DL Lab 6 – Generative Models

112AB8005

曾大衛

I. Introduction

In this assignment, the Conditional Denoising Diffusion Probabilistic Model (Conditional DDPM) was implemented to generate specified images based on given labels. First, a DataLoader was designed to load the training images and labels from the dataset in JSON format, followed by performing one-hot encoding. Next, the configuration for the Conditional DDPM was chosen, and the noise schedule as well as the UNet model architecture were designed for training the model. After training, an Evaluator was used to assess the quality of the generated images, with the goal of achieving 80% accuracy.

II. Implement Details

```
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='squaredcos_cap_v2')
```

I chose to perform the diffusion and denoising at the label level, using the DDPMScheduler[1] to assist in the noise schedule, and opted for the cosine scheduling method.

```
class ClassConditionedUnet(nn.Module):
    def __init__(self, num_classes=24, class_emb_size=128,
    blocks = [0, 1, 1], channels = [1, 2, 2]): #Default setting as tutorial
        super().__init__()
        first_channel = class_emb_size//4
        down_blocks = ["DownBlock2D" if x == 0 else "AttnDownBlock2D" for x in blocks]
        up_blocks = ["UpBlock2D" if x == 0 else "AttnUpBlock2D" for x in reversed(blocks)]
        channels = [first_channel * x for x in channels]
        self.model = UNet2DModel(
            sample_size = 64,
            in_channels = 3,
            out_channels = 3,
            layers_per_block = 2,
            block_out_channels = (channels),
            down_block_types=(down_blocks),
            up_block_types=(up_blocks),
        )
        self.model.class_embedding = nn.Linear(num_classes, class_emb_size)

    def forward(self, x, t, label):
        return self.model(x, t, label).sample
```

For the UNet model, I used the UNet2DModel[2] as the architecture and incorporated class embedding to allow it to learn the label input together. Within the UNet2DModel class, the label is passed through class embedding and then

added to the result of time embedding, capturing both label and timestep information. In the outer class, I structured it so that, during initialization, the corresponding list for the desired model architecture can be input to construct the model. The blocks parameter is used to define whether a regular block or an attention block is used, while the channel parameter sets the number of channels at each layer. The number of channels in each layer is determined by multiplying the channels of the first layer by a scaling factor. The number of channels in the first layer must be one-fourth of the class_embed_size. The default settings come from the parameters used in the tutorial of the diffuser library (unit2)[3], not the exact ones I used during my actual training. Detailed configurations will be discussed in the hyperparameters section.

```
print('Start_Training')
for epoch in range(cur_epoch+1, n_epochs+1):
    train_loss = []
    for x, label in tqdm(train_loader):

        x, label = x.to(device), label.to(device)
        label = label.squeeze(1)
        noise = torch.randn_like(x)
        timesteps = torch.randint(0, 999, (x.shape[0],)).long().to(device)
        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)
        output = model(noisy_x, timesteps, label)

        loss = loss_function(output, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss.append(loss.item())
```

During model training, noise is randomly generated, and a timestep is drawn. Using the noise_scheduler, noise is added to the original image based on the timestep. The noisy image (noisy_x), the timestep, and the label are then fed into the model to predict the noise. The predicted noise is compared with the actual noise, and the loss is calculated using Mean Squared Error (MSELoss). This loss is used to optimize the model.

```
for y in test_loader:
    y = y.to(device)
    x = torch.randn(1, 3, 64, 64).to(device)
    for i, t in tqdm(enumerate(noise_scheduler.timesteps)):

        with torch.no_grad():
            y = y.squeeze(1)
            residual = model(x, t, y)

        x = noise_scheduler.step(residual, t, x).prev_sample
        if count == 0 and i % (timesteps // 10) == 0:
            generating.append(x.detach().cpu().squeeze(0))

    accuracy = eval_model.eval(x, y)
    accuracys.append(accuracy)
    print(f'image {count}  accuracy: {accuracy}')
```

During the testing phase, noise is first generated, and then, starting from the final timestep, the image is progressively denoised step by step to obtain the generated image. Once the image is generated, it and its corresponding label are fed into the evaluation model to calculate the accuracy.

