DLP
Lab6 - Generative Models
Report

林睿騰(Rui-Teng Lin)
313551105

May 5, 2025

Contents

1 Introduction

In this lab, we implement a DDPM model to generate images from a given class label.

1.1 Diffusion Process

The diffusion process is the main idea in Denoising Diffusion Probabilistic Models (DDPM). It has two parts: the forward diffusion process for adding noise in training and the reverse diffusion process for removing noise in inference.

In the forward process, we add the noise to the image for each timestep t with a scheduled variance $\beta_1, ..., \beta_T$. And for training, we add a noise at a random timestep t to the image, and use a neural network to predict the step t noise. After the network predicts the noise, we can remove the noise from the image and get a better estimation of the image.

In the reverse process, we can use the network to predict the noise at each timestep, and gradually remove the noise from the image to get the clean image.

Mathematically, if we denote the original image as x_0 and the noise-corrupted version at timestep t as x_t, the forward process can be written as:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

The reverse process then learns to approximate $p(x_{t-1}|x_t)$ to gradually recover the clean image. This is done by training a model to predict the noise ϵ that was added, allowing us to remove it step by step.

2 Implementation Details

In the implementation, I use the UNet model and DDPM scheduler from diffusers library.

The workflow of a UNet2DModel can be described as follows:

1. Given a image x_0 and a label y, a timestep t is sampled from the noise scheduler.
2. Mix the image with a noise ϵ to get a noisy image x_t
3. Embed the label y and the timestep t to get the embedding e_y and e_t (In this implementation, we use a linear layer to embed the label and use a sinusoidal embedding for the timestep).
4. Get embedding $e = e_y + e_t$
5. Pass the noisy image x_t through all the small ResNet blocks (down, mid, up) in the UNet model, the output of each block is added with the embedding e.
6. The output of the last block is passed through a final out layer to get the predicted noise $\epsilon_\theta(x_t, t, y)$

And the functions of a DDPM scheduler are as following formulas: Beta scheduling in squaredcos_cap_v2 with $s = 0.008$ and $\beta_{max} = 0.999$:

$$\bar{\alpha}(t) = \cos^2\left(\frac{t+s}{1+s}\frac{\pi}{2}\right), \quad s = 0.008 \tag{2}$$

$$\beta_t = \min\left(1 - \frac{\bar{\alpha}(\frac{t}{T})}{\bar{\alpha}(\frac{t-1}{T})}, \beta_{max}\right) \quad \text{for } t = 1, \ldots, T \tag{3}$$

Add noise function:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{4}$$

Step function, where $\alpha_t = 1 - \beta_t$:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t, y) \right) + \sqrt{\beta_t} z, \quad z \sim \mathcal{N}(0, I) \tag{5}$$

```
self.model = UNet2DModel(
    sample_size=config["image_size"],
    in_channels=3,
    out_channels=3,
    layers_per_block=2,
    block_out_channels=(128, 128, 256, 256, 512, 512),
    class_embed_type="identity",
    down_block_types=(
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",
        "DownBlock2D",
    ),
    up_block_types=(
        "UpBlock2D",
        "AttnUpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
).to(self.device)

self.class_embedding = nn.Linear(24, 512).to(self.device)
```

In the original tutorial, the dataset was single-label, so it can use a Embedding layer to embed the label. But in this lab, the images are multi-label, so we need to use a one-hot encoding, and pass through a linear layer to get the embedding.

2.1 Training

In the training process, we want to train a model that can predict the noise at each timestep. To achieve this goal, we randomly choose a timestep t and a random noise ϵ to corrupt the image.

```
images, labels = batch
images = images.to(self.device)
labels = labels.to(self.device)

noise = torch.randn(images.shape).to(self.device)
timesteps = torch.randint(
    0,
    self.config["num_train_timesteps"],
    (images.shape[0],),
    device=self.device,
```

```
).long()
noisy_images = self.noise_scheduler.add_noise(images, noise, timesteps)
```

After that, we can simply use a MSE loss to train the model to predict the noise.

Loss function:

$$L = \mathbb{E}_{x_0, \epsilon \sim \mathcal{N}(0,1)}[\|\epsilon - \epsilon_\theta(x_t, t, y)\|^2] \tag{6}$$

```
noise_pred = self.model(noisy_images, timesteps, labels).sample
loss = nn.MSELoss()(noise_pred, noise)
```

2.2 Inference

In inference, we want to recover a image given a noise and the label.

```
x = torch.randn(1, 3, 64, 64).to(self.device)
y = label.unsqueeze(0).to(self.device)
```

To recover the image, we run the reverse process for T times. At each step, we use the model to predict the noise and then remove the noise from the image.

