Summer 2023 DLP Lab6

312552017 董家鳴

1. Introduction (5%)

In this lab, we need to implement a conditional Denoising Diffusion Probabilistic Model (DDPM), a kind of diffusion model, to generate synthetic images according to multi-label conditions. Recall that we have seen the generation capability of CVAE in Lab 4. To achieve higher generation capacity, especially in computer vision, DDPM is proposed and has been widely applied to style transfer and image synthesis.

In this lab, given a specific condition, the diffusion model should generate the corresponding synthetic images like in figure 1.

For example, given **blue cube** and **yellow cylinder**, the diffusion model should generate the synthetic images with a **blue cube** and a **yellow cylinder**.

After generating the images by using the diffusion model, we should input the generated images to a pre-trained classifier for evaluation.

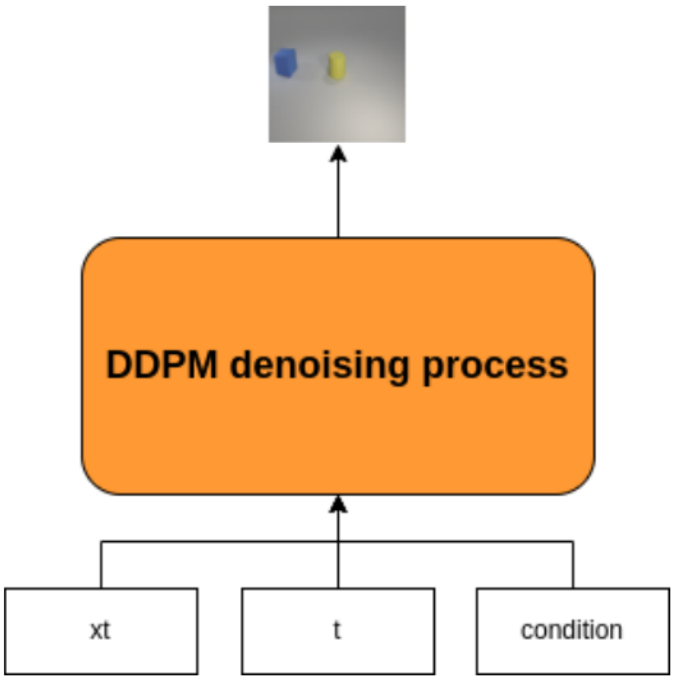

Figure 1: Conditional DDPM

Diffusion concept The diffusion model is actually a predictor of noise, meaning that it generates output with noise, and then subtracts this noise-filled output from the input to obtain the denoised output. The ground truth data of the diffusion model comes from the noise randomly generated by humans, and the input is this randomly generated noise added to the original input. In this lab, additional texts will be inputted, a concept similar to Figure 2.

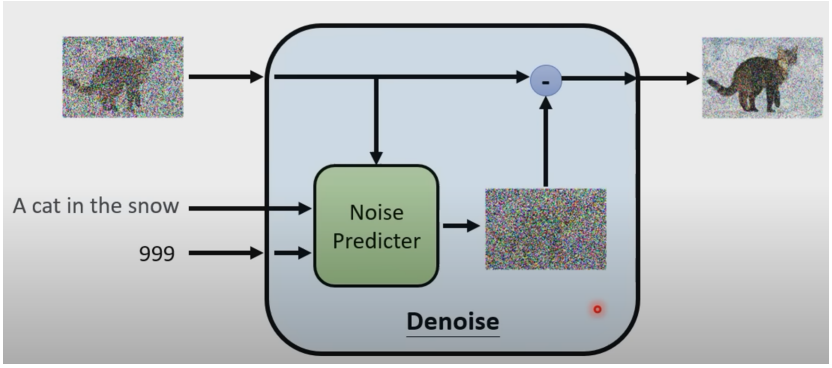

Figure 2: Diffusion module [1]

The training and testing dataset we use is i-CLEVR.