III. Results and discussion

In this section, I will explain what configuration used while training and display what outcomes obtained.

A. Training Configuration

Batch size：8

Loss Function：nn.MSELoss()

Optimizer：Adam

Learning rate：1e-5

DDPMScheduler timesteps：1000

Beta_ schedule：squaredcos_cap_v2

Epoch：100 (I will utilize this checkpoint to show my results)

UNet Config：

- Class_ emb_size: 512
- Blocks : [0, 0, 0, 0, 0, 0 6] (6 DownBlocks and 6 UpBlocks without Attention)
- Channels: [1, 1, 2, 2, 4, 4] (128, 128, 256, 256, 512, 512)

B. Show Experiment Results

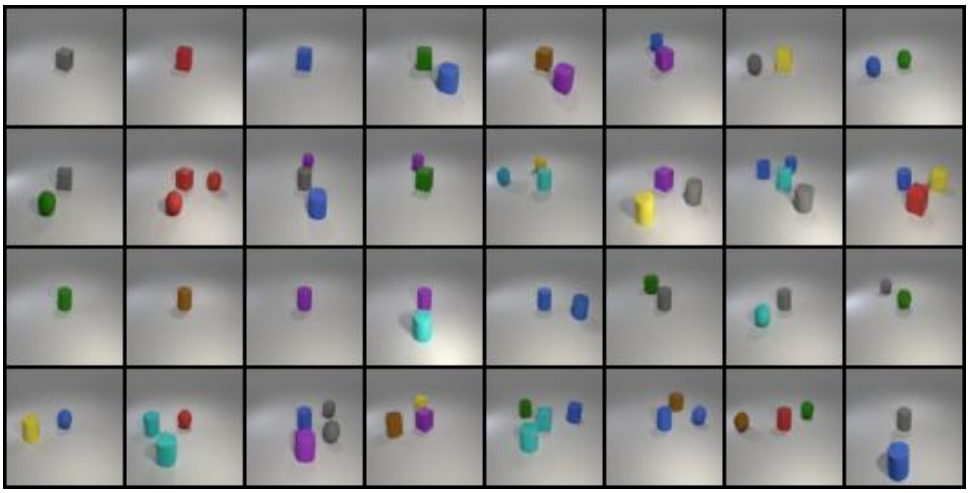
1. Show the accuracy of test.json and new_test.json data

```
Test Accuracy:
0.8072916666666666
```

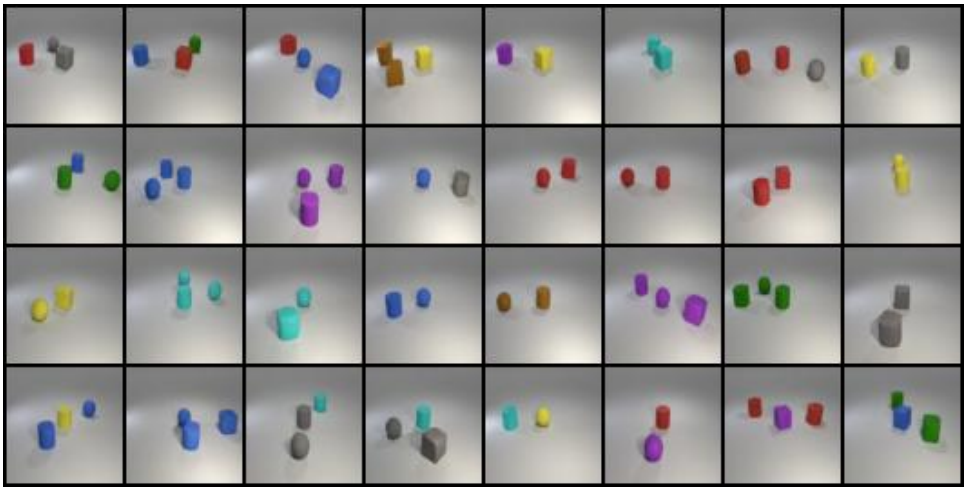
```
New Test Accuracy:
0.8385416666666666
```

