

# Dreamfusion研究报告

姓名：岳东旭

学号：2201212864

指导老师：高伟

## Dreamfusion研究报告

- 1.论文研究什么问题
- 2.论文的主要方法
  - 2.1扩散模型
    - 正向过程
    - 反向过程
    - 小结
  - 2.2 Imagen预训练模型
  - 2.3NeRF网络
    - NeRF简介
    - 三维重建
    - 立体渲染
- 3.Dreamfusion算法流程
  - 3.1随机相机角度和光线采样
  - 3.2渲染
  - 3.3基于视角条件的扩散损失
- 4.实验记录
  - 4.1环境安装及预训练模型下载
  - 4.2模型训练
  - 4.3实验结果
- 5.我的改进
  - 5.1Dreamfusion存在的问题
  - 5.2改进方法
- 6.思考与总结

论文：[DreamFusion: Text-to-3D using 2D Diffusion](#), ICLR 2023

## 1.论文研究什么问题

目前，从文本生成图像的模型非常火热，例如DALLE-2，该领域的突破是由扩散模型（Diffusion model）带动的。

本文主要研究的是如何从文本生成3D数据，主要解决以下两个问题：

1. 该领域没有大量有标签的3D数据集
2. 缺乏有效的3D生成模型

本文的方法不需要任何的3D训练数据，而且无需修改预训练的扩散模型，达到较好效果的同时降低了对计算资源的要求。

Demo：

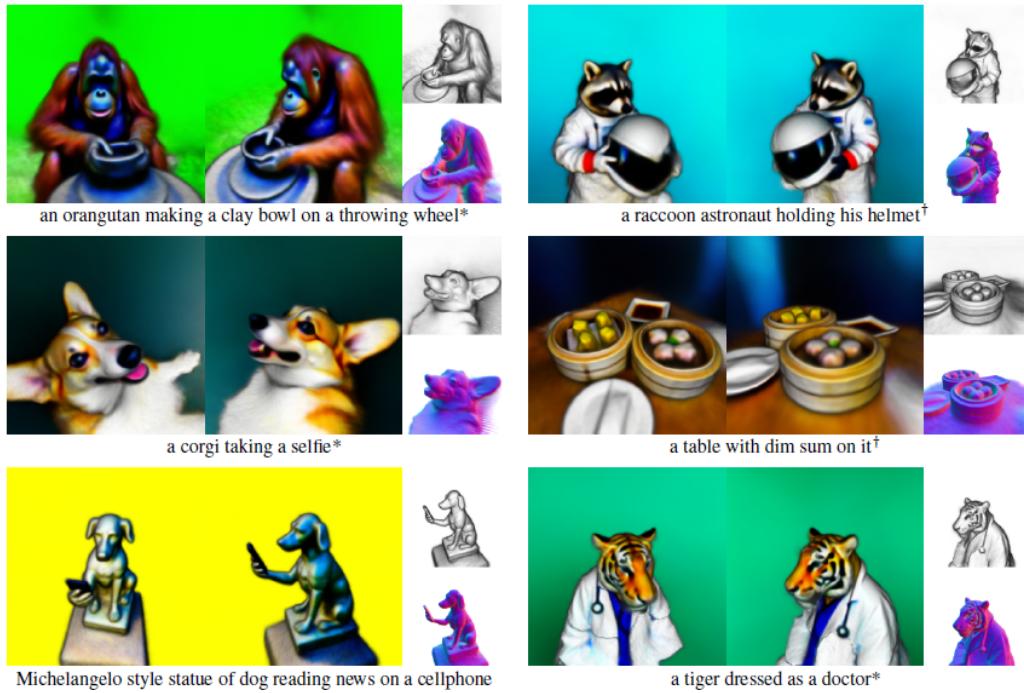


图1.1.1 Dreamfusion生成样例

## 2.论文的主要方法

一句话概括：使用预训练的text to image扩散模型生成的2D图像来进行3D的合成。

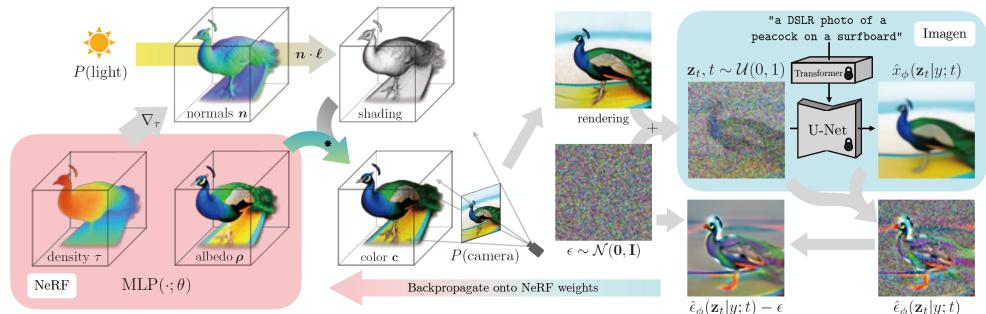


图2 Dreamfusion算法流程

### 2.1 扩散模型

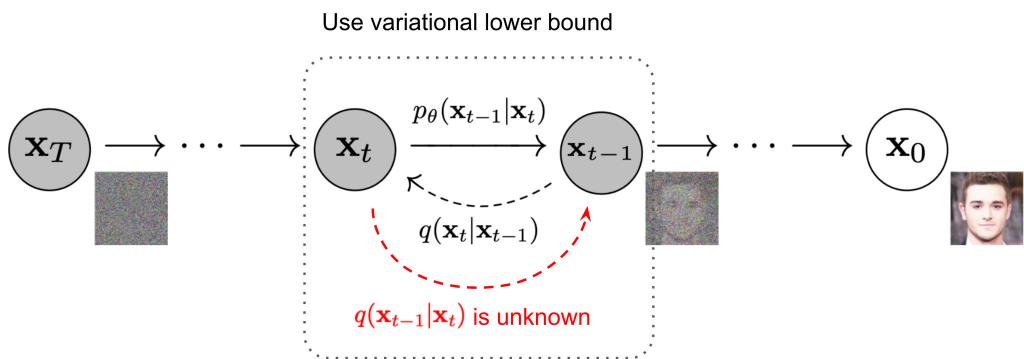


图2.1.1 扩散过程示意

## 正向过程

正向过程是加噪的过程，首先给出的是加噪声的递推公式

可以通过递推公式导出通项公式

参数有约束条件

正向过程中不包含任何可学习的参数，可以通过通项公式快速计算任一时间步的结果

扩散模型正向过程的表达式：

$$\begin{aligned}\mathbf{x}_t &= \alpha_t \mathbf{x}_{t-1} + \beta_t \varepsilon_t \\ &= \alpha_t (\alpha_{t-1} \mathbf{x}_{t-2} + \beta_{t-1} \varepsilon_{t-1}) + \beta_t \varepsilon_t \\ &= \dots \\ &= (\alpha_t \cdots \alpha_1) \mathbf{x}_0 + \underbrace{(\alpha_t \cdots \alpha_2) \beta_1 \varepsilon_1 + (\alpha_t \cdots \alpha_3) \beta_2 \varepsilon_2 + \cdots + \alpha_t \beta_{t-1} \varepsilon_{t-1} + \beta_t \varepsilon_t}_{\text{多个相互独立的正态噪声之和}}\end{aligned}$$

首先，式中花括号所指出的部分，正好是多个独立的正态噪声之和，其均值为 0，方差则分别为  $(\alpha_t \cdots \alpha_2)^2 \beta_1^2, (\alpha_t \cdots \alpha_3)^2 \beta_2^2, \dots, \alpha_t^2 \beta_{t-1}^2, \beta_t^2$ ，