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t, y) \right) \tag{7}$$

```
for i, t in enumerate(self.noise_scheduler.timesteps):
    with torch.no_grad():
        r = self.model(x, t, y).sample

    x = self.noise_scheduler.step(r, t, x).prev_sample
```

After that, we can get a clean image given the label.

3 Analysis and Discussion

The synthetic image grids are shown in Figure 1 and 2. And the process of generating a image with label ["red sphere", "cyan cylinder", "cyan cube"] is shown in Figure 3. Also, the accuracy of the output images are shown in Figure 4.

The parameters of the model are shown in Table 3.

3.1 Extra experiments

Aside from the experiments with the parameters in Table 3, I also tried to use different parameters to train the model. To be specific, I tried to use different training / evaluation steps, and maintaining other parameters the same. The results with 500 training/evaluation steps are shown in Table 1, and the results with 1000 training/evaluation steps are shown in Table 2.

We can see from the results that using more training/evaluation steps does not always lead to better results, even their result of label ["red sphere", "cyan cylinder", "cyan cube"] are missing one object. In my opinion that using more training/evaluation steps does not lead to better results because of the following reasons:

3.1.1 Difficulty of the task

When the number of timesteps is large, the perturbation to predict is more difficult, so it can't learn the mapping in a short time.

3.1.2 Too few epochs

Like above, when the timesteps is large but the number of epochs is not enough, the training process does not cover all the timesteps enough, so the model can't learn the mapping.

3.1.3 Error accumulation

When the number of timesteps is large, the error of the model will accumulate, so the result will be worse.

A References

the original tutorial from huggingface diffusers library:
`https://github.com/huggingface/diffusion-models-class/blob/main/unit2/02_class_conditioned_diffusion_model_example.ipynb`

A.1 Parameters

Parameter	Value
Learning Rate	1e-5
Number of Epochs	200
Batch Size	32
Beta Schedule	squaredcos_cap_v2
Image Size	64
Number of Train Timesteps	100
Number of Inference Timesteps	100

Table 3: Parameters of the model

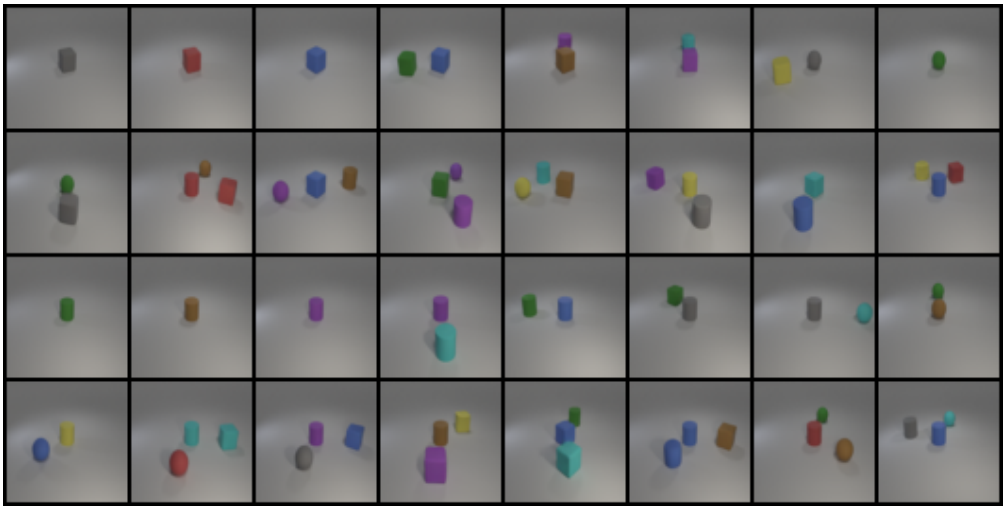


Figure 1: Synthetic image grids of test.json

Figure 2: Synthetic image grids of new_test.json

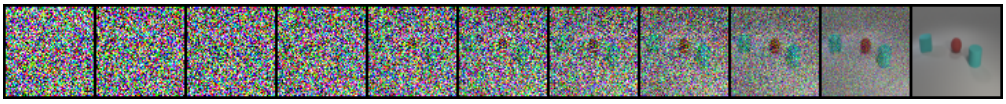


Figure 3: The process of generating a image with label [”red sphere”, ”cyan cylinder”, ”cyan cube”]

