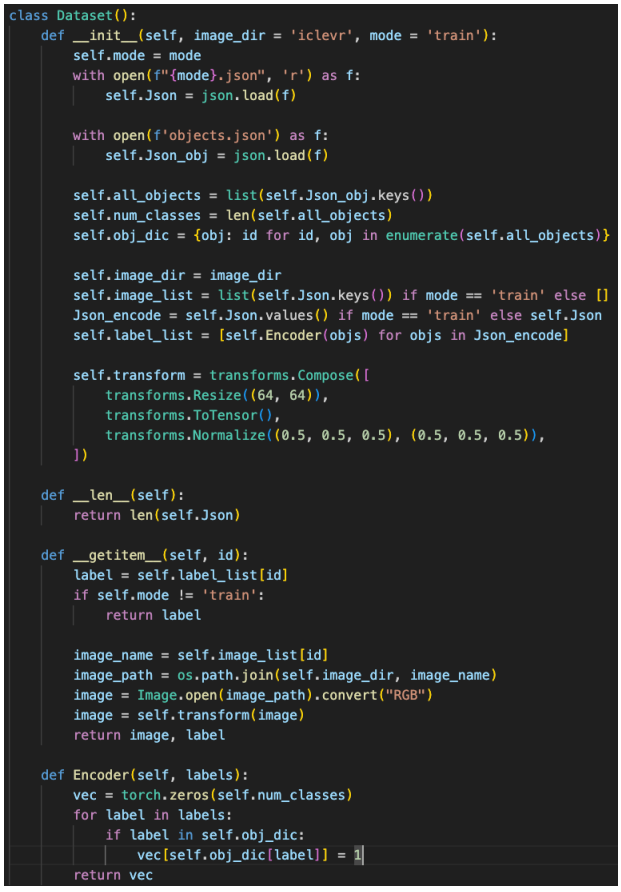
Introduction:

The theme of this lab is image generation. We use DDPM to train a model that can reconstruct images of specific geometric shapes from noise, aiming to achieve the highest possible accuracy and closely resemble real-world geometries.

Implementation Details:

Dataloader:

We determine which JSON file to load based on the current mode and create a dictionary to map object names to corresponding indices. Then, we extract the image names from the JSON file and obtain the corresponding labels. Next, we use one-hot encoding to convert each label into a vector representation, and finally, we have all the required data. When loading the dataset, we decide based on the mode whether to load and preprocess the images, returning the processed images and labels, or just returning the labels (in the test mode).



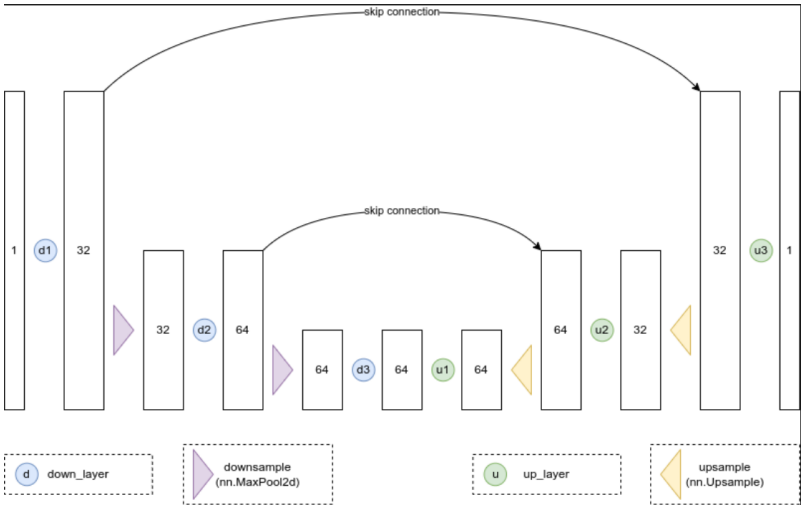
```python
class Dataset():
    def __init__(self, image_dir = 'iclevr', mode = 'train'):
        self.mode = mode
        with open(f"{mode}.json", 'r') as f:
            self.Json = json.load(f)

        with open(f'objects.json') as f:
            self.Json_obj = json.load(f)

        self.all_objects = list(self.Json_obj.keys())
        self.num_classes = len(self.all_objects)
        self.obj_dic = {obj: id for id, obj in enumerate(self.all_objects)}

        self.image_dir = image_dir
        self.image_list = list(self.Json.keys()) if mode == 'train' else []
        Json_encode = self.Json.values() if mode == 'train' else self.Json
        self.label_list = [self.Encoder(objs) for objs in Json_encode]

        self.transform = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

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

    def __getitem__(self, id):
        label = self.label_list[id]
        if self.mode != 'train':
            return label

        image_name = self.image_list[id]
        image_path = os.path.join(self.image_dir, image_name)
        image = Image.open(image_path).convert("RGB")
        image = self.transform(image)
        return image, label

    def Encoder(self, labels):
        vec = torch.zeros(self.num_classes)
        for label in labels:
            if label in self.obj_dic:
                vec[self.obj_dic[label]] = 1
        return vec
```

