网络来预测噪声，其优点是能很好学习到数据的增广分布，缺点是给训练带来很大的挑战，也让模型复杂度倍增。DDIM 很巧妙的发现了这一弊端并推翻了文中的马尔可夫链的关系，另辟门路设定一个隐式变量来控制，在实验效果上也有一定的体现。

在复现过程中，首先参考 DDPM 论文中的设计思路设计并实现实验代码，然后逐步根据 DDIM 论文中的设计原理去修改实验代码，多次测试后选取最优实验结果作为实验展示。通过这段时间对扩散模型从无到有的学习，惊叹扩散模型不同于 gan 模型的生成方式和生成能力，未来会继续努力学习并探索扩散模型与各领域结合的相关知识，增强学术水平和科研能力。

参考文献

[1] SONG J, MENG C, ERMON S. Denoising Diffusion Implicit Models[J]. ArXiv, 2020, abs/2010.02502.

[2] HO J, JAIN A, ABBEEL P. Denoising Diffusion Probabilistic Models[J]. ArXiv, 2020, abs/2006.11239.

[3] NICHOL A, DHARIWAL P. Improved Denoising Diffusion Probabilistic Models[J]. ArXiv, 2021, abs/2102.09672.

[4] RONNEBERGER O, FISCHER P, BROX T. U-Net: Convolutional Networks for Biomedical Image Segmentation[J]. ArXiv, 2015, abs/1505.04597.

[5] SOHL-DICKSTEIN J N, WEISS E A, MAHESWARANATHAN N, et al. Deep Unsupervised Learning using Nonequilibrium Thermodynamics[J]. ArXiv, 2015, abs/1503.03585.

[6] RASUL K, SEWARD C, SCHUSTER I, et al. Autoregressive Denoising Diffusion Models for Multivariate Probabilistic Time Series Forecasting[C]//International Conference on Machine Learning. 2021.