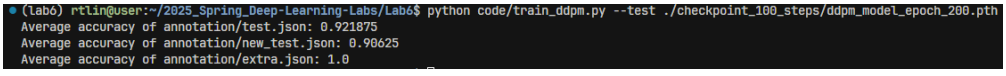
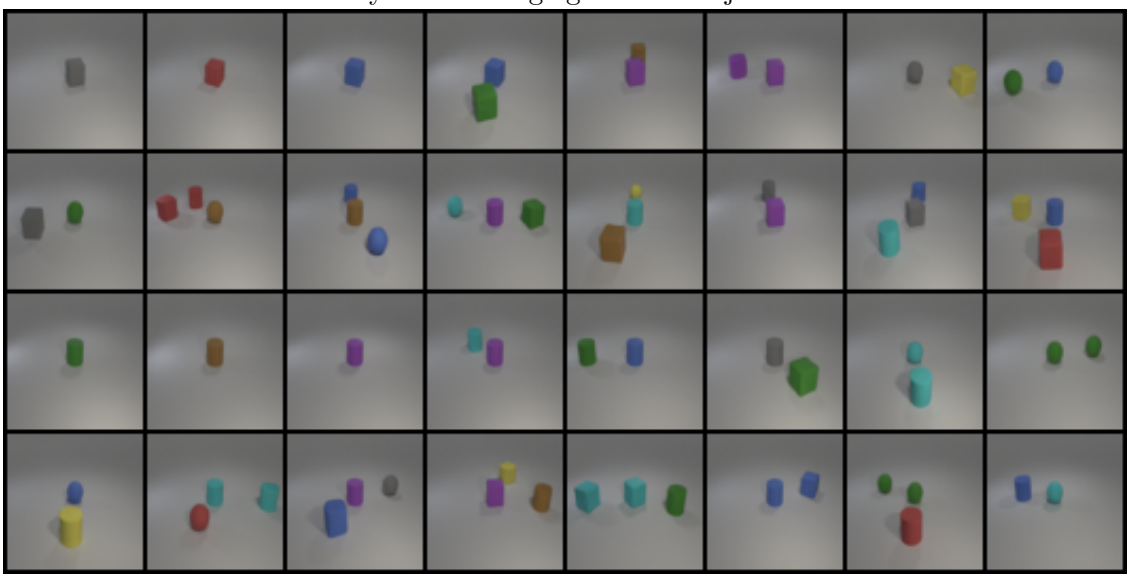


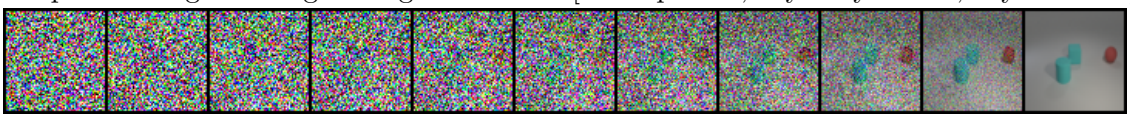
Figure 4: The accuracy of the output images

Synthetic image grids of test.json



Synthetic image grids of new_test.json

The process of generating a image with label [”red sphere”, ”cyan cylinder”, ”cyan cube”]

