Report DL_Lab6_313553024_蘇柏叡

Introduction

在本實驗中，實作了以U-Net作為架構作為基礎的DDPM，其中U-Net中引入了殘差卷積塊、自注意力機制和時間條件嵌入，其概念為利用隨時間逐步消除噪聲的方式並且使用特定的標籤條件以生成圖像。其中本Lab中我使用的Noise Scheduler為DPPM Scheduler，而我們在evaluate上則是使用ResNet18作為計算準確度的pretrained model。至於本次Lab目標則是透過建構一個DPPM並結合上述之技巧試圖在Test、New_Test上提高準確度。

Implementation details

Describe how you implement your model, including your choice of DDPM, noise schedule.

1. Choice of DDPM、Noise Schedule

本Lab參考之文獻為**Denoising Diffusion Probabilistic Models**，我們依據其架構、實驗與參數設定進行實作，以下為論文中呈現之參數與實驗設定截圖：

> We chose the β_t schedule from a set of constant, linear, and quadratic schedules, all constrained so that $L_T \approx 0$. We set $T = 1000$ without a sweep, and we chose a linear schedule from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$.

> We set $T = 1000$ for all experiments so that the number of neural network evaluations needed during sampling matches previous work [53, 55]. We set the forward process variances to constants increasing linearly from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$. These constants were chosen to be small relative to data scaled to $[-1, 1]$, ensuring that reverse and forward processes have approximately the same functional form while keeping the signal-to-noise ratio at $\mathbf{x}_T$ as small as possible ($L_T = D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I})) \approx 10^{-5}$ bits per dimension in our experiments).
>
> To represent the reverse process, we use a U-Net backbone similar to an unmasked PixelCNN++ [52, 48] with group normalization throughout [66]. Parameters are shared across time, which is specified to the network using the Transformer sinusoidal position embedding [60]. We use self-attention at the 16×16 feature map resolution [63, 60]. Details are in Appendix B.

因此，在實驗設定部份除了因為設備限制無法和原論文一致外，其餘部分我們將Noise_Step設定為1000，並且將initial beta設定為1E-4，End beta設定為0.02，並採用linear schedule進行調整。此外，本論文中亦在16*16的feature map上使用了Self-Attention，因此在本Lab中我亦在U-Net架構中加入自注意力機制以求能提升模型處理圖像中細節特徵的能力。

2. Model Implementation

- Residual Convolutional Block:

ResidualConvBlock 包含兩層卷積層，每層卷積之後都緊跟著批歸一化（Batch Normalization）和GELU激活函數。如果輸入和輸出通道數不同，則會使用殘差連接，（1x1卷積來調整通道數），使得殘差連接可以進行有效地相加。也因為有這個特性，因此適合於處理不同尺度的特徵圖，使得模型在下採樣和上採樣過程中都能有效地提取和整合特徵。在forward部分，如果啟用殘差連接，則將輸入與經過卷積的輸出相加，並用sqrt(2)來做normalization，以確保輸出值的穩定性。

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from tqdm import tqdm
from diffusers import DDPMScheduler # to load the scheduler

class ResidualConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, is_residual=False):
        super(ResidualConvBlock, self).__init__()
        self.is_residual = is_residual
        self.same_channels = (in_channels == out_channels)

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

        # if the number of channels is different, use a 1x1 convolution to match the number of channels
        if not self.same_channels:
            self.residual_conv = nn.Conv2d(in_channels, out_channels, 1, 1, 0, bias=False)
```

```python
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if self.is_residual:
            residual = x if self.same_channels else self.residual_conv(x)
            return (residual + self.conv2(self.conv1(x))) / math.sqrt(2) #
        else:
            return self.conv2(self.conv1(x))
```

- **UpSampling、DownSampling、Embedding:**

```python
class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSample, self).__init__()
        self.model = nn.Sequential(
            ResidualConvBlock(in_channels, out_channels),
            nn.MaxPool2d(2)
        )

    def forward(self, x):
        return self.model(x)

class UpSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UpSample, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResidualConvBlock(out_channels, out_channels)
        )

    def forward(self, x, skip):
        x = torch.cat((x, skip), dim=1)
        return self.model(x)
```

```python
class Embed(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super(Embed, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim)
        )

    def forward(self, x):
        return self.model(x.view(-1, self.model[0].in_features))
```

DownSample:透過整合殘差卷積和最大池化層來實現下採樣，在有效減少數據的空間尺寸同時又能增強特徵的提取能力，以利於深層網絡中的資訊傳遞和抽象表達，而在上採樣部分使用轉置卷積和殘差卷積塊來增加特徵圖的空間尺寸，恢復被降維過程中丟失的資訊。並且通過skip connection與先前層的輸出合併，可以保留並恢復圖像的重要細節，增強模型的預測精準度。其中skip connection也是U-Net架構中，極其核心的一部分。Embed的功用則是將輸入的條件訊息轉化為更高維度的嵌入向量。這些嵌入向量使模型能夠根據不同的條件進行調整，以生成對應於這些條件的特定輸出。

-Self Attention

