# Lab6 313605019 方敏

## 1. Introduction

In this project, we implement a Conditional Denoising Diffusion Probabilistic Model (DDPM) to generate images based on the i-CLEVR dataset. The goal is to create realistic 64×64 RGB images from given object descriptions. We use one-hot encoded labels to help the model generate images that match the desired objects.

We choose the DDPM framework because it is stable during training and can generate high-quality and diverse images. The model uses a U-Net architecture and is trained to remove noise from images step by step. The input includes noisy image, time step, and the label information. During sampling, the model starts from pure noise and slowly denoises it to generate an image based on the given labels.

We use a pretrained ResNet18 classifier from given evaluator.py to evaluate the accuracy of the generated images. Output classification accuracy, image grids, and denoising process images to demonstrate the model's performance.

## 2. Implementation details

### A. dataset.py

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

with open('objects.json', 'r') as f:
    OBJ2IDX = json.load(f) # { "red cube": 0, "blue sphere": 1}
IDX2OBJ = {v: k for k, v in OBJ2IDX.items()} # { 0: "red cube", 1: "blue sphere"}
NUM_CLASSES = len(OBJ2IDX)

def labels_to_onehot(label_list): # one-hot encoding
    onehot = torch.zeros(NUM_CLASSES)
    for obj in label_list:
        onehot[OBJ2IDX[obj]] = 1
    return onehot

class CLEVRDataset(Dataset):
    def __init__(self, json_path, img_folder=None, transform=None):
        with open(json_path, 'r') as f:
            data = json.load(f)
        if isinstance(data, dict): # train的格式
            self.samples = list(data.items())
        else:
            self.samples = data # test的格式
        self.img_folder = img_folder
        self.transform = transform

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        if isinstance(self.samples[idx], tuple): # train
            img_name, labels = self.samples[idx]
            img_path = os.path.join(self.img_folder, img_name)
            img = Image.open(img_path).convert('RGB')
        else:
            labels = self.samples[idx] # test
            img = Image.new('RGB', (64, 64), (0, 0, 0))  # 黑色空白圖像
        if self.transform:
            img = self.transform(img)
        label_onehot = labels_to_onehot(labels)
        return img, label_onehot
```

In this project, we use the i-CLEVR dataset. It contains three JSON files: train.json, test.json, and new_test.json. train.json is a dictionary, each entry links a filename to a list of object labels. test.json and new_test.json are lists, each only contains object labels without any images.

I created a custom class called CLEVRDataset to load and process files. The class checks if the input is training or testing data. For training, it loads both the image and its corresponding labels; for testing, since there are no images, it first generate a blank 64×64 black image as a placeholder.

Each label is a string like "red cube" or "blue sphere". I use labels_to_onehot() to convert labels into one-hot vectors This function maps each object name to an index defined in objects.json and creates a 24-dimensional one-hot tensor. So if the label list is ["red cube", "blue sphere"], the output will have two 1s at the corresponding positions and 0s at all other places.

B. utils.py

```python
import torch
from torchvision import transforms
from evaluator import evaluation_model

def denormalize(x):
    # 原x: (batch, 3, 64, 64), [-1, 1]
    return (x.clamp(-1, 1) + 1) / 2 # 將影像資料從[-1, 1]轉回[0, 1]區間（RGB顯示用）