1. **objects.json**
   (a) This file is a dictionary file that contains the number of objects and the indexes. There are a totally of 24 objects in i-CLEVR datasets with 3 shapes and 8 colors.
2. **train.json**
   (a) The file is for training. The number off training data is 18009. train.json is a dictionary where keys are filenames and values are objects/
   (b) For example: "CLEVR_train_001032_0.png": ["yellow sphere"]
   (c) One image can include objects from 1 to 3
3. **test.json, new_test.json**
   (a) The file are for testing. The number of testing data is 32 for each file. Both file contain a list where each element includes multiple objects
   (b) For example: [['gray cube'], ['red cube'], ['blue cube'], ['blue cube', 'green cube'], ...]
4. **evlauator.py, checkpoint.pth**
   (a) Here are the pre-trained classifier model's script and its weight for evaluating your synthetic images.

2. Implementation Detail (20%)

Describe how to implement your model (15%)

- **DDPM** In the forward function in the figure 3, it samples noise and timestep and adds noise of different intensities based on different timesteps. In train step function in the figure 4, the model predicts the noise. Finally, through the loss function, it calculates the difference between the sampled noise and the predicted noise to perform backpropagation.

```python
def forward(clean_images, text_label):
        '''
        add noise
        '''
        # x0 ~ q(x0)
        # sample clean image
        # handle by dataloader

        # t ~ Uniform(0, num_train_timesteps)
        # sample a random timestep for each image
        bs = clean_images.shape[0]
        timesteps = torch.randint(0, noise_scheduler.num_train_timesteps, (bs,)).long().to(device)


        # ε ~ N(0, 1)
        # sample noise to add to the images
        noise = torch.randn_like(clean_images)


        # Add noise to the clean images according to the noise magnitude at each timestep
        noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)

        return noisy_images, timesteps, noise
```

Figure 3: DDPM add noise

```python
def train_step(train_dataloader, model, args, train_loss, epoch):
    model.train()
    tol_loss = 0

    for idx, (clean_images, text_label) in enumerate((pbar := tqdm(train_dataloader))):
        clean_images = clean_images.to(device)
        text_label = text_label.to(device)

        # noise
        noisy_images, timesteps, noise = forward(clean_images, text_label)


        # predict noise
        noise_pred = model(noisy_images, timesteps, text_label).sample

        # loss
        loss = args.loss_fn(noise_pred, noise)
        tol_loss += loss.item()

        # gradient descent
        args.optimizer.zero_grad()
        loss.backward(loss)
        args.optimizer.step()
        args.scheduler.step()
        tqdm_bar('train', pbar, loss.detach().cpu(), args.scheduler.get_last_lr()[0], epoch)

    train_loss.append(tol_loss / len(train_dataloader.dataset))

    return train_loss
```

Figure 4: DDPM training step

- **UNet architectures** Regarding the U-Net architectures, I initially used the diffusion model class provided by the TA, which can be found at this hyperlink: diffusion-models-class. However, I encountered errors, possibly due to differences in the library versions between the tutorial and my setup. After examining the source code of the libraries shown in figure 5 I imported, I thought that setting the class_embedding variable to nn.Embedding(num_class_embeds, time_embed_dim) would work, but I

still encountered errors. Eventually, I directly used nn.Linear to map the input and output, and it worked. My unet architecture shown in the figure 6.

```
# input
self.conv_in = nn.Conv2d(in_channels, block_out_channels[0], kernel_size=3, padding=(1, 1))

# time
if time_embedding_type == "fourier":
    self.time_proj = GaussianFourierProjection(embedding_size=block_out_channels[0], scale=16)
    timestep_input_dim = 2 * block_out_channels[0]
elif time_embedding_type == "positional":
    self.time_proj = Timesteps(block_out_channels[0], flip_sin_to_cos, freq_shift)
    timestep_input_dim = block_out_channels[0]

self.time_embedding = TimestepEmbedding(timestep_input_dim, time_embed_dim)

# class embedding
if class_embed_type is None and num_class_embeds is not None:
    self.class_embedding = nn.Embedding(num_class_embeds, time_embed_dim)
elif class_embed_type == "timestep":
    self.class_embedding = TimestepEmbedding(timestep_input_dim, time_embed_dim)
elif class_embed_type == "identity":
    self.class_embedding = nn.Identity(time_embed_dim, time_embed_dim)
else:
    self.class_embedding = None

self.down_blocks = nn.ModuleList([])
self.mid_block = None
self.up_blocks = nn.ModuleList([])
```