DDPM model:

Reference: https://github.com/huggingface/diffusion-models-class/blob/main/unit2/02_class_conditioned_diffusion_model_example.ipynb

DDPM is a highly popular image generation model in recent years. It gradually adds Gaussian noise to images, transforming them into pure noise, and trains a model to reverse this process step by step, ultimately enabling it to generate realistic images from pure noise.

Using a UNet is a good approach for implementing the reverse process. Our model takes a grayscale image as input, applies three convolutional layers with max pooling for downsampling to extract features and reduce size. Then, three upsampling layers restore the image to its original size. The convolutional layers capture local patterns, while max pooling helps with abstraction. Skip connections preserve important details like edges and contours. Upsampling uses simple nn.Upsample, which doesn't require trainable parameters.



However, the traditional UNet still has some limitations when used in DDPM. First, it cannot condition on the timestep, making it difficult to apply varying levels of denoising and reducing its ability to flexibly and precisely control the generation process. Additionally, traditional UNet struggles to capture long-range dependencies across different regions of the image, which can lead to unrealistic or inconsistent results in complex or large-scale scenes. Its simple architecture also limits its ability to process multi-level features and fine details.

To address these issues, we use the UNet2DModel. It supports timestep

conditioning, uses attention mechanisms to capture long-range relationships, applies multiple ResNet layers per block to extract richer features, includes GroupNorm for improved stability, adds Dropout to reduce overfitting, and replaces simple upsampling/downsampling with learnable blocks. These improvements result in more stable, detailed, and coherent image generation.

```
model = UNet2DModel(
    sample_size=28,           # the target image resolution
    in_channels=1,            # the number of input channels, 3 for RGB images
    out_channels=1,           # the number of output channels
    layers_per_block=2,       # how many ResNet layers to use per UNet block
    block_out_channels=(32, 64, 64), # Roughly matching our basic unet example
    down_block_types=(
        "DownBlock2D",        # a regular ResNet downsampling block
        "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
        "AttnDownBlock2D",
    ),
    up_block_types=(
        "AttnUpBlock2D",
        "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",          # a regular ResNet upsampling block
      ),
)
```

My DDPM model:

The DDPM architecture I used is based on the original example but includes several improvements. First, I adjusted the input image size to meet specific requirements and optimized the architecture. Specifically, I added an extra DownBlock in the DDPM and increased the number of channels at each layer. This allows the model to learn more diverse and higher-level image features at different resolution levels, improving the accuracy of image reconstruction. I also retained the attention mechanism to enable the model to process information effectively at lower resolutions, but I removed one layer of attention and replaced it with a sample block to accelerate training and reduce the risk of overfitting.

Additionally, I introduced a label encoder that converts one-hot class labels into feature vectors, which serve as generation conditions. In addition to transforming the label into a one-hot encoded vector via a linear layer, I applied a ReLU activation to capture non-linear relationships and added layer normalization at the end. This allows the DDPM to learn deeper conditional representations and produce smoother, more coherent outputs. These vectors are passed into the UNet through class_embed_type="identity", allowing the model to generate corresponding images based on different class conditions.