def evaluate_images(images, labels):
    # images: (batch, 3, 64, 64), labels: (batch, 24)
    norm = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # 讓圖片從[0, 1]轉為[-1, 1]
    images = norm(images)
    evaluator = evaluation_model() # 呼叫evaluator
    acc = evaluator.eval(images.cuda(), labels.cuda())
    print(f"Accuracy: {acc:.3f}")
    return acc
```

It includes two main helper functions that are important for image evaluation. The first function denormalize() is used to convert images from range [-1, 1] back to [0, 1], to correspond standard range for RGB images.

The second function evaluate_images() is used to evaluate the generated images using a pretrained classifier( called by eval.py). Since the classifier expects images in the [-1, 1] range, I normalize the input using transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), which will map [0, 1] images to [-1, 1]. Then load the evaluator model (based on ResNet18 that given by TAs) and compute the classification accuracy by comparing the predicted labels with the provided one-hot labels.

C. train.py

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
from dataset import CLEVRDataset, NUM_CLASSES
from model import ConditionalUNet
from diffusion import DiffusionSchedule, q_sample
from tqdm import tqdm

def train():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(device)
    transform = transforms.Compose([ # [0, 1] -> [-1, 1]
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])
    trainset = CLEVRDataset('train.json', img_folder='train_images', transform=transform)
    trainloader = DataLoader(trainset, batch_size=64, shuffle=True, num_workers=4)
    model = ConditionalUNet(num_classes=NUM_CLASSES).to(device)
    schedule = DiffusionSchedule() # 建立diffusion時間表(包含 β, α 值)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    mse = nn.MSELoss()
    epochs = 5000
    best_loss = float('inf')
    loss_list = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for img, label in tqdm(trainloader, desc=f"Epoch {epoch+1}/{epochs}", leave=False): # tqdm顯示進度條
            img = img.to(device)
            label = label.to(device)
            t = torch.randint(0, schedule.timesteps, (img.size(0),), device=device) #  隨機生成時間步數t，決定加多少noise
            noise = torch.randn_like(img)
            noisy_img = q_sample(img, t, noise, schedule.alpha_bars.to(device)) # 用q_sample()把noise加到圖片上
            pred_noise = model(noisy_img, t, label) # 預測noisy image原本加了多少噪聲noise
            loss = mse(pred_noise, noise)
            optimizer.zero_grad() # 清空梯度
            loss.backward() # 反向傳播
            optimizer.step() # 更新參數
            running_loss += loss.item()
        avg_loss = running_loss / len(trainloader)
        loss_list.append(avg_loss)
        print(f"Epoch {epoch+1}: Loss={avg_loss:.4f}")
        # 每50epoch存最佳模型
        if (epoch + 1) % 50 == 0:
            if avg_loss < best_loss:
                best_loss = avg_loss
                torch.save(model.state_dict(), f'best_model_epoch{epoch+1}.pth') #  loss是最佳就儲存
                print(f"Best model saved at epoch {epoch+1} with loss {best_loss:.4f}")

    torch.save(model.state_dict(), 'ddpm_ckpt.pth')
    # 畫loss圖
    plt.figure(figsize=(10,5))
    plt.title("Training Loss Curve")
    plt.plot(range(1, epochs+1), loss_list, label="train loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig("loss.png", dpi=300)
    plt.close()

if __name__ == '__main__':
    train()
```

To train Conditional DDPM, First I apply a preprocessing pipeline to each image by resizing to 64×64, converting to a tensor, and normalizing the pixel values to the range of [-1, 1], to fit both model and the pretrained evaluator. Then I called CLEVRDataset class from dataset.py to load the data from train.json.

My model is a Conditional UNet, a modified version of the original

UNet, it needs three inputs:

- a noisy image (simulating how a clean image becomes noisy in the forward diffusion process)
- a timestep t (how much noise has been added)
- a one-hot encoded label vector (24-dimensional vector)

During training, I select a timestep t by random for each image then add corresponding noise using q_sample(), and feed the noisy image, timestep, label into the model. The model would try to predict the noise that was added. Then compute the MSE loss between the predicted and actual noise, and use backpropagation to update the model parameters.

I train for a total of 5000 epochs. Every 50 epochs, save the model checkpoint if it has the lowest loss so far. After training, also save the final model and plot a loss curve to visualize training progress.

D. model.py

i.

```python
import torch
import torch.nn as nn

class TimeEmbedding(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.lin = nn.Sequential(
            nn.Linear(1, dim),
            nn.ReLU(),
            nn.Linear(dim, dim)
        )
    def forward(self, t):
        t = t.float().unsqueeze(-1) / 1000 # default: 1000/1000，不爆梯度
        return self.lin(t) # 擴充維度變成128層，embedding讓模型學複雜特徵

class LabelEmbedding(nn.Module): # 將one-hot label轉成跟時間一樣維度(128)
    def __init__(self, num_classes, dim):
        super().__init__()
        self.lin = nn.Sequential(
            nn.Linear(num_classes, dim),
            nn.ReLU(),
            nn.Linear(dim, dim)
        )
    def forward(self, y):
        return self.lin(y)
```

The TimeEmbedding class is used to convert each timestep (ranging from 0 to 1000) into an embedding vector, it can let the model understand which stage of the denoising process that the image is currently at. First divide the timestep by 1000 to scale it down and prevent gradient explosion, then pass it through two Linear layers and ReLU activation to obtain a 128-dimensional time representation vector.

Similarly, the LabelEmbedding class transforms the one-hot encoded label of each image (24 dimensions) into a 128-dimensional conditional vector (same as time embedding). Lebel embedding and time embedding will be fed into the model as a condition to help it generate images that match the given object descriptions.

ii.