```python
class SelfAttention(nn.Module):
    def __init__(self, in_channels):
        super(SelfAttention, self).__init__()
        self.query = nn.Conv2d(in_channels, in_channels // 8, kernel_size=1)
        self.key = nn.Conv2d(in_channels, in_channels // 8, kernel_size=1)
        self.value = nn.Conv2d(in_channels, in_channels, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        batch_size, C, width, height = x.size()
        proj_query = self.query(x).view(batch_size, -1, width * height).permute(0, 2, 1)  # (B, N, C)
        proj_key = self.key(x).view(batch_size, -1, width * height)  # (B, C, N)
        energy = torch.bmm(proj_query, proj_key)  # (B, N, N)
        attention = F.softmax(energy, dim=-1)  # (B, N, N)
        proj_value = self.value(x).view(batch_size, -1, width * height)  # (B, C, N)

        out = torch.bmm(proj_value, attention.permute(0, 2, 1))  # (B, C, N)
        out = out.view(batch_size, C, width, height)
        out = self.gamma * out + x
        return out
```

首先使用 1x1 的卷積層分別生成query、key、value vectors，這些vectors用來捕捉輸入特徵間的相互關係，並通過內積計算這些特徵之間的相似性得到注意力分數，再通過 softmax 函數進行標準化，並利用這些注意力權重重新組合值向量，結合原始輸入透過學習參數 gamma 融合，從而提取重要特徵並增強模型的表達能力。

- U-Net

```python
class UNet(nn.Module):
    def __init__(self, in_channels, n_feature=64, n_classes=24):
        super(UNet, self).__init__()

        self.in_channels = in_channels
        self.n_feature = n_feature
        self.n_classes = n_classes

        """conv first"""
        self.initial_conv = ResidualConvBlock(in_channels, n_feature, is_residual=True)

        """down sampling"""
        self.down1 = DownSample(n_feature, n_feature)
        self.down2 = DownSample(n_feature, 2 * n_feature)

        """self-attention layer"""
        self.attn1 = SelfAttention(n_feature)
        self.attn2 = SelfAttention(2 * n_feature)

        """bottom hidden of unet"""
        self.hidden = nn.Sequential(nn.AvgPool2d(8), nn.GELU())

        """embed time and condition"""
        self.time_embed1 = Embed(1, 2 * n_feature)
        self.time_embed2 = Embed(1, n_feature)
        self.cond_embed1 = Embed(n_classes, 2 * n_feature)
        self.cond_embed2 = Embed(n_classes, n_feature)
```

```
        """up sampling """
        self.up0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feature, 2 * n_feature, 8, 8),
            nn.GroupNorm(8, 2 * n_feature),
            nn.ReLU(True),
        )
        self.up1 = UpSample(4 * n_feature, n_feature)
        self.up2 = UpSample(2 * n_feature, n_feature)

        """output"""
        self.out = nn.Sequential(
            nn.Conv2d(2 * n_feature, n_feature, 3, 1, 1),
            nn.GroupNorm(8, n_feature),
            nn.ReLU(True),
            nn.Conv2d(n_feature, in_channels, 3, 1, 1),
        )
        self.initialize_weights()
```

```
def initialize_weights(self):
    def init_fn(m):
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight)  # usew kaiming_uniform_ to initialize the weight
            if m.bias is not None:
                m.bias.data.zero_()
        elif isinstance(m, nn.BatchNorm2d):
            m.weight.data.fill_(0.5)  # set BN weight to 0.5
            m.bias.data.fill_(0.1)    # use a different constant to initialize the bias
        elif isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)  #  use xavier_uniform_
            if m.bias is not None:
                m.bias.data.zero_()

    self.apply(init_fn)
```

```python
def forward(self, x, cond, time):
    cond_embed1 = self.cond_embed1(cond).reshape(-1, self.n_feature * 2, 1, 1)
    time_embed1 = self.time_embed1(time).reshape(-1, self.n_feature * 2, 1, 1)
    cond_embed2 = self.cond_embed2(cond).reshape(-1, self.n_feature, 1, 1)
    time_embed2 = self.time_embed2(time).reshape(-1, self.n_feature, 1, 1)

    cond_embed_down1 = self.cond_embed_down1(cond).reshape(-1, self.n_feature, 1, 1)
    time_embed_down1 = self.time_embed_down1(time).reshape(-1, self.n_feature, 1, 1)
    cond_embed_down2 = self.cond_embed_down2(cond).reshape(-1, self.n_feature, 1, 1)
    time_embed_down2 = self.time_embed_down2(time).reshape(-1, self.n_feature, 1, 1)

    x = self.initial_conv(x)
    # down sampling + attention
    down1 = self.down1(cond_embed_down1 * x + time_embed_down1)
    down1 = self.attn1(down1)
    down2 = self.down2(cond_embed_down2 * down1 + time_embed_down2)
    down2 = self.attn2(down2)

    hidden = self.hidden(down2)