Figure 5: Part of the unet_2d source code I import

```
class ClassConditionedUnet(nn.Module):
    def __init__(self, num_classes=24, class_emb_size=512):
        super(ClassConditionedUnet, self).__init__()

        self.sample_size = 64
        self.out_dim = 128
        # The underlying UNet model
        self.model = UNet2DModel(
            sample_size=self.sample_size,
            in_channels=3,
            out_channels=3,
            block_out_channels=(self.out_dim, self.out_dim, self.out_dim*2, self.out_dim*2, self.out_dim*4, self.out_dim*4),
            down_block_types=(
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
                "DownBlock2D",
            ),
            up_block_types=(
                "UpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
            class_embed_type=None
        )

        # The class embedding layer
        self.model.class_embedding = nn.Linear(num_classes, class_emb_size)

    def forward(self, sample, timestep, class_labels=None, return_dict=True):
        return self.model(sample, timestep, class_labels, return_dict)
```

Figure 6: My Unet architecture

- **noise schedule** Using the DDPMScheduler library, beta_schedule adopts squaredcos_cap_v2 as the following figure shown.

```
# Set the noise scheduler
noise_scheduler = DDPMScheduler(
    num_train_timesteps=1000, beta_schedule="squaredcos_cap_v2"
)
```

Figure 7: DDPM Scheduler

- **loss functions** As shown in the figure 4, the loss function I used is the MSE loss function.

hyperparameters setting (5%)

- **batch size**: 64
- **learning rate**: 1e-4
- **epochs**: 200
- **optimizer**: AdamW
- **learning rate scheduler**: Cosine Schedule with warmup step 200
- **DDPMScheduler**: squaredcos_cap_v2
- **Down/upsampling blocks**: DownBlock2D*4, AttnDownBlock2D, DownBlock2D, UpBlock2D, AttnUpBlock2D, UpBlock2D*4
- **blocks output channels**: 128, 128, 256, 256, 512, 512, 512, 512, 256, 256, 128, 128

3. Results and discussion (25%)

accuracy screenshot based on the testing data (5%) (although the total epoch I set is 300 you see in the figure 8, I only train my model 200 epochs)

Figure 8: Test and NewTest data accurracy (above 90%)

synthetic image grids and a progressive generation image (10%)

progressive generation image (use test data as the example)