2. Show the synthetic images in grids for two models for two testing files

   Test Result



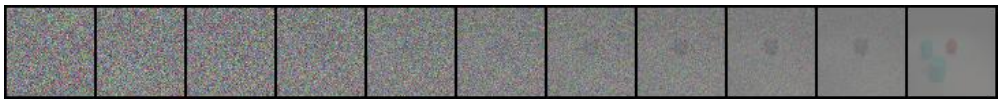
   New Test Result



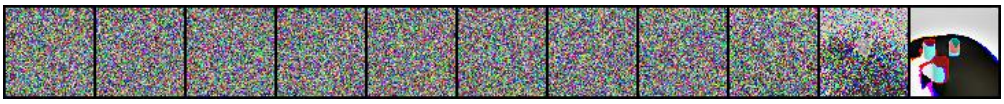
3. Show denoising process image for two testing files

   I just show the denoising process image with label set [“red sphere”, “cyan cylinder”, “cyan cube”] in test.json because only test.json has the label. On the other hand, I show the first denoising process image in new_test.json.

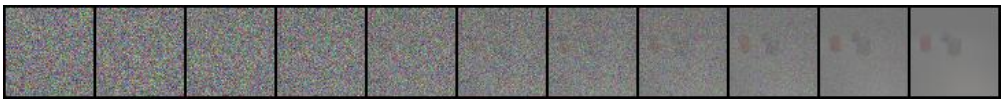
Test Denoising Process Image with normalization



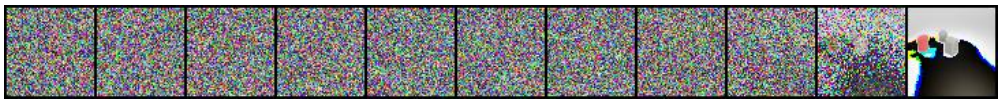
Test Denoising Process Image with denormalization



New Test Denoising Process Image with normalization



New Test Denoising Process Image with denormalization



C. Other Experiment

I change the timesteps from 1000 to 2000 and keep other settings still. For this experiment, I know it just pluses more iteration to train and test, but I would like to see the result if I use more timesetps. Eventually, I found the accuracy for these two test json file is 0.7239 and 0.7552. It's clear that the result is much lower than the experiment using timesteps 1000. Therefore, it seems that more timesteps are not helpful for training and testing.

```
noise_scheduler = DDPMScheduler(num_train_timesteps=2000, beta_schedule='squaredcos_cap_v2')
```

```
timesteps = 2000
ckpt = 'ckpt_v2/100.pth'

test_set = DiffusionLoader(json_file = 'json/test.json')
new_test_set = DiffusionLoader(json_file = 'json/new_test.json')

test_loader = DataLoader(test_set, batch_size = 1, shuffle = False)
new_test_loader = DataLoader(new_test_set, batch_size = 1, shuffle = False)

noise_scheduler = DDPMScheduler(num_train_timesteps=timesteps, beta_schedule='squaredcos_cap_v2')
```

```
Test Accuracy:
0.7239583333333334
```

```
New Test Accuracy:
0.7552083333333334
```

IV. Reference

[1] DDPMScheduler
https://huggingface.co/docs/diffusers/en/api/schedulers/ddpm

[2] UNet2DModel
https://huggingface.co/docs/diffusers/en/api/models/unet2d

[3] HuggingFace Diffuser Tutorial Unit2
https://github.com/huggingface/diffusion-models-class/tree/main/unit2