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.block = nn.Sequential( # 兩層convolution + batchnorm 一層ReLU
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
        )
        # 如果輸入和輸出維度不同，用1x1卷積做shortcut，否則輸入輸出相加
        self.shortcut = nn.Conv2d(in_channels, out_channels, 1) if in_channels != out_channels else nn.Identity()
    def forward(self, x): # 讓差異加上源資訊
        return nn.ReLU(inplace=True)(self.block(x) + self.shortcut(x)) # 再經過一個ReLU激活
```

The ResidualBlock is a feature extraction module that comes from the residual structure in ResNet. Its main purpose is to help deep neural networks train more stably by avoiding the vanishing gradient problem. This block consists of two convolutional layer, BatchNorm2d and a ReLU activation function.

In addition, it includes a shortcut connection, if the number of input and output channels is different, it will use a 1×1 convolution to fit them, or it just adds the input to the output. This design allows the original input to jump over the intermediate transformations, which can preserve information and improve gradient flow, helps the model learn faster.

iii.

```python
class ConditionalUNet(nn.Module):
    def __init__(self, in_channels=3, cond_dim=128, num_classes=24):
        super().__init__()
        self.time_emb = TimeEmbedding(cond_dim)
        self.label_emb = LabelEmbedding(num_classes, cond_dim)
        # Encoder
        # 解析度下降，通道數上升
        self.enc1 = ResidualBlock(in_channels, 64)    # 64x64
        self.enc2 = ResidualBlock(64, 128)            # 32x32
        self.enc3 = ResidualBlock(128, 256)           # 16x16
        self.pool = nn.MaxPool2d(2)
        # bottleneck
        self.middle = ResidualBlock(256, 256)         # 8x8
        # Decoder
        # 先上採樣，skip connection結合後再卷積
        self.up1 = nn.ConvTranspose2d(256, 128, 2, 2) # 16x16
        self.dec1 = ResidualBlock(128 + 256, 128)
        self.up2 = nn.ConvTranspose2d(128, 64, 2, 2)  # 32x32
        self.dec2 = ResidualBlock(64 + 128, 64)
        self.up3 = nn.ConvTranspose2d(64, 64, 2, 2)   # 64x64
        self.dec3 = ResidualBlock(64 + 64, 64)
        self.out_conv = nn.Conv2d(64, in_channels, 1) # RGB = 3
        self.cond_proj = nn.Linear(cond_dim * 2, 256) # embedding合併後轉256維
```

```python
def forward(self, x, t, y):
    te = self.time_emb(t)
    le = self.label_emb(y)
    # 合併兩個embedding，unsqueeze讓shape變成[B, 256, 1, 1]方便家道feature map
    cond = self.cond_proj(torch.cat([te, le], dim=-1)).unsqueeze(-1).unsqueeze(-1) #一維轉四維
    e1 = self.enc1(x)                # [B, 64, 64, 64]
    e2 = self.enc2(self.pool(e1))    # [B, 128, 32, 32]
    e3 = self.enc3(self.pool(e2))    # [B, 256, 16, 16]
    # bottleneck
    m = self.middle(self.pool(e3)) + cond  # [B, 256, 8, 8]
    d1 = self.up1(m)                 # [B, 128, 16, 16]
    d1 = self.dec1(torch.cat([d1, e3], dim=1)) # skip connection
    d2 = self.up2(d1)                # [B, 64, 32, 32]
    d2 = self.dec2(torch.cat([d2, e2], dim=1))
    d3 = self.up3(d2)                # [B, 64, 64, 64]
    d3 = self.dec3(torch.cat([d3, e1], dim=1))
    out = self.out_conv(d3)          # [B, 3, 64, 64]
    return out
```

The ConditionalUNet is a U-Net architecture enhanced with conditional information. It has four main parts.

The encoder has three layers that gradually reduce the spatial resolution of the input (64x64 → 32x32 → 16x16), while increasing the number of feature channels (64 → 128 → 256).

At the bottom ( bottleneck), the feature map will downsample to 8×8, and add a condition vector which is formed by the combining of outputs of the TimeEmbedding and LabelEmbedding layers, it can guide the model learned what I want to generate.

The decoder gradually restores the image resolution (8×8 → 16×16 → 32×32 → 64×64). At each step, the decoder uses torch.cat() to connect the corresponding encoder layer (skip connection) to preserve origin details.

Finally, in output layer, a 1×1 convolutional layer is used to project the output back to the RGB image format [B(batch), 3(channel), 64, 64].

E. diffusion.py