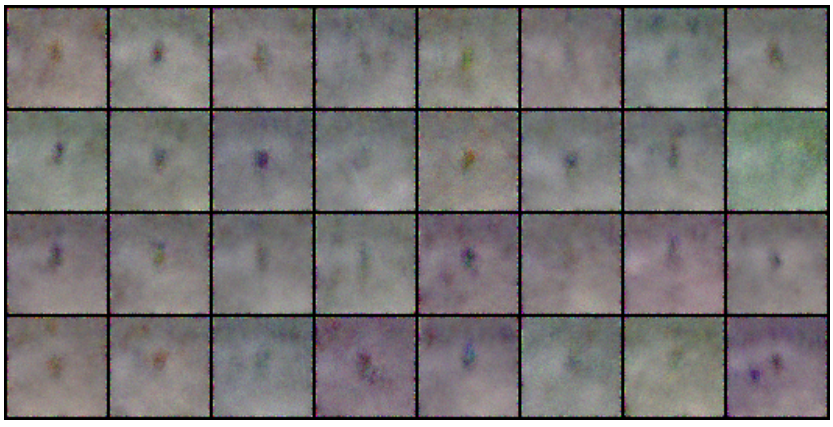

Figure 9: Test data image grid at 1 epoch

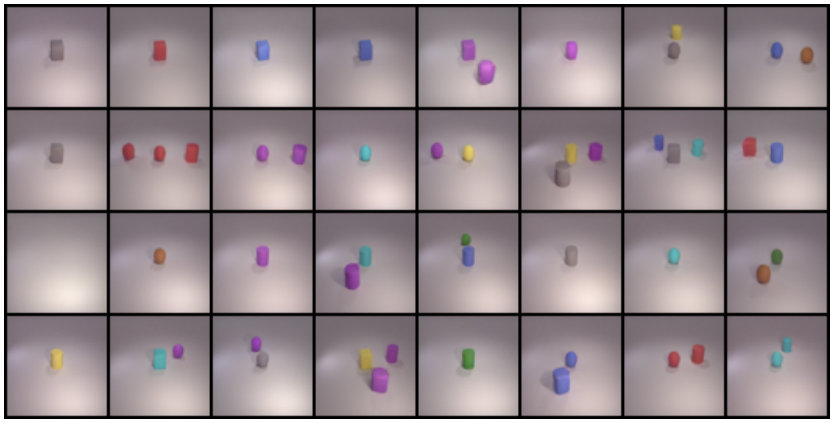

Figure 10: Test data image grid at 78 epoch

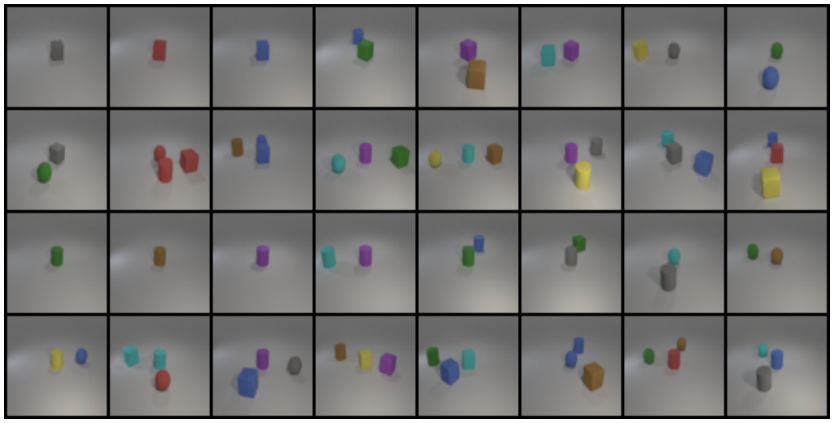

Figure 11: Test data image grid at 196 epoch, acc 93.06%

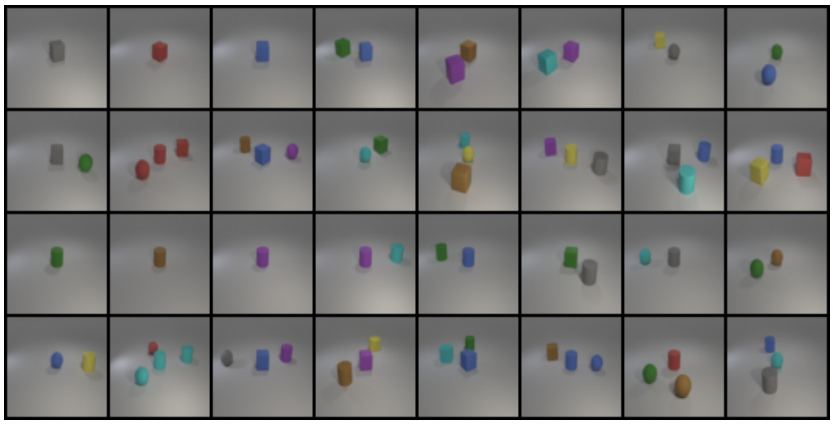

Figure 12: New Test image grid at 196 epoch, acc 90.28%

Discuss the results of different model architectures or methods (10%)

- **different scheduler**: I tried the linear scheduler, which schedules the noise amount to decrease linearly over time. Comparing figures 13 and 14, it can be observed that the performance of the linear schedule is slightly worse. However, this doesn't necessarily mean it is always inferior. The performance could be related to the hyperparameter

settings or the dataset I used.