然后，我们利用一个概率论的知识——正态分布的叠加性，即上述多个独立的正态噪声之和的分布，实际上是均值为 0、方差为  $(\alpha_t \cdots \alpha_2)^2 \beta_1^2 + (\alpha_t \cdots \alpha_3)^2 \beta_2^2 + \cdots + \alpha_t^2 \beta_{t-1}^2 + \beta_t^2$  的正态分布；

最后，在  $\alpha_t^2 + \beta_t^2 = 1$  恒成立之下，我们可以得到式(4)的各项系数平方和依旧为 1，即

$$(\alpha_t \cdots \alpha_1)^2 + (\alpha_t \cdots \alpha_2)^2 \beta_1^2 + (\alpha_t \cdots \alpha_3)^2 \beta_2^2 + \cdots + \alpha_t^2 \beta_{t-1}^2 + \beta_t^2 = 1$$

所以实际上相当于有

$$\mathbf{x}_t = \underbrace{(\alpha_t \cdots \alpha_1)}_{\text{记为 } \bar{\alpha}_t} \mathbf{x}_0 + \underbrace{\sqrt{1 - (\alpha_t \cdots \alpha_1)^2} \bar{\varepsilon}_t}_{\text{记为 } \bar{\beta}_t}, \quad \bar{\varepsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

## 反向过程

反向过程即重建过程，首先构造损失函数：

$$\|\mathbf{x}_{t-1} - \mu(\mathbf{x}_t)\|^2$$

进一步改写损失函数为：

$$\|\mathbf{x}_{t-1} - \mu(\mathbf{x}_t)\|^2 = \frac{\beta_t^2}{\alpha_t^2} \|\varepsilon_t - \epsilon_\theta(\mathbf{x}_t, t)\|^2$$

可以先推导出  $x_t$  的表达式：

$$\mathbf{x}_t = \alpha_t \mathbf{x}_{t-1} + \beta_t \varepsilon_t = \alpha_t \left( \bar{\alpha}_{t-1} \mathbf{x}_0 + \bar{\beta}_{t-1} \bar{\varepsilon}_{t-1} \right) + \beta_t \varepsilon_t = \bar{\alpha}_t \mathbf{x}_0 + \alpha_t \bar{\beta}_{t-1} \bar{\varepsilon}_{t-1} + \beta_t \varepsilon_t$$

带入上式，可得到损失函数表达式：

$$\left\| \varepsilon_t - \epsilon_\theta \left( \bar{\alpha}_t \mathbf{x}_0 + \alpha_t \bar{\beta}_{t-1} \bar{\varepsilon}_{t-1} + \beta_t \varepsilon_t, t \right) \right\|^2$$

在此表达式中，包含了两个需要采样的噪声参数，在实际训练时，会导致方差过大产生波动，因此需要考虑将噪声随机变量进行合并。

合并后得到最终的损失函数：

$$\left\| \varepsilon - \frac{\bar{\beta}_t}{\beta_t} \epsilon_\theta \left( \bar{\alpha}_t \mathbf{x}_0 + \bar{\beta}_t \varepsilon, t \right) \right\|^2$$