```python
import torch

class DiffusionSchedule:
    def __init__(self, timesteps=1000, beta_start=1e-4, beta_end=0.02):
        self.timesteps = timesteps
        self.betas = torch.linspace(beta_start, beta_end, timesteps)
        self.alphas = 1. - self.betas
        self.alpha_bars = torch.cumprod(self.alphas, dim=0)

    def get_params(self, t):
        return self.betas[t], self.alphas[t], self.alpha_bars[t]

def q_sample(x_start, t, noise, alpha_bars):
    # x_t = sqrt(alpha_bar) * x_0 + sqrt(1-alpha_bar) * noise
    sqrt_ab = torch.sqrt(alpha_bars[t]).reshape(-1, 1, 1, 1)
    sqrt_1_ab = torch.sqrt(1 - alpha_bars[t]).reshape(-1, 1, 1, 1)
    return sqrt_ab * x_start + sqrt_1_ab * noise
```

The DiffusionSchedule class sets up how noise is added in each step of the diffusion process.

- **timesteps**: the total number of noise-adding steps **(I set 1000).**
- betas: linearly increase from beta_start(0.0001) to beta_end(0.02), size of the list is timesteps. **(linear beta schedule)**
- alphas:1 - betas (how much the original image remains).
- alpha_bars: the cumulative product of alphas, shows how much original image remains after n steps.

The q_sample function is the forward diffusion equation, which creates a noisy version of the image.

Formula:

$$x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{1 - \bar{\alpha}_t} \cdot \epsilon$$

- $x_0$: original clean image.
- t: step now.
- $\epsilon$: random noise tensor need to be added.
- $\bar{\alpha}_t$: controls how much original image and noise are combined.

F. sample.py

```python
import os
import json
import torch
from torchvision import transforms
from torchvision.utils import save_image, make_grid
from dataset import labels_to_onehot, NUM_CLASSES
from model import ConditionalUNet
from diffusion import DiffusionSchedule

def sample(model, schedule, labels, device, save_dir, img_names=None, show_process=False):
    model.eval()
    n = len(labels)
    x = torch.randn(n, 3, 64, 64, device=device) # 建立隨機noisy影像為初始輸入，每張影像: 3x64x64 ( RGB )
    labels = torch.stack([labels_to_onehot(l) for l in labels]).to(device) # one-hot encoding
    process_imgs = [] # denoise image process
    for t in reversed(range(schedule.timesteps)): # 從最雜訊回推
        t_tensor = torch.full((n,), t, device=device, dtype=torch.long)
        with torch.no_grad(): # 節省記憶體
            noise_pred = model(x, t_tensor, labels) # 預測當前影像noise
        beta = schedule.betas[t] # 對應timestep的參數
        alpha = schedule.alphas[t]
        alpha_bar = schedule.alpha_bars[t]
        if t > 0: # 不是最後一步
            z = torch.randn_like(x)
        else: # 最後一步不加雜訊
            z = torch.zeros_like(x)
        # 去雜訊公式
        x = (1 / torch.sqrt(alpha)) * (x - (1 - alpha) / torch.sqrt(1 - alpha_bar) * noise_pred) + torch.sqrt(beta) * z
        if show_process and t % (schedule.timesteps // 8) == 0: # 至少8個 process過程
            process_imgs.append(x.clone().cpu())
    # 儲存影像
    x = (x.clamp(-1, 1) + 1) / 2  # [-1,1] -> [0,1] denormalized
    for i in range(n):
        save_path = os.path.join(save_dir, f"{img_names[i] if img_names else i}.png")
        save_image(x[i], save_path)
    # 儲存grid
    grid = make_grid(x, nrow=8)
    save_image(grid, os.path.join(save_dir, "grid.png"))
    # 儲存denoising process
    if show_process and len(process_imgs) > 0:
        process_grid = make_grid(torch.cat(process_imgs, dim=0), nrow=len(process_imgs))
        process_grid = (process_grid.clamp(-1, 1) + 1) / 2
        save_image(process_grid, os.path.join(save_dir, "denoise_process.png"))
    return x
```

```python
if __name__ == '__main__':
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = ConditionalUNet(num_classes=NUM_CLASSES).to(device)
    model.load_state_dict(torch.load('best_model_epoch4200.pth', map_location=device))
    schedule = DiffusionSchedule()