    # up sampling
    up1 = self.up0(hidden)
    up2 = self.up1(cond_embed1 * up1 + time_embed1, down2)
    up3 = self.up2(cond_embed2 * up2 + time_embed2, down1)
    # output
    out = self.out(torch.cat((up3, x), 1))
    return out
```

U-Net從一個初始的殘差卷積塊開始處理輸入圖像，並且在每次下採樣後接入自注意力層以增強模型對關鍵特徵的學習，而在嵌入層部分則將時間和條件訊息轉換成適合與特徵圖結合的形式，並且在不同層中reshaped以與特徵圖匹配。此外，參考了[2]對每一層進行權重初始化，而在這邊分別使用了不同的權重初始方式，針對Convolutional Layer來說是使用Kaiming常態分布初始化，其特點是在捲積層使用Kaiming常態分布初始化是基於層的輸入和輸出單元數量自動調整變數的方式。而在Batch normalization時則是將權重初始化至0.5，並且把bias設為0.1(意即在初始化時可以提供非0的梯度)。針對全連接層則是使用Xavier均勻分布方法初始化權重適用於連接層在傳播過程中，防止激活值過大或過小。至於，一張照片進入本U-Net結構之後的tracing process如下:

首先，圖像以[64, 64, 3]的形式輸入，經過ResidualConvBlock後通道數擴增至64，輸出形狀為[64, 64, 64]。接著，經過**第一次DownSample**後，包括一次卷積和一次最大池化，尺寸減半至[32, 32, 64]。

第二次DownSample進一步將通道數增加至128並縮小尺寸至[16, 16, 128]。至於，自注意力層分別應用在兩次下採樣的結果上，不改變尺寸且輸出保持[32, 32, 64]和

[16, 16, 128]。底部隱藏層通過8x8的Average Pooling、GELU後，feature map縮減至[2, 2, 128]。至此開始**進行兩次上採樣**，第一次將特徵圖從[2, 2, 128]上採樣到[4, 4, 128]並與[16, 16, 128]的特徵圖進行結合，通過UpSample後尺寸擴展至[32, 32, 64]。第二次上採樣則是將[32, 32, 64]與[32, 32, 64]的特徵圖進行結合並上採樣到[64, 64, 64]，最終的輸出層將[64, 64, 64]**的特徵圖通過ResidualConvBlock調整回3個通道，尺寸保持為**[64, 64, 3]。

-DDPM

```python
class DDPM(nn.Module):
    def __init__(self, unet_model, betas, noise_steps, device):
        super(DDPM, self).__init__()

        self.n_T = noise_steps
        self.device = device
        self.unet_model = unet_model

        # initialize hte DDPMScheduler
        self.scheduler = DDPMScheduler(
            beta_start=0.0001,
            beta_end=0.02,
            num_train_timesteps=noise_steps
        )
        self.mse_loss = nn.MSELoss()

    def forward(self, x, cond):
        # select a random timestep for each sample
        timestep = torch.randint(0, self.n_T, (x.shape[0],), device=self.device).long()
        noise = torch.randn_like(x)

        # add noise to input
        x_t = self.scheduler.add_noise(x, noise, timestep)

        # use U-Net to predict noise
        predict_noise = self.unet_model(x_t.to(self.device), cond.to(self.device), timestep.float().to(self.device) / self.n_T)

        # calculate loss
        return self.mse_loss(noise, predict_noise)
```

```python
    def sample(self, cond, size):
        n_sample = len(cond)
        x_i = torch.randn(n_sample, *size).to(self.device)  # initial noise

        x_seq = [x_i]

        with tqdm(self.scheduler.timesteps, desc='Sampling Process', total=self.n_T, leave=True, ncols=100, mininterval=0.1) as pbar:
            for t in pbar:
                t = torch.full((n_sample,), t, device=self.device).long()

                # use U-Net to predict noise
                eps = self.unet_model(x_i, cond, t.float().to(self.device) / self.n_T)
                # update noise using DDPM update rule
                x_i = self.scheduler.step(eps, t[0], x_i).prev_sample
                if t[0] % (self.n_T // 10) == 0:
                    x_seq.append(x_i)
                pbar.set_description(f'Sampling: {t[0]}/{self.n_T}')

        return x_i, torch.stack(x_seq, dim=1)
```

一開始，我們先是使用了diffuser裡面的DPPMScheduler，因為原論文設定關係，運氣好的是scheduler裡原先default的都是當初實驗設定，我們其實並不需要額外去定義。如下圖之設定:

DDPMScheduler

class diffusers.DDPMScheduler <source>

```
( num_train_timesteps: int = 1000, beta_start: float = 0.0001, beta_end: float = 0.02,
beta_schedule: str = 'linear', trained_betas: Union = None, variance_type: str = 'fixed_small',
clip_sample: bool = True, prediction_type: str = 'epsilon', thresholding: bool = False,
dynamic_thresholding_ratio: float = 0.995, clip_sample_range: float = 1.0, sample_max_value:
float = 1.0, timestep_spacing: str = 'leading', steps_offset: int = 0, rescale_betas_zero_snr:
bool = False )
```

在每次forward過程中，模型首先為每個條件和時間步驟添加噪聲，並放到DPPMScheduler，再用 U-Net 預測這些噪聲，並且在計算loss function時則是採用MSE(意即實際和預測噪聲之間的均方誤差)來更新模型。Sample function則是會先逐漸將噪音加入圖片中，並且模型逐步進行denoise，已求逐漸還原當初未添加噪音的原始圖片。

3.DataLoader