## 小结

扩散模型包含前向过程和反向过程。

- 前向过程为加噪的过程，其中不包含任何可学习的参数
- 反向过程为去噪的过程，其核心就是要得到一个条件概率分布 $P(x_{t-1} | x_t)$ 的均值和方差，逐步进行推断。

算法流程：

### 1. 模型训练

- 随机采样一批图像数据、随机采样对应数量的时间步、随机采样一个高斯噪声
- 给U-net的主干网络输入图像数据 $x_0$ 、对应数量的时间步 $t$ ，得到网络输出
- 将网络输出与随机生成做对比，得到损失值，进行梯度下降更新网络参数

### 2. 模型推理

- 输入为 $T$ 时刻的高斯噪声（与文本编码的叠加）。
- U-net推理得到前一时间步的结果 $x_{T-1}$ ，依据公式：

$$x_{t-1} = \frac{1}{\alpha_t} (x_t - \beta_t \epsilon_\theta(x_t, t)) + \sigma_t z, \quad z \sim \mathcal{N}(0, I)$$

- 将 $x_{T-1}$ 作为新的输入U-net做进一步推理，不断重复此过程。

## 2.2 Imagen预训练模型

Dreamfusion使用论文《Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding》中的方法，载入Imagen的预训练模型，该模型是基于上述扩散模型的方法，做了进一步改进，可以生成高分辨率的图片。

模型的大致结构如下图所示：

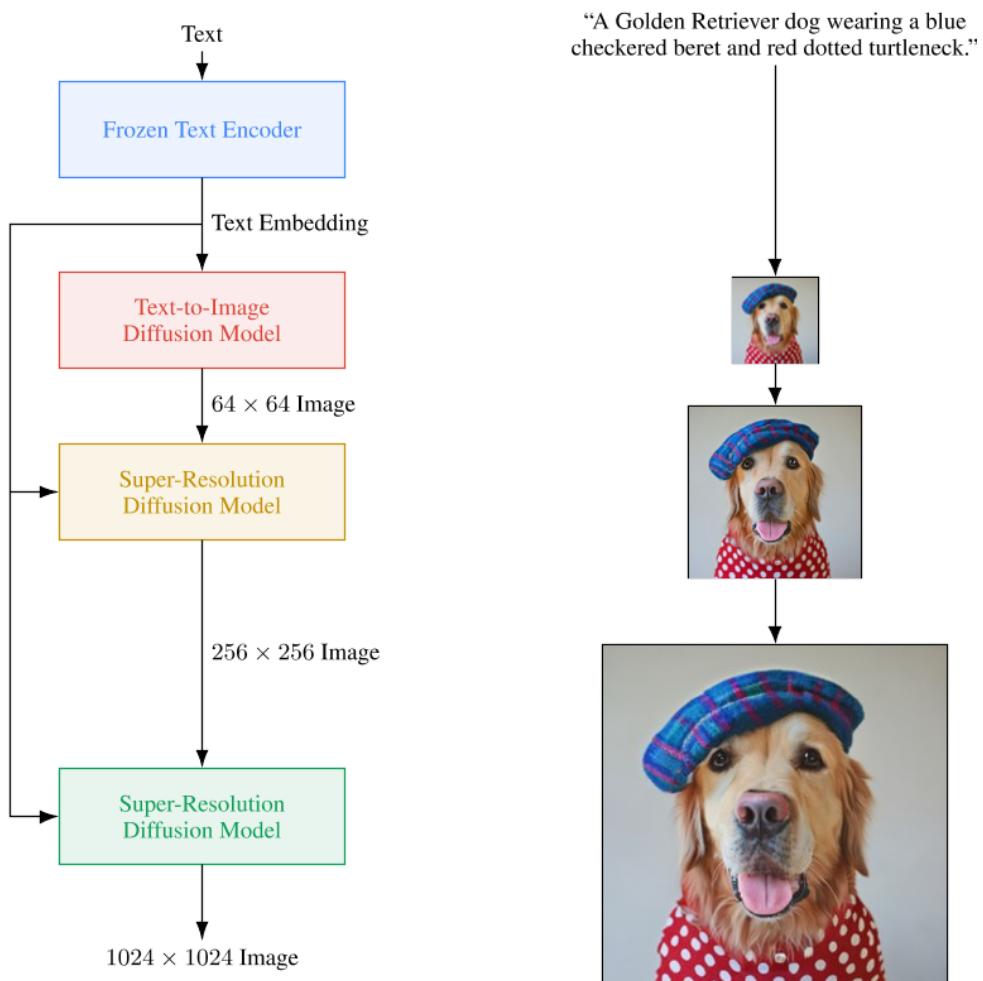


图2.2.1 Imagen模型结构与生成过程

该模型可以在给定文本监督信息的条件下通过扩散模型的方法生成高分辨率的图片。

## 2.3 NeRF网络

### NeRF简介

2020年ECCV最佳论文Neural Radiance Field (NeRF) 将三维场景的隐表示方法推向了新的高度，其本质在于利用神经网络通过多视角2D图像进行3D场景重建，并进行渲染合成新视角的2D图像。