    # 產生test.json和new_test.json的圖片
    for test_file in ['test.json', 'new_test.json']:
        with open(test_file, 'r') as f:
            labels = json.load(f)
        save_dir = f"images_5000_4200/{test_file.split('.')[0]}"
        os.makedirs(save_dir, exist_ok=True)
        sample(model, schedule, labels, device, save_dir, img_names=[str(i) for i in range(len(labels))])

    # denoising過程圖片
    denoise_labels = [["red sphere", "cyan cylinder", "cyan cube"]]
    save_dir = "images_5000_4200/denoise_process"
    os.makedirs(save_dir, exist_ok=True)
    sample(model, schedule, denoise_labels, device, save_dir, img_names=["denoise"], show_process=True)
```

It use trained Conditional DDPM model to generate synthetic images based on specific object labels. First loads the trained model and sets up the diffusion noise schedule. Then reads labels from test.json and new_test.json, and generates one image for each label set. These images are saved as one and as a grid.

At last, showing a denoising process image of label set ["red sphere", "cyan cylinder", "cyan cube"] and starts from pure Gaussian noise.

G. eval.py

```python
import os
import json
import torch
from torchvision import transforms
from PIL import Image
from utils import evaluate_images
from dataset import labels_to_onehot

def load_generated_images(image_folder):
    images = []
    for i in range(32):  # 32 張圖片
        path = os.path.join(image_folder, f"{i}.png")
        img = Image.open(path).convert("RGB")
        img_tensor = transforms.ToTensor()(img)  # 將圖片轉成(3, 64, 64)，範圍[0, 1]
        images.append(img_tensor)
    return torch.stack(images)  # 32張圖片變成一batch(32, 3, 64, 64)

def load_labels(json_file):
    with open(json_file, 'r') as f:
        label_list = json.load(f)
    return torch.stack([labels_to_onehot(l) for l in label_list]) # 轉成 one-hot encoding

def main():
    for name in ['test', 'new_test']:
        print(f"== Evaluating {name}.json ==")
        images = load_generated_images(f'images_5000_4200/{name}')
        labels = load_labels(f'{name}.json')
        evaluate_images(images, labels) # 計算正確率