- **differnet number of blocks**: I tried constructing a smaller model as shown in figure 15. The smaller model trained slightly faster, and surprisingly, I found that its performance was even better (see figure 16) than my original larger model. From the ResNet paper in Lab3, I learned that this could be due to suboptimal optimization settings. Theoretically, the performance of a larger model should be better, so it might be necessary to adjust the optimizer.

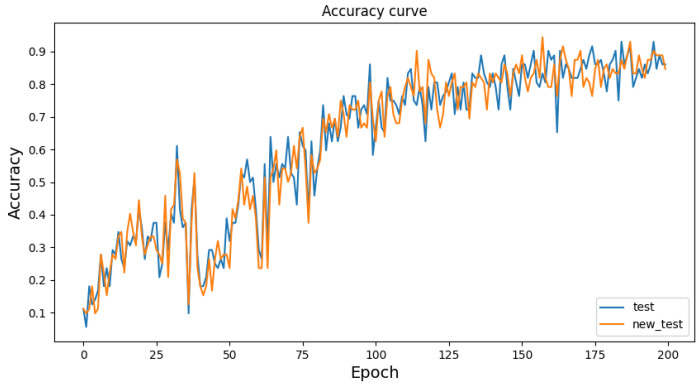

Figure 13: Accuracy Curve of squaredcos_cap_v2 noise scheduler

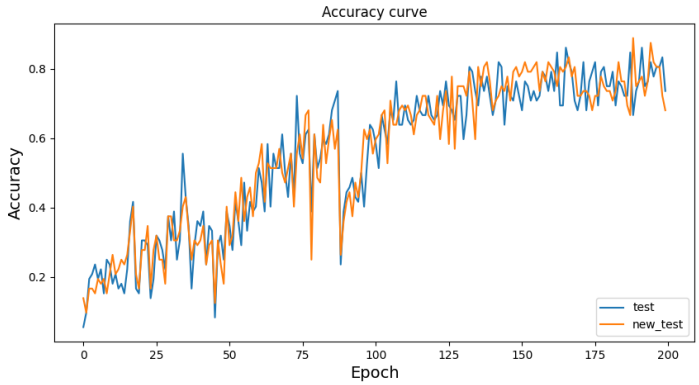

Figure 14: Accuracy Curve of linear noise scheduler

```
self.model = UNet2DModel(
    sample_size=self.sample_size,
    in_channels=3,
    out_channels=3,

    block_out_channels = (self.out_dim, self.out_dim*2, self.out_dim*2),

    down_block_types=(
        "DownBlock2D",
        "AttnDownBlock2D",
        "AttnDownBlock2D",
    ),

    up_block_types = (
        "AttnUpBlock2D",
        "AttnUpBlock2D",
        "UpBlock2D",
    ),
    class_embed_type=None
)
```

Figure 15: smaller UNet model

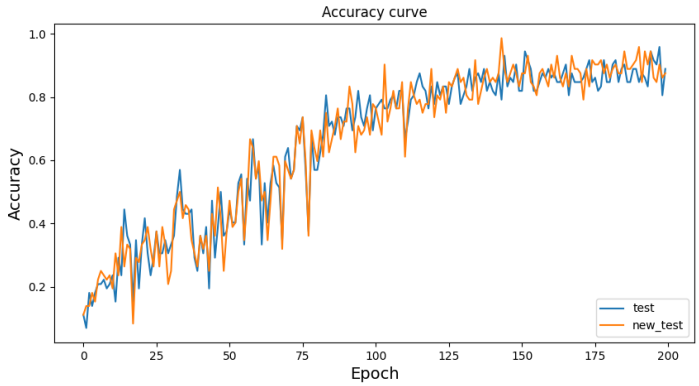

Figure 16: Accuracy Curve of smaller UNet model

References

[1] Hung-yi Lee. 【生成式 *AI*】淺談圖像生成模型 *Diffusion Model* 原理. 2023. URL: https://youtu.be/o_g9JUMw1Oc?t=2073.