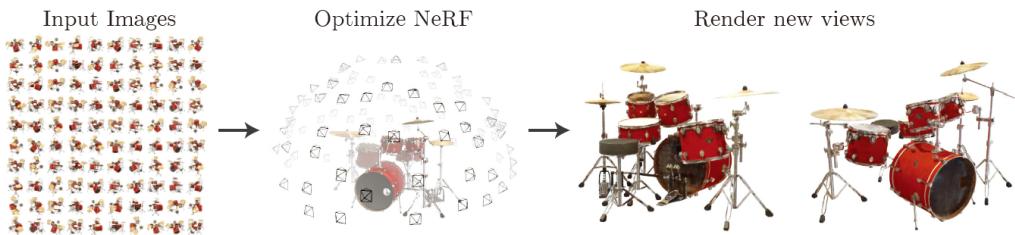


图2.3.1 神经辐射场算法流程

NeRF可以分为两部分：三维重建和渲染。

- **重建**本质上是一个2D到3D的建模过程，NeRF利用3D点的位置及方位作为输入，通过多层感知机（MLP）建模该点对应的颜色及不透明度，形成了3D场景的隐表示。
- **渲染**本质上是一个3D到2D的建模过程，NeRF利用经典的体素渲染将重建部分得到的3D点的颜色及不透明度沿着光线进行整合得到最终的2D图像像素值，在训练的过程中通过与真值做L2损失函数进行网络优化。

### 三维重建

相机内外参数估计：

在重建阶段，我们还需要各个点的位置  $\mathbf{x} = \{x, y, z\}$  和方位  $\mathbf{d} = \{\theta, \phi\}$  作为输入进行重建。

为了获得这些数据，NeRF中采用了传统方法COLMAP进行参数估计。通过COLMAP可以得到场景的稀疏重建结果，其输出文件包括相机内参，相机外参和3D点的信息，然后进一步利用LLFF开源代码中的文件将内外参整合到一个文件中，该文件记录了相机的内参，包括图片分辨率（图片高与宽度）、焦距，共3个维度、外参（包括相机坐标到世界坐标转换的平移矩阵  $\mathbf{t}$  与旋转矩阵  $\mathbf{R}$ ，其中旋转矩阵为  $\mathbf{R} \in \mathbb{R}^{3 \times 3}$  的矩阵，共9个维度，平移矩阵为  $\mathbf{t} \in \mathbb{R}^{3 \times 1}$  的矩阵，3个维度，因此该文件中的数据维度为  $N \times 17$ （另有两个维度为光线的始发深度与终止深度，通过COLMAP输出的3D点位置计算得到），其中  $N$  为图片样本数。

而当我们得到了相机内外参数后，就可以采样光线的起始位置和方向，进而输入一个神经网络预测出条光线上采样点的颜色及不透明度，形成了3D场景的隐表示，如下图所示：

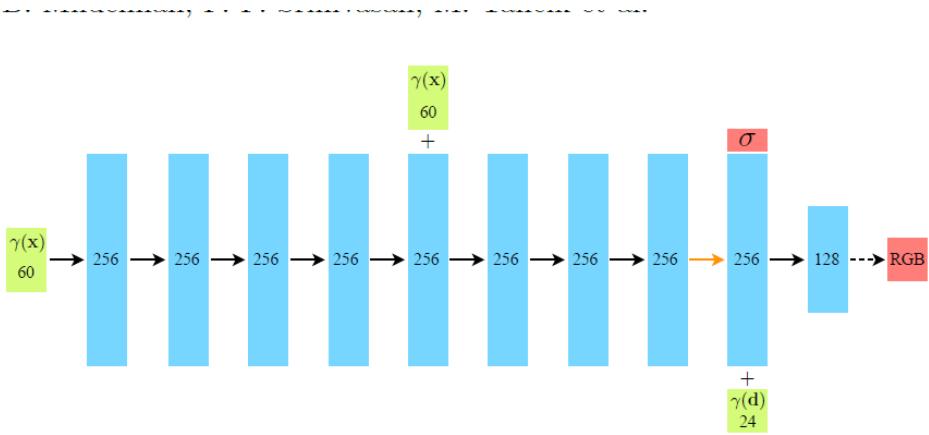


图2.3.2 NeRF所用的神经网络

值得注意的是：

- MLP中带有一个RES连接，在第四层
- 此外，透明度的输出只与位置有关，与方向无关
- 颜色与位置和方向均有关

### 立体渲染

- 连续公式：

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t) \sigma(\mathbf{r}(t)) \mathbf{c}(\mathbf{r}(t), \mathbf{d}) dt, \text{ where } T(t) = \exp \left( - \int_{t_n}^t \sigma(\mathbf{r}(s)) ds \right).$$

- 离散公式

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^N T_i (1 - \exp(-\sigma_i \delta_i)) \mathbf{c}_i, \text{ where } T_i = \exp \left( - \sum_{j=1}^{i-1} \sigma_j \delta_j \right),$$

- 之前的采样算法

$$t_i = \mathbf{U} \left[ t_n + \frac{i-1}{N} (t_f - t_n), t_n + \frac{i}{N} (t_f - t_n) \right]$$