```python
import os
import json
from PIL import Image
import torch
from torch.utils.data import Dataset
import torchvision.transforms as transforms

def transform_img(img):
    transform = transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    return transform(img)
class CLEVR_DATALOADER(Dataset):
    def __init__(self, mode="train", root=None):
        super().__init__()
        assert mode in ['train', 'test', 'new_test'], "mode should be either 'train', 'test', or 'new_test'"

        json_root = 'C:/Users/蘇柏叡/Desktop/Lab6'
        json_path = os.path.join(json_root, f'{mode}.json')
        objects_path = os.path.join(json_root, 'objects.json')

        # check if the JSON files exist
        assert os.path.exists(json_path), f"JSON file not found: {json_path}"
        assert os.path.exists(objects_path), f"Objects JSON file not found: {objects_path}"

        with open(json_path, 'r') as json_file:
            self.json_data = json.load(json_file)
            print(f"Loaded {mode} data from {json_path}")
```

```
        with open(objects_path, 'r') as json_file:
            self.objects_dict = json.load(json_file)
            print(f"Loaded objects data from {objects_path}")

        if mode == 'train':
            self.img_paths = list(self.json_data.keys())
            self.labels = list(self.json_data.values())
        else:
            self.labels = self.json_data

        self.labels_one_hot = self._generate_one_hot_labels(self.labels, self.objects_dict)
        print(f"Generated one-hot labels for {mode} data")

        self.root = root
        self.mode = mode

    def _generate_one_hot_labels(self, labels, objects_dict):
        num_labels = len(labels)
        num_objects = len(objects_dict)
        labels_one_hot = torch.zeros(num_labels, num_objects)

        for i, label in enumerate(labels):
            indices = [objects_dict[obj] for obj in label]
            labels_one_hot[i, indices] = 1

        return labels_one_hot
```

```
    def __getitem__(self, index):
        if self.mode == 'train':
            img_path = os.path.join(self.root, self.img_paths[index])
            assert os.path.exists(img_path), f"Image not found: {img_path}"
            img = Image.open(img_path).convert('RGB')
            img = transform_img(img)
            label_one_hot = self.labels_one_hot[index]
            return img, label_one_hot

        elif self.mode in ['test', 'new_test']:
            label_one_hot = self.labels_one_hot[index]
            return label_one_hot

if __name__ == '__main__':
    print("Testing train mode dataset:")
    train_dataset = CLEVR_DATALOADER(root='C:/Users/蘇柏叡/Desktop/Lab6/iclevr', mode='train')
    print(f"Train dataset length: {len(train_dataset)}")
    x, y = train_dataset[0]
    print(f"Image shape: {x.shape}, Label shape: {y.shape}")

    print("\nTesting test mode dataset:")
    test_dataset = CLEVR_DATALOADER(root='C:/Users/蘇柏叡/Desktop/Lab6/iclevr', mode='test')
    print(f"Test dataset length: {len(test_dataset)}")
    y = test_dataset[0]
    print(f"Label shape: {y.shape}")
```

CLEVR_DATALOADER 的實現主要涵蓋了圖像預處理、One Hot Encoding生成以及Data Retrieval and Indexing三個關鍵步驟。在預處理階段，圖像被調整大小、轉換為張量並標準化，使其適合用於神經網絡的輸入。在一熱編碼生成過程中，將多標籤分類的標籤轉換為二進制向量，表示圖像中包含的對象。在Data Retrieval and Indexing時會根據數據集的模式（訓練或測試），返回經過處理的圖像和對應的標籤，確保數據在進入模型前已經過適當處理。

4. Training Implementation

```python
import torch
import torch.nn as nn
import torch.optim as optim
import argparse
import numpy as np
import random
from torch.utils.data import DataLoader
import os
from tqdm import tqdm
from dataloader import CLEVR_DATALOADER
from torchvision.utils import save_image, make_grid
from evaluator import evaluation_model
from DDPM import DDPM, UNet

class Trainer():
    def __init__(self, args, device):
        self.args = args
        self.device = device
        self.lr = args.lr
        self.n_epoch = args.epochs
        self.run_name = args.run_name
        # Set the random seed for reproducibility
        self.set_seed(3)
        self._prepare_directories()
        self._prepare_data_loaders()
        self.build_model()
        self.evaluator = evaluation_model()
```

```python
def set_seed(self, seed):
    """Set the random seed for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

def _prepare_directories(self):
    """Prepare directories for saving models and results."""
    os.makedirs(self.args.model_root, exist_ok=True)
    os.makedirs(self.args.result_root, exist_ok=True)

def _prepare_data_loaders(self):
    """Prepare the data loaders for training, testing, and new testing."""
    modes = ['train', 'test', 'new_test']
    loaders = {}

    for mode in modes:
        dataset = CLEVR_DATALOADER(mode, 'C:/Users/蘇柏叡/Desktop/Lab6/iclevr')
        loaders[mode] = DataLoader(dataset, batch_size=self.args.batch_size, shuffle=(mode == 'train'), num_workers=self.args.num_workers)