This design enables conditional image generation, producing more diverse and precise results.

```python
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from torch.utils.data import DataLoader
from diffusers import UNet2DModel
from matplotlib import pyplot as plt

class DDPM(nn.Module):
    def __init__(self, n_object_class):
        super().__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.label_encoder = self.label_encoder = nn.Sequential(
            nn.Linear(n_object_class, 512),
            nn.ReLU(),
            nn.LayerNorm(512)
        ).to(self.device)
        self.net = UNet2DModel(
            sample_size=(64,64),  # the target image resolution
            in_channels=3,  # the number of input channels, 3 for RGB images
            out_channels=3,  # the number of output channels
            layers_per_block=2,  # how many ResNet layers to use per UNet block
            block_out_channels=(128, 256, 512, 512),  # Roughly matching our basic unet example
            down_block_types=(
                "DownBlock2D",  # a regular ResNet downsampling block
                "DownBlock2D",  # a ResNet downsampling block with spatial self-attention
                "DownBlock2D",
                "AttnDownBlock2D",
            ),
            up_block_types=(
                "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
                "UpBlock2D",   # a regular ResNet upsampling block
                "UpBlock2D",
                "UpBlock2D",
            ),
            class_embed_type="identity"
        ).to(self.device)

    def forward(self, img, t, label):
        label = label.to(self.device)
        return self.net(sample=img, timestep=t, class_labels=self.label_encoder(label)).sample
```

Training and Testing process:

Noise Scheduler:

In the training process of DDPM, noise is gradually added to the image until it becomes pure noise. The noise scheduler controls this process, ensuring that the model learns how to "reverse" the noise and recover the original image. Here, I have chosen squaredcos_cap_v2 as the basis for noise scheduling. This method adjusts the noise strength using the square of the cosine function, which creates a trend where the noise increases slowly in the early stages and more rapidly in the later stages. This setting helps maintain stability during the initial training phase, preventing instability caused by too-fast learning, and accelerates the learning process in the later stages, helping the model more effectively learn how to recover the image.

```python
noise_scheduler = DDPMScheduler( num_train_timesteps=1000,
        beta_start=0.0001,
        beta_end=0.02,
        beta_schedule='squaredcos_cap_v2'
    )
```

Training Process:

To enable the model to accurately predict noise at any given timestep, we randomly sample a timestep and use the noise scheduler to mix the original image with random noise of the same size, generating a noisy image for that specific timestep. The model then takes this noisy image, along with the timestep and the conditional label, to predict the original noise that was added. We use the Mean Squared Error (MSE) loss between the predicted noise and the actual noise as the training objective.

```
for epoch in range(num_epochs):
    total_loss = 0
    for _, (img, label) in enumerate(tqdm(dataloader)):
        img = img.to(device)
        label = label.to(device)

        noise = torch.randn_like(img).to(device)
        timesteps = torch.randint(0, noise_scheduler.num_train_timesteps, (img.size(0),), device=device).long()
        noisy_img = noise_scheduler.add_noise(img, noise, timesteps)

        pred = ddpm_model.forward(noisy_img, timesteps, label)

        loss = F.mse_loss(pred, noise)
        total_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()

        torch.nn.utils.clip_grad_norm_(ddpm_model.net.parameters(), max_norm=1.0)

        optimizer.step()

    scheduler.step()
    avg_loss = total_loss / len(dataloader)
    print(f"[Epoch {epoch+1}] Loss: {avg_loss:.5f}")
```

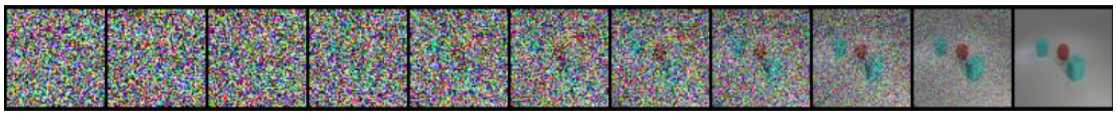
During the testing phase (inference), we start from pure noise and follow the timesteps defined by the Noise Scheduler, gradually decreasing from the largest timestep. At each step, the model predicts the amount of noise to be removed, and the image is updated accordingly. This iterative denoising process gradually transforms the noise into a clear and meaningful image.

```
for _, label in enumerate(tqdm(dataloader)):
    progress = []
    label = label.to(device)
    if not os.path.exists(out):
        os.makedirs(out)

    img = torch.randn(1, 3, 64, 64).to(device)
    for i, t in enumerate(noise_scheduler.timesteps):
        predicted_noise = ddpm_model.forward(img, t, label)

        img = noise_scheduler.step(predicted_noise, t, img).prev_sample

        if i % 100 == 0:
            progress.append(img.squeeze(0))

    progress.append(img.squeeze(0))
    result.append(img.squeeze(0))
    progress = torch.stack(progress)
    grid = make_grid((progress + 1) / 2,nrow=len(progress))
    save_image(grid, f'./{out}/processing_{_}.jpg')
    save_image((img + 1) / 2, f'./{out}/{_}.png')


    acc = evaluator.eval(img, label)
    accs.append(acc)
    #print(label)
    print(f"Accuracy: {acc:.4f}")
result = torch.stack(result)
test_grid = make_grid((result + 1) / 2, nrow=8)
save_image(test_grid, f'{out}/test_result.jpg')
print(f"Average Accuracy: {np.mean(accs):.4f}")
```