即通过把一条光线上各个采样点的不透明度和颜色做积分，来求得该光线所投影到的二维像素点的颜色。

然后将渲染得到的图片与真实图片做比较即可得到损失函数值，进而可以更新网络参数。

大致算法流程图如下：

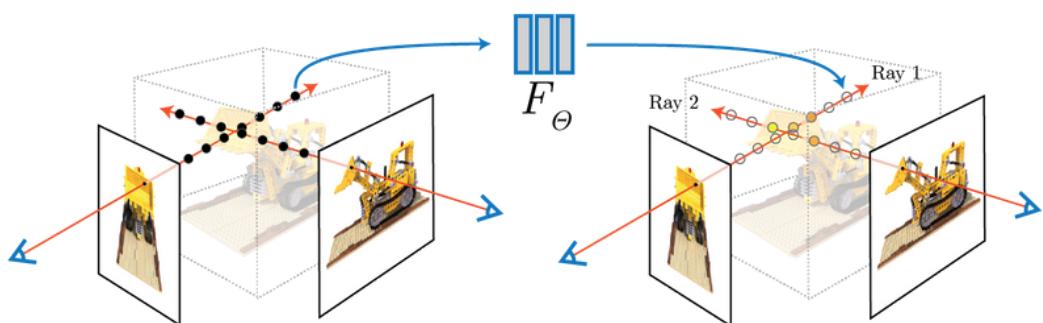


图2.3.3 NeRF渲染过程

### 3.Dreamfusion算法流程

#### 3.1随机相机角度和光线采样

在主程序中，构建数据集即进行随机相机角度采样，使用如下代码：

```
train_loader = NeRFDataset(opt, device=device, type='train', H=opt.h, W=opt.w, size=100).dataloader()

class NeRFDataset:
    def __init__(self, opt, device, type='train', H=256, W=256, size=100):
        super().__init__()

        self.opt = opt
        self.device = device
        self.type = type # train, val, test

        self.H = H
        self.W = W
        self.radius_range = opt.radius_range # default=[1.0, 1.5], help="training camera radius range"
        self.fovy_range = opt.fovy_range # default=[40, 70], help="training camera fovy range"
        self.size = size # 数据集大小

        self.training = self.type in ['train', 'all']

        self.cx = self.H / 2 # 半高，图片中心
        self.cy = self.W / 2 # 半宽，图片中心

    def collate(self, index):

        B = len(index) # always 1因为batchsize设为1

        if self.training:
            # random pose on the fly
            poses, dirs = rand_poses(B, self.device, radius_range=self.radius_range, return_dirs=self.opt.dir_text,
                                      angle_overhead=self.opt.angle_overhead, angle_front=self.opt.angle_front, jitter=self.opt.jitter_pose,
                                      uniform_sphere_rate=self.opt.uniform_sphere_rate)
            # 上述函数返回相机的外参pose，以及观察方向，共分有6个观察方向分类

            # random focal
            fov = random.random() * (self.fovy_range[1] - self.fovy_range[0]) + self.fovy_range[0]
            focal = self.H / (2 * np.tan(np.deg2rad(fov) / 2))
            intrinsics = np.array([focal, focal, self.cx, self.cy])

        else:
            # circle pose
            phi = (index[0] / self.size) * 360
            poses, dirs = circle_poses(self.device, radius=self.radius_range[1] * 1.2, theta=60, phi=phi,
                                         return_dirs=self.opt.dir_text, angle_overhead=self.opt.angle_overhead, angle_front=self.opt.angle_front)

            # fixed focal
            fov = (self.fovy_range[1] + self.fovy_range[0]) / 2
            focal = self.H / (2 * np.tan(np.deg2rad(fov) / 2))
            intrinsics = np.array([focal, focal, self.cx, self.cy])
```

```

# sample a low-resolution but full image for CLIP
rays = get_rays(poses, intrinsics, self.H, self.W, -1) # 用于解算每条光线的起始位置和方向

data = {
    'H': self.H,
    'W': self.W,
    'rays_o': rays['rays_o'],
    'rays_d': rays['rays_d'],
    'dir': dirs, # 分区标签
}

return data

def dataloader(self):
    loader = DataLoader(list(range(self.size)), batch_size=1, collate_fn=self.collate, shuffle=self.training,
    num_workers=0)
    return loader