    # Keep these assignments outside of the function to maintain the original request
    self.train_loader = loaders['train']
    self.test_loader = loaders['test']
    self.new_test_loader = loaders['new_test']
```

首先，引入相關套件，再將dataset分為三種不同資料集(一個訓練、一個Test、一個New_Test)，另外指定random seed確保訓練過程是可以復現。

```python
def build_model(self):
    """Initialize the model and optimizer."""
    unet = UNet(in_channels=3, n_feature=self.args.n_feature, n_classes=24).to(self.device)
    self.ddpm = DDPM(unet_model=unet, betas=(self.args.beta_start, self.args.beta_end),
                     noise_steps=self.args.noise_steps, device=self.device).to(self.device)

def train(self):
    '''
    first initialize some variables to store the best test score and new test score
    then start the training loop with tqdm to show the progress bar
    in each epoch, we first set the model to train mode, then iterate over the train_loader
    then store model weights and calculate the mean loss of the epoch
    '''
    optimizer = optim.Adam(self.ddpm.parameters(), lr=self.lr)
    best_test_epoch, best_new_test_epoch = 0, 0
    best_test_score, best_new_test_score = 0, 0

    for epoch in range(self.n_epoch):
        self.ddpm.train()
        epoch_loss = []  # store the loss of each batch

        with tqdm(self.train_loader, leave=True, ncols=100) as pbar:
            current_lr = optimizer.param_groups[0]['lr']
            pbar.set_description(f"Epoch {epoch}, lr: {current_lr:.4e}")
            for input, cond in pbar:
                optimizer.zero_grad()
                input = input.to(self.device)  # [batch, 3, 64, 64]
                cond = cond.to(self.device)  # [batch, 24]
                loss = self.ddpm(input, cond)
```

```python
                cond = cond.to(self.device)  # [batch, 24]
                loss = self.ddpm(input, cond)
                loss.backward()
                pbar.set_postfix_str(f"loss: {loss:.4f}")
                optimizer.step()

                epoch_loss.append(loss.item())
            pbar.close()

        mean_epoch_loss = sum(epoch_loss) / len(epoch_loss)
        print(f"Epoch {epoch} Mean Training Loss: {mean_epoch_loss:.4f}")

        """testing time"""
        test_score, new_test_score = self.save(epoch)

        # update and save the best test score model
        if test_score > best_test_score:
            best_test_score = test_score
            best_test_epoch = epoch

        # update and save the best new test score model
        if new_test_score > best_new_test_score:
            best_new_test_score = new_test_score
            best_new_test_epoch = epoch
```

首先先初始化U-Net、DDPM model，並且依據DDPM.py撰寫需要傳入的超參數填入。接著開始計算loss以及撰寫存取權重檔案更新的條件式，並且輸出該Epoch計算出Test、New_Test的Score。下頁截圖則是會有更新學習率，有嘗試過線性降低學習率、使用ReduceLR 但需要設置如test+new_test的score越高越好，且若超過n個epoch未提升，則*factor。不過最終仍以線性降低學習率方法作為本次Lab實作方式。

```
        #   save the model with test score and new test score both over 0.6
        if test_score > 0.6 and new_test_score > 0.6:
            save_path = os.path.join(self.args.model_root, f"wanna_try_{epoch}_test{test_score:.4f}_new_test{new_test_score:.4f}.pth")
            torch.save(self.ddpm.state_dict(), save_path)

        # print current score and best score
        print(f"Epoch {epoch}: Test Score: {test_score:.4f}, New Test Score: {new_test_score:.4f}")
        print(f"Best Test Score: {best_test_score:.4f} (Epoch {best_test_epoch}), Best New Test Score: {best_new_test_score:.4f} (Epoch {best_ne

        # reduce learning rate after each epoch
        # lr_scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
        #                                                     factor=0.9, patience=3,
        #                                                     min_lr=0)
        new_lr = max(optimizer.param_groups[0]['lr'] - 2.5e-6, 0)
        for param_group in optimizer.param_groups:
            param_group['lr'] = new_lr

    def save(self, epoch):
        test_score, grid1, _ = self.test(self.test_loader)
        test_new_score, grid2, _ = self.test(self.new_test_loader)

        path1 = os.path.join(self.args.result_root, f"test_{epoch}.png")
        path2 = os.path.join(self.args.result_root, f"test_new_{epoch}.png")

        save_image(grid1, path1)
        save_image(grid2, path2)

        return test_score, test_new_score
```

```
def test(self, test_loader):
    self.ddpm.eval()
    input_gen, label = [], []
    with torch.no_grad():
        for cond in test_loader:
            cond = cond.to(self.device)
            input_fin, _ = self.ddpm.sample(cond, (3, 64, 64))
            input_gen.append(input_fin)
            label.append(cond)
        input_gen = torch.cat(input_gen)
        label = torch.stack(label, dim=0).squeeze()
        score = self.evaluator.eval(input_gen, label)
        grid = make_grid(input_gen, nrow=8, normalize=True)

        cond = torch.zeros((1, 24), device=self.device)
        # random_indices = random.sample(range(24), 6)  # 隨機選擇6個不同的索引
        # for idx in random_indices:
        #     cond[0, idx] = 1  # 將選擇的索引位置設置為1
        cond[0, 9], cond[0, 7], cond[0, 22] = 1, 1, 1  # red sphere, yellow cube, cyan cylinder
        _, input_seq = self.ddpm.sample(cond, (3, 64, 64))
        grid_seq = make_grid(input_seq, nrow=5, normalize=True, scale_each=True)