if __name__ == '__main__':
    main()
```

It can evaluate the quality of the images generated by Conditional DDPM model by compares the generated images with the ground truth object labels from test.json and new_test.json using a pretrained ResNet18-based evaluator (given by TAs).

The script first loads the generated images (0.png to 31.png) and converts them into PyTorch tensors. Then converts them into 24-dimensional one-hot vectors. Both images and labels are passed into the evaluate_images() function and computes classification accuracy using the pretrained evaluator.

It will run this process twice, test.json and new_test.json.

**Command**

**Train: python train.py**

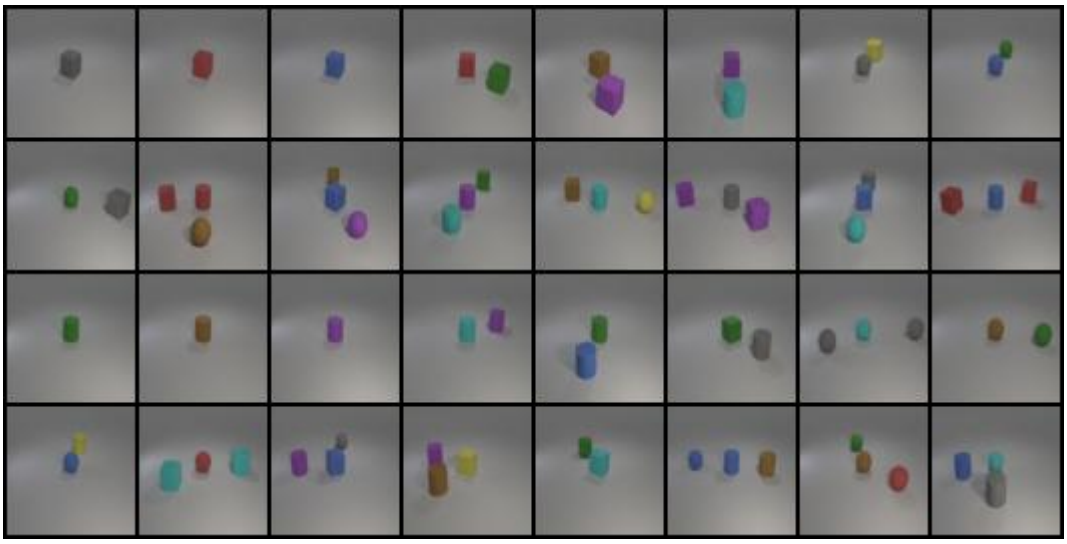
**Produce test images: python sample.py**

**Calculate accuracy: eval.py**

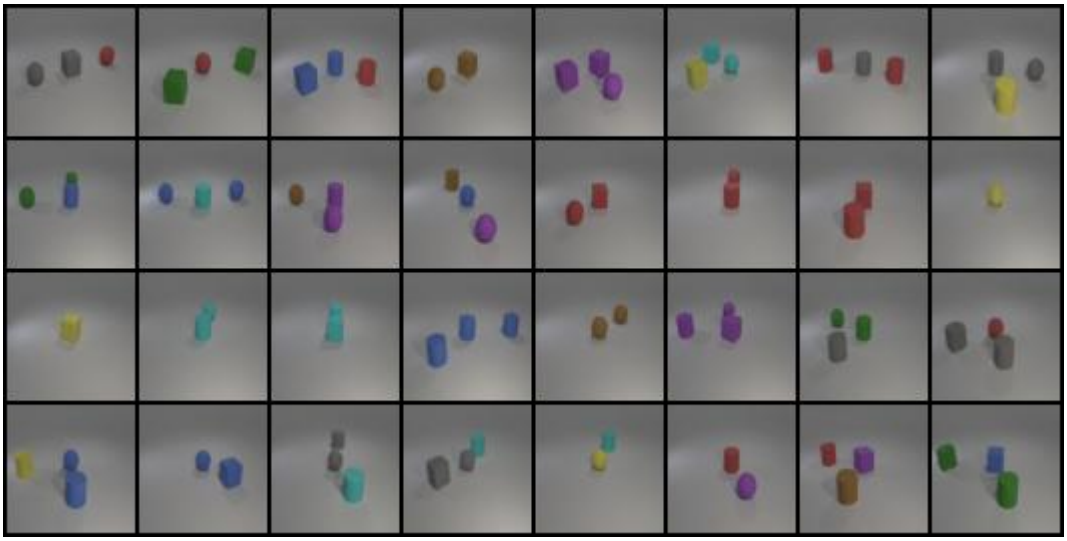
## 3. Results and discussion

Show your synthetic image grids

test:



new_test:

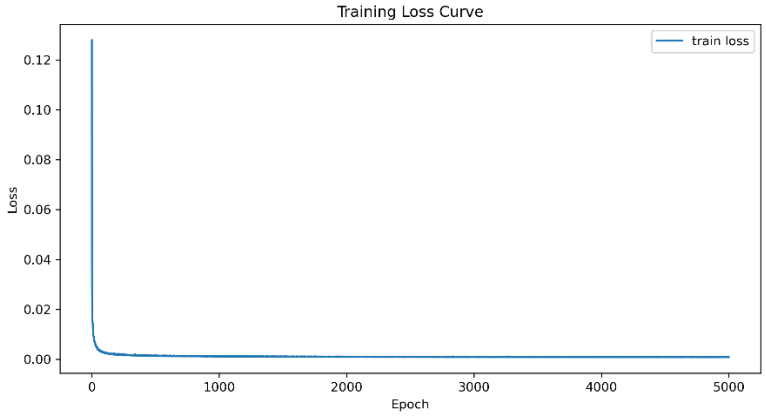


denoising process image ["red sphere", "cyan cylinder", "cyan cube"]:

Accuracy

```
PS C:\Users\annie95162\Desktop\lab6> python eval.py
== Evaluating test.json ==
Accuracy: 0.833
== Evaluating new_test.json ==
Accuracy: 0.738
```

Loss




Discussion of your extra implementations or experiments.

I implemented classifier guidance using sample_guided.py, where a pretrained classifier is used to guide the sampling process. And experimented with different values of the guidance scale parameter (gamma) then compared their impact on the classification accuracy of the generated images.

| classifier guidance | without | with | with | with | with |
|---|---|---|---|---|---|
| gamma | x | 1.0 | 1.5 | 2.0 | 2.5 |
| test accuracy | 0.694 | 0.681 | 0.667 | 0.805 | 0.736 |
| new_test accuracy | 0.750 | 0.762 | 0.774 | 0.719 | 0.755 |

I observed that adding classifier guidance improved the overall accuracy on the new test set, indicating better generalization. But when the gamma value was too large (2.0), the test accuracy increased significantly but the new test accuracy dropped, maybe overfitting to the training data. In contrast, using gamma = 1.5 achieved the highest accuracy on the new test set, showing that this moderate guidance strength provided the best generalization performance.