```

随机相机位姿生成效果：

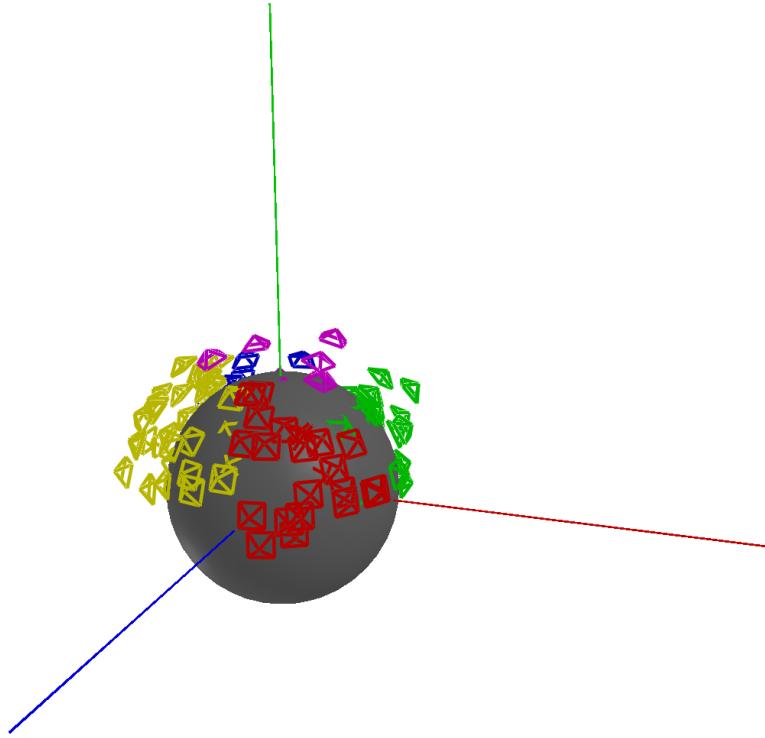


图3.1.1 随机相机位姿采样结果

## 3.2 渲染

将上述随机视角的相机位姿生成对应的光线，每个视角作为一个数据样本，使用传统渲染方法得到二维图像。

```

outputs = self.model.render(rays_o, rays_d, staged=False, perturb=True, bg_color=bg_color, ambient_ratio=ambient_ratio,
shading=shading, force_all_rays=True, **vars(self.opt))

```

### 3.3 基于视角条件的扩散损失

视角条件信息加入：

```
def prepare_text_embeddings(self):

    if self.opt.text is None:
        self.log(f"[WARN] text prompt is not provided.")
        self.text_z = None
        return

    if not self.opt.dir_text:
        self.text_z = self.guidance.get_text_embeds([self.opt.text], [self.opt.negative])
    else:
        self.text_z = []
        for d in ['front', 'side', 'back', 'side', 'overhead', 'bottom']:
            # construct dir-encoded text
            text = f'{self.opt.text}, {d} view'

            negative_text = f'{self.opt.negative}'

            # explicit negative dir-encoded text
            if self.opt.suppress_face:
                if negative_text != "": negative_text += ','

            if d == 'back': negative_text += "face"
            # elif d == 'front': negative_text += ""
            elif d == 'side': negative_text += "face"
            elif d == 'overhead': negative_text += "face"
            elif d == 'bottom': negative_text += "face"

        text_z = self.guidance.get_text_embeds([text], [negative_text])
        self.text_z.append(text_z)
```

即从前、后、边、顶部、底部去编码文本信息，生成的效果：

a cute cat, front view

a cute cat, side view...

...

- 首先根据一个数据样本渲染图片：

```
bg_color = torch.rand((B * N, 3), device=rays_o.device) # pixel-wise random
outputs = self.model.render(rays_o, rays_d, staged=False, perturb=True, bg_color=bg_color,
                            ambient_ratio=ambient_ratio, shading=shading, force_all_rays=True, **vars(self.opt))
pred_rgb = outputs['image'].reshape(B, H, W, 3).permute(0, 3, 1, 2).contiguous() # [1, 3, H, W] # 先推理并渲染图片
```

输入：一个样本中的随机光线的位置和方向

输出：渲染好的RGB图像

- 筛选出本数据样本位姿的文本描述：

```

# text embeddings
if self.opt.dir_text:
    dirs = data['dir'] # [B,]
    text_z = self.text_z[dirs] # 选择出本数据样本对应的位姿
else:
    text_z = self.text_z

```

- 将上述渲染出的RGB图像和文本信息输入到diffusion model:

```

def train_step(self, text_embeddings, pred_rgb, guidance_scale=100):

    # interp to 512x512 to be fed into vae.
    pred_rgb_512 = F.interpolate(pred_rgb, (512, 512), mode='bilinear', align_corners=False)

    # timestep ~ U(0.02, 0.98) to avoid very high/low noise level
    t = torch.randint(self.min_step, self.max_step + 1, [1], dtype=torch.long, device=self.device) # 20-980之间采样

    # encode image into latents with vae, requires grad!
    latents = self.encode_imgs(pred_rgb_512)
    # torch.cuda.synchronize(); print(f'[TIME] guiding: vae enc {time.time() - _t:.4f}s')

    # predict the noise residual with unet, NO grad!
    with torch.no_grad():
        # add noise
        noise = torch.randn_like(latents)
        latents_noisy = self.scheduler.add_noise(latents, noise, t)
        # pred noise
        latent_model_input = torch.cat([latents_noisy] * 2)
        noise_pred = self.unet(latent_model_input, t, encoder_hidden_states=text_embeddings).sample
        # perform guidance (high scale from paper!)
        noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)
        noise_pred = noise_pred_uncond + guidance_scale * (noise_pred_text - noise_pred_uncond)

        # w(t), sigma_t^2
        w = (1 - self.alphas[t])
        # w = self.alphas[t] ** 0.5 * (1 - self.alphas[t])
        grad = w * (noise_pred - noise)

        # clip grad for stable training?
        grad = torch.nan_to_num(grad)

    # manually backward, since we omitted an item in grad and cannot simply autodiff.
    latents.backward(gradient=grad, retain_graph=True)
    # torch.cuda.synchronize(); print(f'[TIME] guiding: backward {time.time() - _t:.4f}s')

    return 0 # dummy loss value

```

在此步骤中，

- 首先对上述图片（64 by 64）进行插值，因为预训练stable-diffusion只能处理512 by 512的图片。
- 生成随机的时间步t
- 将图片编码为隐空间编码
- 将图片编码加上随机高斯噪声