        # Save the final denoised image to evaluate clarity
        final_image = (input_seq[-1] + 1) / 2  # adjust the range of the image to [0, 1]
        save_image(final_image, os.path.join(self.args.result_root, "final_denoised_image.png"))

    return score, grid, grid_seq
```

這邊test則是進行評估，切換為eval模式，並且根據spec則是先指定條件為red sphere、yellow cube、cyan cylinder去循環遍歷並生成對應之圖案(採用make_grid產生出生成圖)以及將denoise過程慢慢輸出，跳出迴圈之外時會把累積的image、label放到input_gen、label內，並做拼接。在test最後則是會輸出score、生成圖案grid、圖像序列(grid_seq)。

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--lr", default=1e-3, type=float)
    parser.add_argument("--save_path", default='ckpt')
    parser.add_argument("--run_name", default='ddpm')
    parser.add_argument("--inference", default=True, action='store_true')
    parser.add_argument('--batch_size', default=32, type=int)
    parser.add_argument('--num_workers', default=6, type=int)
    parser.add_argument('--epochs', default=300, type=int)

    parser.add_argument('--beta_start', default=1e-4, type=float, help='start beta value')
    parser.add_argument('--beta_end', default=0.02, type=float, help='end beta value')
    parser.add_argument('--noise_steps', default=1000, type=int, help='frequency of sampling')
    parser.add_argument("--img_size", default=64, type=int, help='image size')
    parser.add_argument('--n_feature', default=64, type=int,
                        help='time/condition embedding and feature maps dimension')#64 cuz 128 and 256 are too large for my GPU...
    parser.add_argument('--resume', default="over_60_epoch298_test0.6806_new_test0.7500.pth", action='store_true', help='resume training')

    parser.add_argument("--dataset_path", default="iclevr", type=str, help="root of dataset dir")
    parser.add_argument("--model_root", default="ckpt", type=str, help="model ckpt path")
    parser.add_argument("--result_root", default="results", type=str, help="save img path")
    args = parser.parse_args()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    trainer = Trainer(args, device)
```

```python
    if args.inference:
        print("Running inference...")
        #path = os.path.join(args.model_root, "over_60_epoch298_test0.6806_new_test0.7500.pth")
        path = os.path.join(args.model_root, "latest_epoch71.pth")
        #
        trainer.ddpm.load_state_dict(torch.load(path))
        test_score, grid_test, grid_process = trainer.test(trainer.test_loader)
        # Save test images
        path = os.path.join(args.result_root, "___test___.png")

        save_image(grid_test, path)
        #denoise_process(trainer.ddpm, trainer.test_loader, args, save_prefix='test')


        test_new_score, grid_test_new, _ = trainer.test(trainer.new_test_loader)
        # Save new test images
        path = os.path.join(args.result_root, "___test_new___.png")
        save_image(grid_test_new, path)
        #denoise_process(trainer.ddpm, trainer.new_test_loader, args, save_prefix='new_test')

        print("Test accuracy: {:.4f}, New test accuracy: {:.4f}".format(test_score, test_new_score))
        print("Inference done")
    elif not args.inference:
        if args.resume:
            print("Resume training...")
            path = os.path.join(args.model_root, "over_60_epoch298_test0.6806_new_test0.7500.pth")
            trainer.ddpm.load_state_dict(torch.load(path))
        print("Start training...")
        trainer.train()
```

本頁上圖則是會運用到的args，其中我們將會使用--inference進行推斷，而在推斷的過程中我們一樣會load權重檔案，並且針對其輸出進行評估(score、生成圖片、降噪過程圖)，此外若非inference且為繼續訓練的話，則會接續著load進去的權重檔案繼續訓練。

Command Line:

Training時，直接執行trainer.py即可

Inference時，先於inference處插入權重檔案路徑如下圖:

```python
#args.inference = True
if args.inference:
    print("Running inference...")

    path = os.path.join(args.model_root, "DL_lab6_313553024_蘇柏叡.pth")
    #
    trainer.ddpm.load_state_dict(torch.load(path))
    test_score, grid_test, grid_process = trainer.test(trainer.test_loader)
    # Save test images
    path = os.path.join(args.result_root, "___test___.png")

    save_image(grid_test, path)
```

並且在我的終端機是直接key C:/Users/蘇柏叡/Desktop/DL_Lab6_313553024_蘇柏叡/Trainer.py --inference 即可