Results and Discussion:

Train command: python main.py

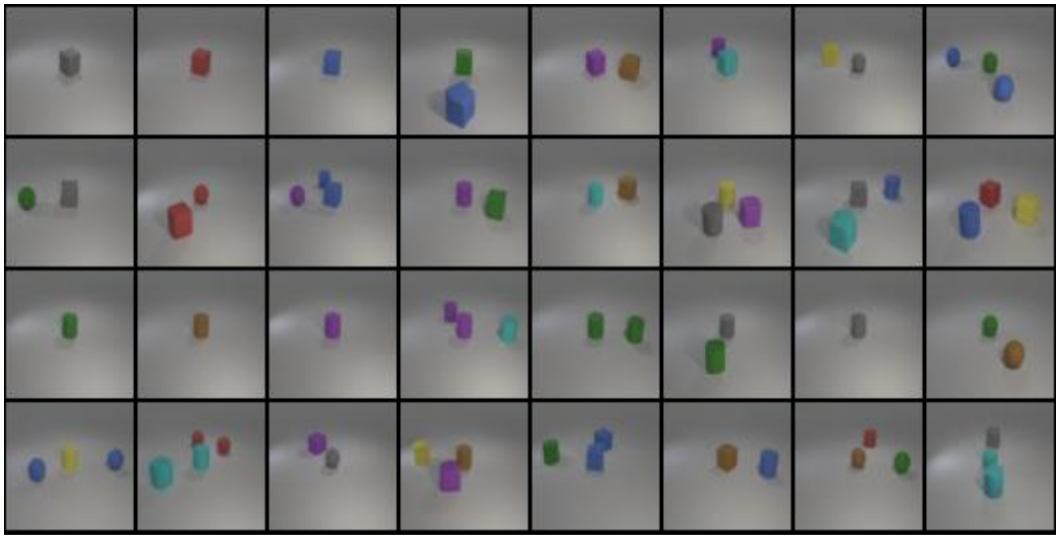
1. denoising process image with the label set ["red sphere", "cyan cylinder", "cyan cube"]



2. Show your synthetic image grids:

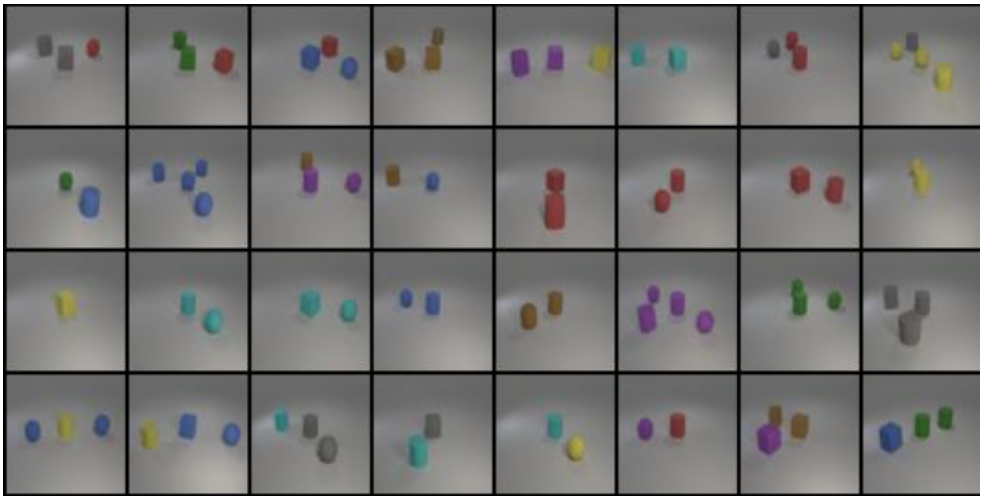
Command: python main.py --mode test

Test:



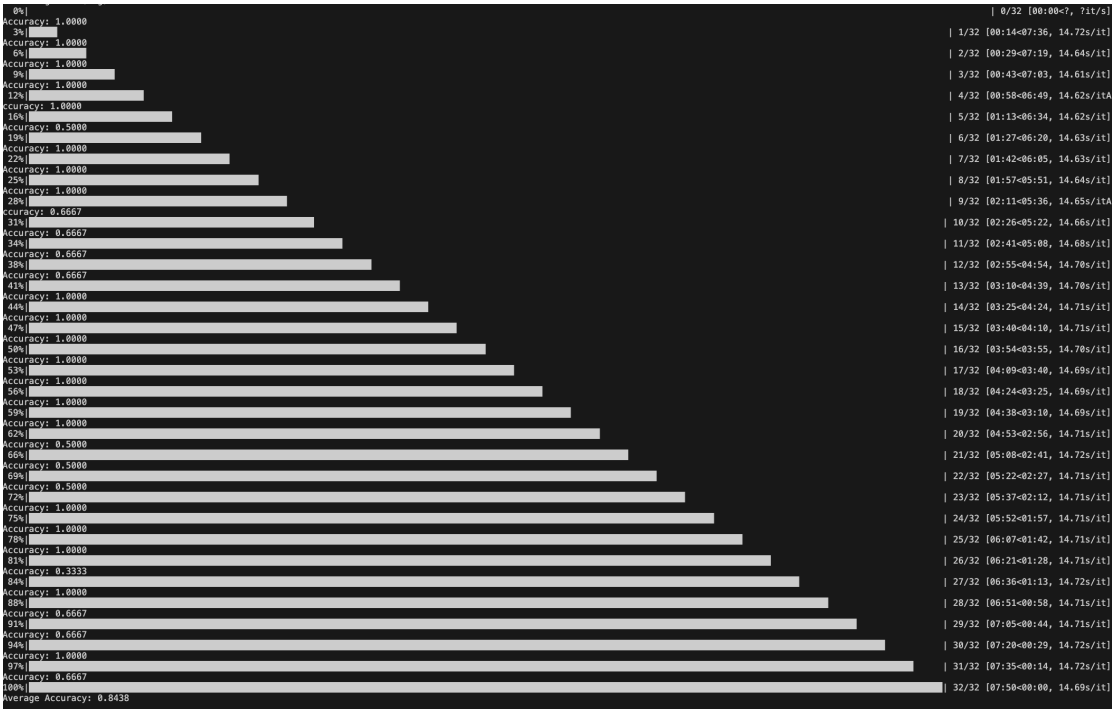
Command: python main.py --mode new_test

New_test:

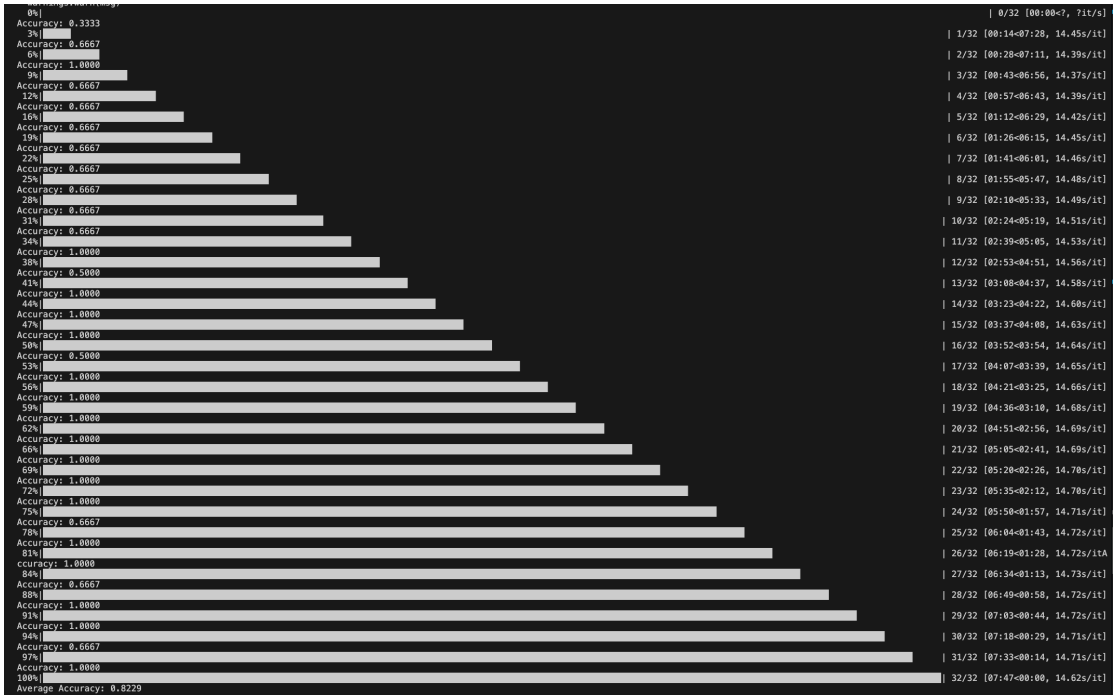


Accuracy:

Test:



New_test:



3. Other Experiments:

1.the Effect of end of beta:

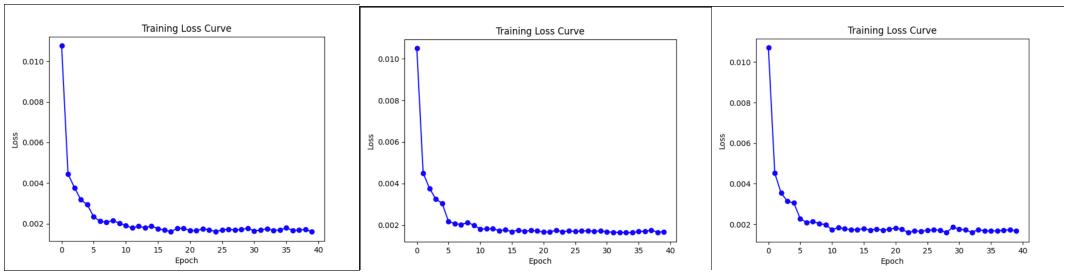
Command (Adjust the beta_end based on your case):

Train: python main.py --mode train --beta_end 0.01

Test: python main.py --mode test --beta_end 0.01

End of beta:

0.01 0.02 0.06



Average Accuracy: 0.6979 Average Accuracy: 0.8438 Average Accuracy: 0.7812

It can be observed that during the training process, the trends of all models are roughly similar. However, when beta is at its maximum (0.06), there is a slightly more noticeable fluctuation. When beta = 0.01, the accuracy is poor, likely

because the noise added is too small, making it difficult for the model to learn enough variations in the features. On the other hand, when beta = 0.06, the accuracy is still lower than at beta = 0.02, which may be due to the high noise intensity making it difficult for the model to accurately recover image details, and excessive noise leads to instability during training, affecting the final generation performance.

2. the Effect of timesteps:

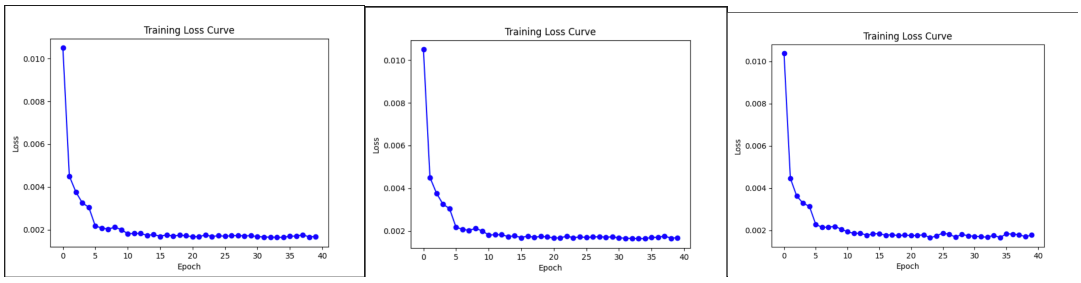
Command(Adjust the timesteps based on your case):

Train: python main.py --mode train --num_train_timesteps 500

Test: python main.py --mode test --num_train_timesteps 500

Timesteps:

500 1000 2500



Average Accuracy: 0.7708 Average Accuracy: 0.8438 Average Accuracy: 0.6823

We used different timesteps to compare their accuracy. It can be seen that the training process was smooth for all, and the trends were similar, but the accuracy was lower than the 1000-step results. The low accuracy for 500 timesteps may be due to insufficient training steps, which prevented the model from fully learning the features. On the other hand, the accuracy for 2500 timesteps was much lower than for 500, which could be because too many timesteps caused the model to overfit the noise, making it difficult to accurately reconstruct the image or learn useful features, resulting in lower-quality generated images and less accurate predictions. As shown in the figure, some examples exhibit blurred edges due to overfitting, which distorts the original

geometric shapes—for instance, turning a cuboid into a sphere. The color of the geometric objects may also change, which could be a major reason for the poor performance.