5. 根据加噪后的图片和文字监督信息送入stable-diffusion的U-net，得到去噪的扩散图像隐码表示
6. 手动计算Nerf网络的梯度
7. 手动更新网络参数

## 4.实验记录

### 4.1环境安装及预训练模型下载

- 在服务器上安装部署好环境，下载预训练模型

```
root@6e4e0307b3c2:~/.cache# ls
huggingface/.cache/llm/pip
root@6e4e0307b3c2:~/.cache# cd huggingface/diffusers/models--runwayml--stable-diffusion-v1-5/
root@6e4e0307b3c2:~/cache/huggingface/diffusers/models--runwayml--stable-diffusion-v1-5# ls
blobs  refs  snapshots
root@6e4e0307b3c2:~/cache/huggingface/diffusers/models--runwayml--stable-diffusion-v1-5#
```

```
root@6e4e0307b3c2:~/remote-home/ydk/stable-diffusion] python main.py --text "a hamburger" --workspace trial -o
NameSpace(h=800, w=800, o2=False, w=800, albedo=False, albedo_iters=1000, angle_front=60, angle_overhead=30, backbone='grid', bg_radius=1.4,
        bounces=1, cprg=1, latents=True, ray=True, density_threshold=10, dir_text=True, dt_gamma=0, eval_interval=10, fovy=60, fovy_range=[40, 70], fp16=True,
        gui=False, guidance='stable-diffusion', he64, iters=10000, jitter_pose=False, lambda_entropy=0.0001, lambda_opacity=0, lambda_orient=0.01
        , lambda_smooth=0, light_phi=0, light_theta=60, lr=0.001, max_ray_batch=4096, max_spp=1, max_steps=512, min_near=0.1, negative='',
        num_steps=64, radius=3, radius_range=[1.0, 1.5], save_mesh=False, seed=0, suppress_face=False, test=False, text='a hamburger', uniform_sphere_rate=0.5,
        update_extra_interval=16, upsample_steps=32, w=64, workspace='trial')
NeRFNetwork(
    (encoder): GridEncoder: input_dim=3 num_levels=16 level_dim=2 resolution=16 -> 2048 per_level_scale=1.3819 params=(903480, 2) gridtype=tile
    d_align_corners=False
    (sigma_net): MLP(
        (net): ModuleList(
            (0): Linear(in_features=32, out_features=64, bias=True)
            (1): Linear(in_features=64, out_features=64, bias=True)
            (2): Linear(in_features=64, out_features=4, bias=True)
        )
    )
    (encoder_bg): FreqEncoder: input_dim=3 degree=4 output_dim=27
    (bg_net): MLP(
        (net): ModuleList(
            (0): Linear(in_features=27, out_features=64, bias=True)
            (1): Linear(in_features=64, out_features=3, bias=True)
        )
    )
)
[INFO] try to load hugging face access token from the default place, make sure you have run 'huggingface-cli login'.
[INFO] loading stable diffusion...
```

图4.1.1 环境安装与预训练模型加载截图

- 在Colab上用相同的参数验证

#### Training Settings:

Prompt\_text: " a DSLR photo of a delicious hamburger"

Training\_iters: 5000

Learning\_rate: 1e-3

Training\_nerf\_resolution: 64

Seed: 0

Lambda\_entropy: 1e-4

Max\_steps: 512

Checkpoint: " latest"

#### Output Settings:

Workspace: " trial"

图4.1.2 模型超参数设置

## 4.2模型训练

运行命令：

```
python main.py --text "a hamburger" --workspace trial -O
```

```
[INFO] try to load hugging face access token from the default place, make sure you have run `huggingface-cli login`.
[INFO] loading stable diffusion...
[INFO] loaded stable diffusion!
[INFO] trainer: df | 2022-11-27_14-51-06 | cuda | fp16 | trial-2
[INFO] fparameters: 1015479
[INFO] Loading latest checkpoint ...
[WARN] No checkpoint found, model randomly initialized.
=> Start Training trial-2 Epoch 1, lr=0.010000 ...
loss=0.0000 (0.0000), lr=0.009772: : 100% 100/100 [00:29<00:00,  3.44it/s]
=> Finished Epoch 1.
=> Start Training trial-2 Epoch 2, lr=0.009772 ...
loss=0.0000 (0.0000), lr=0.009761: : 5% 5/100 [00:01<00:24,  3.85it/s]
```