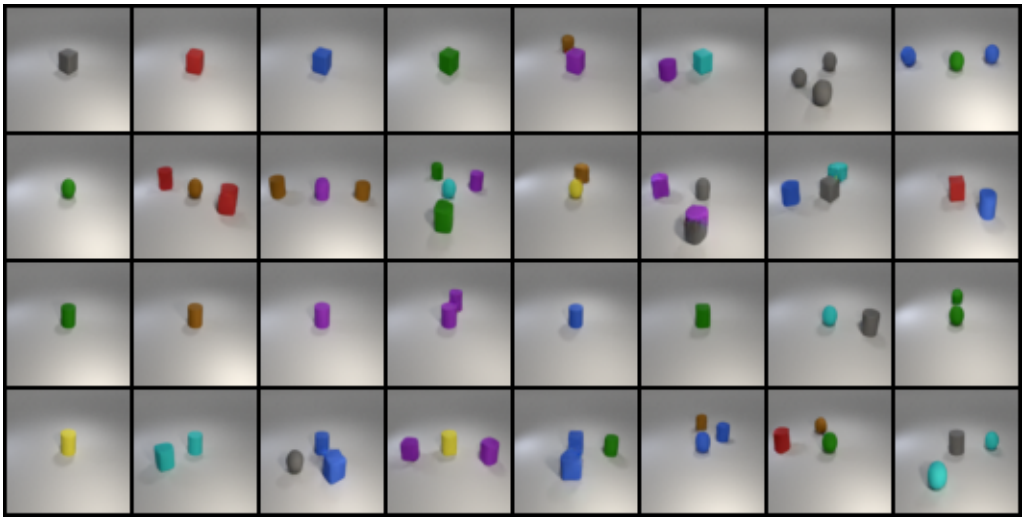
會有三種圖片輸出

Results and discussion

Show your synthetic image grids and a denoising process image

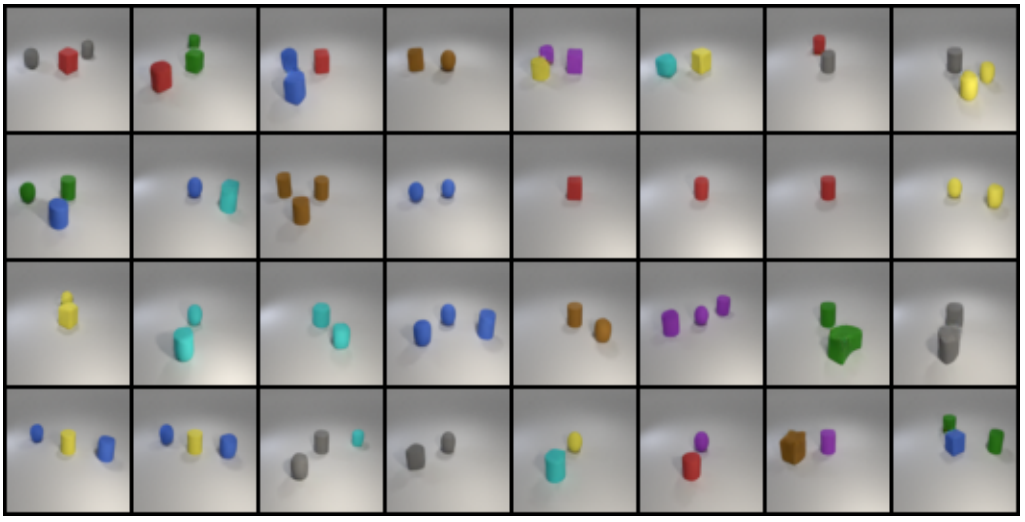
1. synthetic image grids

a. test data:



本圖在(6,3)(以左下角作為(0,0))處有較為嚴重之誤判，出現了灰色圓柱體但卻有紫色的頂面。而我們仔細會發現在輸出的圖中，方形體、圓柱體的形狀上有較為精準的輪廓，但在圓球體部分卻有不少張呈現了橢圓狀。此外我們亦發現當物體靠較近時(即出現類別間、類別內遮蔽時)在圖片的生成也較容易出現誤判以及變形的情況(5,1)。

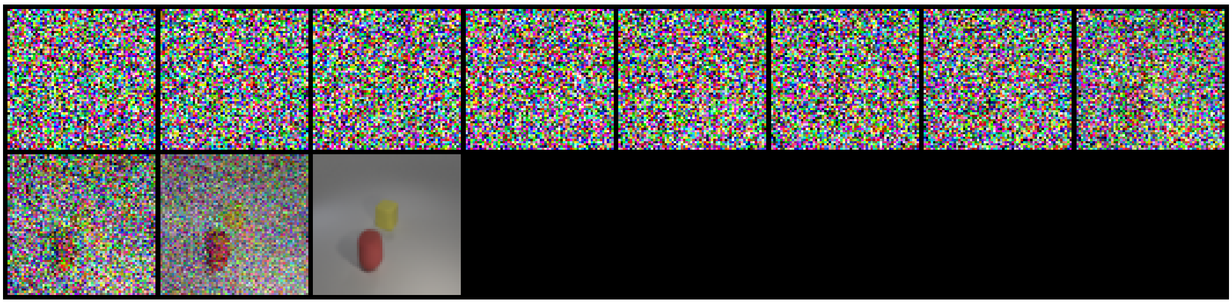
b. new test data:



本圖可發現依然存在圓球體呈現橢圓狀之問題，此外上述提到的類別內、間遮蔽的問題也是存在。至於生成上雖大致上輪廓清楚，但偶爾也是會出現不規則形狀，如(7,2)綠色疑似方塊的物體。

c. Inference Result(Best I’ve tried): (Test:66.67%、New_Test: 76.19%)

2. denoising process image:



本張圖可以看出在降躁的過程中，noise會逐漸被消除，其中最後一張圖則是消除後所生成之圖像。

Discussion of your extra implementations or experiments

1. 使用6種隨機條件進行生成，程式碼如註解掉部分

```python
def test(self, test_loader):
    self.ddpm.eval()
    input_gen, label = [], []
    with torch.no_grad():
        for cond in test_loader:
            cond = cond.to(self.device)
            input_fin, _ = self.ddpm.sample(cond, (3, 64, 64))
            input_gen.append(input_fin)
            label.append(cond)
        input_gen = torch.cat(input_gen)
        label = torch.stack(label, dim=0).squeeze()
        score = self.evaluator.eval(input_gen, label)
        grid = make_grid(input_gen, nrow=8, normalize=True)

        cond = torch.zeros((1, 24), device=self.device)
        # random_indices = random.sample(range(24), 6)  # 隨機選擇6個不同的索引
        # for idx in random_indices:
        #     cond[0, idx] = 1  # 將選擇的索引位置設置為1
        cond[0, 9], cond[0, 7], cond[0, 22] = 1, 1, 1  # red sphere, yellow cube, cya
        _, input_seq = self.ddpm.sample(cond, (3, 64, 64))
        grid_seq = make_grid(input_seq, nrow=5, normalize=True, scale_each=True)
```

效果是相較於固定三種的方式而言並沒有比較優勢。

```
Running inference...
Sampling: 0/1000: 100%|          | 1000/1000 [00:59<00:00, 16.71it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:14<00:00, 67.36it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:52<00:00, 18.95it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:14<00:00, 69.19it/s]
Test accuracy: 0.6528, New test accuracy: 0.7500
Inference done
```

```
Running inference...
Sampling: 0/1000: 100%|          | 1000/1000 [00:59<00:00, 16.84it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:14<00:00, 69.66it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:52<00:00, 18.94it/s]
Sampling: 0/1000: 100%|          | 1000/1000 [00:14<00:00, 70.53it/s]
Test accuracy: 0.6111, New test accuracy: 0.7500
Inference done
```

2. 使用Deformable Convolution試圖增加面對較多遮蔽、圓球體等形狀進行優化。然而，雖然有實作如下圖在UpSample、DownSample時採用，但效果極不佳且極度耗資源。

```python
class DeformableConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, padding=1, stride=1):
        super(DeformableConvBlock, self).__init__()
        self.offset_conv = nn.Conv2d(in_channels, 2 * kernel_size * kernel_size,
                                     kernel_size=kernel_size, stride=stride, padding=padding)
        self.deform_conv = DeformConv2d(in_channels, out_channels, kernel_size=kernel_size,
                                        stride=stride, padding=padding, bias=False)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        offset = self.offset_conv(x)
        x = self.deform_conv(x, offset)
        x = self.bn(x)
        return self.relu(x)
```

分析可能原因為，因為本身尺寸已經resize成64*64即足夠小了，換言之，幾何空間的不規則性會被resize後而簡化，使得變形捲積無法發揮其優勢，再者使用變形捲積計算偏移量時可能會因為本身尺寸小導致偏移量計算上有嚴重的偏誤。

Others:

__pycache__	2024/8/29 上午 09:01	檔案資料夾	
ckpt	2024/8/29 上午 08:34	檔案資料夾	
iclevr	2024/8/16 下午 08:08	檔案資料夾	
results	2024/8/29 上午 08:38	檔案資料夾	
checkpoint.pth	2024/8/16 下午 03:22	PTH 檔案	43,766 KB
dataloader	2024/8/29 上午 05:59	JetBrains PyChar...	4 KB
DDPM	2024/8/29 上午 09:01	JetBrains PyChar...	10 KB
evaluator	2024/8/16 下午 03:22	JetBrains PyChar...	3 KB
new_test	2024/8/16 下午 03:22	JSON 來源檔案	2 KB
objects	2024/8/16 下午 03:22	JSON 來源檔案	1 KB
readme	2024/8/16 下午 03:22	文字文件	2 KB
test	2024/8/16 下午 03:22	JSON 來源檔案	2 KB
train	2024/8/16 下午 03:22	JSON 來源檔案	1,077 KB
Trainer	2024/8/29 上午 08:35	JetBrains PyChar...	11 KB

檔案結構如上，會將權重檔案存於ckpt folder，並且將結果圖存於results folder

reference:

[1]
Nichol, A.Q. & Dhariwal, P.. (2021). Improved Denoising Diffusion Probabilistic Models. Proceedings of the 38th International Conference on Machine Learning, in Proceedings of Machine Learning Research 139:8162-8171 Available from https://proceedings.mlr.press/v139/nichol21a.html.

[2]
https://github.com/labmlai/annotated_deep_learning_paper_implementations?tab=readme-ov-file