图4.2.1 模型在服务器运行截图

colab训练截图：

```
=> Start Training trial Epoch 43, lr=0.001445 ...
loss=0.0014 (0.0014), lr=0.001380: : 100% 100/100 [01:15<00:00,  1.33it/s]
=> Finished Epoch 43.
=> Start Training trial Epoch 44, lr=0.001380 ...
loss=0.0000 (0.0013), lr=0.001318: : 100% 100/100 [01:13<00:00,  1.35it/s]
=> Finished Epoch 44.
=> Start Training trial Epoch 45, lr=0.001318 ...
loss=0.0015 (0.0013), lr=0.001259: : 100% 100/100 [01:13<00:00,  1.36it/s]
=> Finished Epoch 45.
=> Start Training trial Epoch 46, lr=0.001259 ...
loss=0.0018 (0.0012), lr=0.001202: : 100% 100/100 [01:13<00:00,  1.36it/s]
=> Finished Epoch 46.
=> Start Training trial Epoch 47, lr=0.001202 ...
loss=0.0018 (0.0013), lr=0.001148: : 100% 100/100 [01:14<00:00,  1.35it/s]
=> Finished Epoch 47.
=> Start Training trial Epoch 48, lr=0.001148 ...
loss=0.0016 (0.0013), lr=0.001096: : 100% 100/100 [01:15<00:00,  1.33it/s]
=> Finished Epoch 48.
=> Start Training trial Epoch 49, lr=0.001096 ...
loss=0.0016 (0.0014), lr=0.001047: : 100% 100/100 [01:16<00:00,  1.32it/s]
=> Finished Epoch 49.
=> Start Training trial Epoch 50, lr=0.001047 ...
loss=0.0021 (0.0012), lr=0.001000: : 100% 100/100 [01:13<00:00,  1.36it/s]
=> Finished Epoch 50.
++> Evaluate trial at epoch 50 ...
loss=0.0000 (0.0000): : 100% 5/5 [00:00<00:00,  6.16it/s]
++> Evaluate epoch 50 Finished.
[INFO] training takes 60.5381 minutes.
```

图4.2.2 模型在colab运行截图

由于colab的资源限制，我们只训练50个epoch。

## 4.3实验结果

模型推理生成一段360°旋转的视频：



图4.3.1 模型推理生成的视频演示

## 5.我的改进

本节从文章中存在的问题出发，思考改进解决方案。

### 5.1 Dreamfusion存在的问题

Dreamfusion主要存在的问题就是在某些生成场景下，会生成“多头”的3D模型，如下图所示：

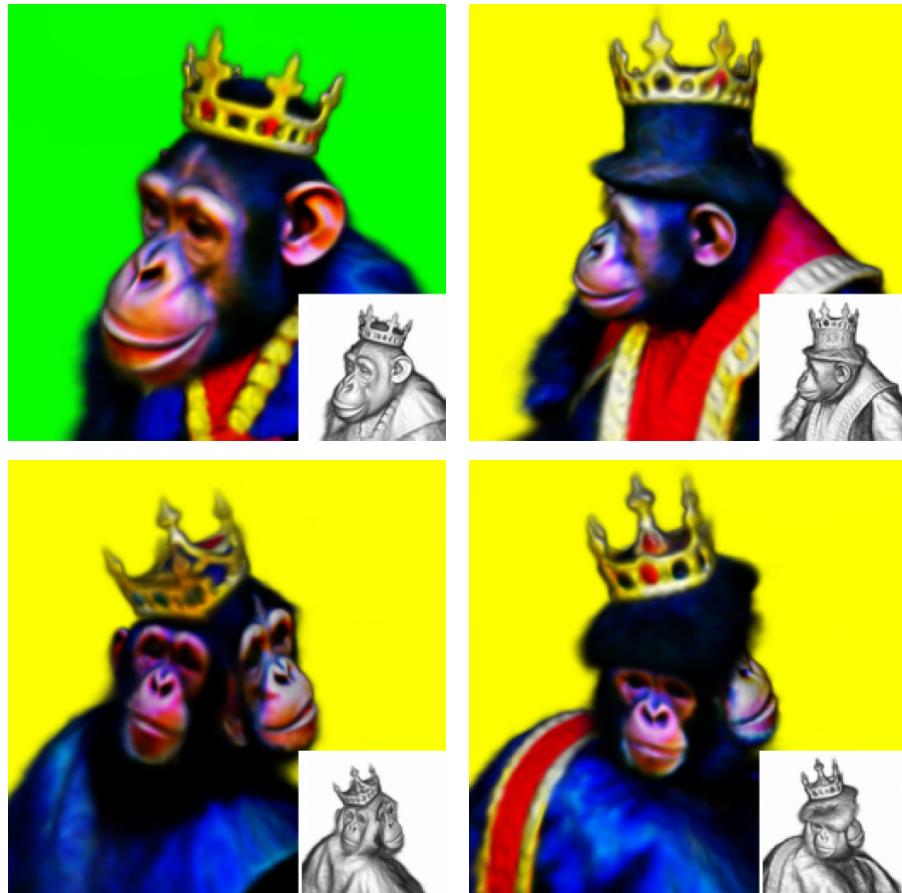


图5.1.1 Dreamfusion存在的问题

从论文源码出发，该问题产生的主要原因是：

- 仅仅使用5个带有方向性的词汇进行监督生成的图片方向太粗糙，不能精细表征出NeRF重建所必要的不同角度的图像信息。
- text2img模型没有对角度控制进行针对性的训练。

### 5.2 改进方法

首先针对第二个问题，我们无能为力，因为从头训练一个Imagen规模的扩散模型的成本是不可估量的（约50w\$+）

因此考虑针对第一个问题进行改进，在生成时使用更细粒度的描述，比如：left side/right side/Slanting top.

尝试生成，效果如下：



图5.2.1 改进生成的实验效果

## 6.思考与总结

在生成模型领域，扩散模型的方法已经成为近来学术界争相研究的热点方法，而文本到3D的多模态生成任务还处在比较初始的阶段，一方面直接训练3D模型的开销可能会过于庞大，另一方面是因为扩散模型在3D数据上的融合还缺乏理论上的完善。

本作业深度调研了Dreamfusion所用到的核心方法：Diffusion和NeRF，并在原模型的基础上提出了一定的改进方法。