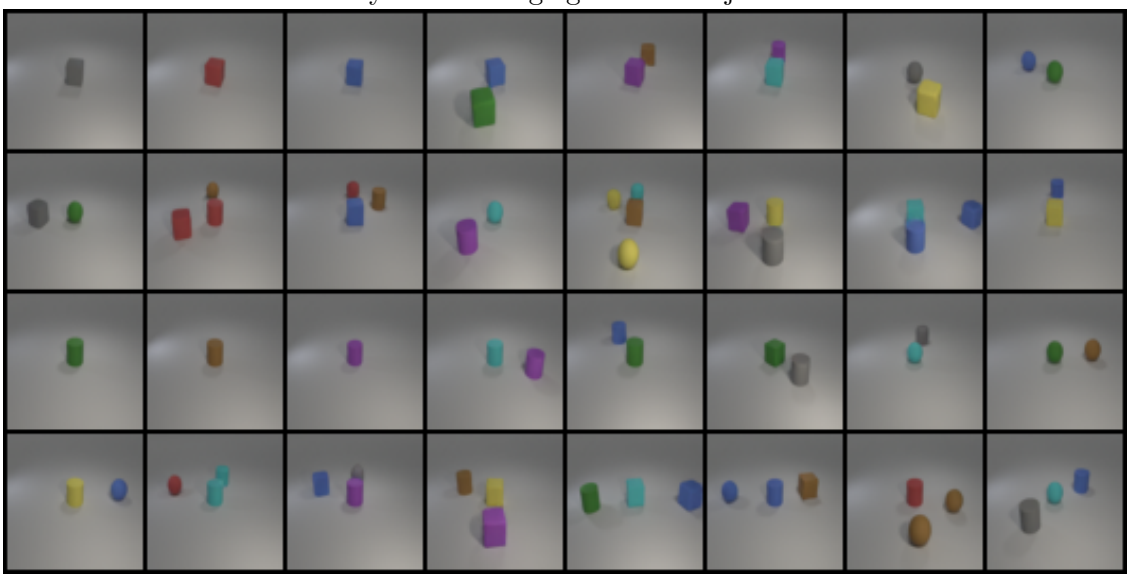


The accuracy of the output images

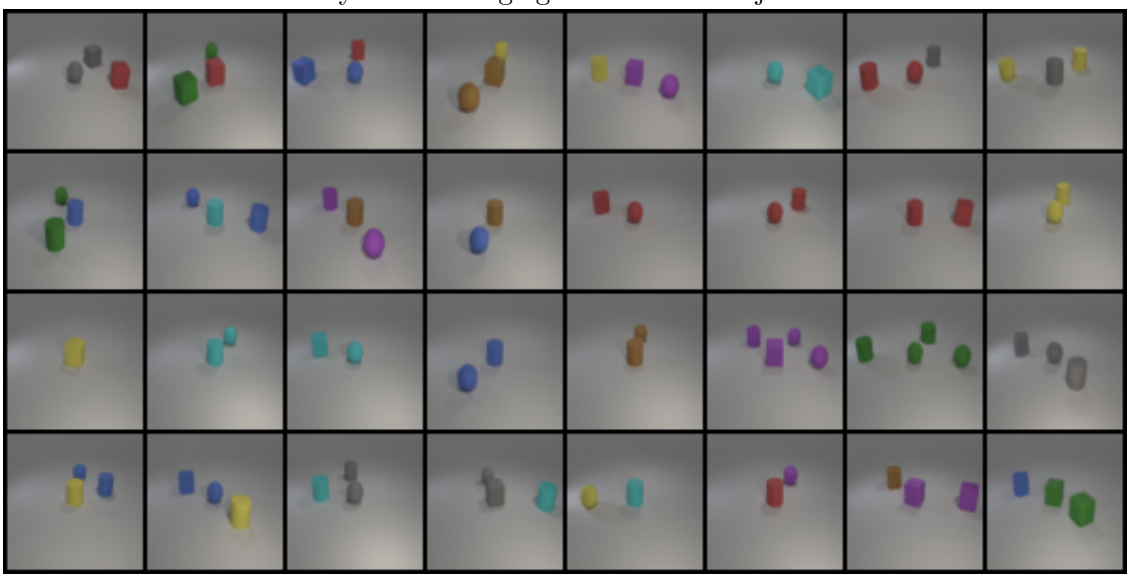
```
(lab6) rtlin@user:~/2025_Spring_Deep-Learning-Labs/Lab6$ python code/train_ddpm.py --test ./checkpoint_500_steps
/ddpm_model_epoch_200.pth --timesteps 500
Average accuracy of annotation/test.json: 0.8333333333333334
Average accuracy of annotation/new_test.json: 0.8697916666666666
Average accuracy of annotation/extra.json: 1.0
```

Table 1: Results with 500 training/evaluation steps

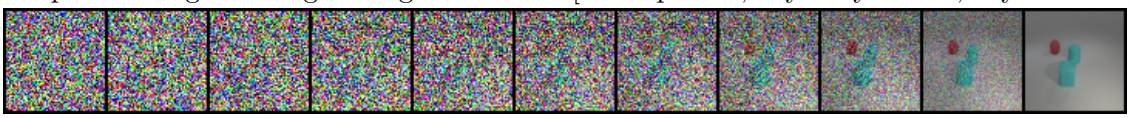
Synthetic image grids of test.json



Synthetic image grids of new_test.json



The process of generating a image with label [”red sphere”, ”cyan cylinder”, ”cyan cube”]



The accuracy of the output images

```
(lab6) rtlin@user:~/2025_Spring_Deep-Learning-Labs/Lab6$ python code/train_ddpm.py --test ./checkpoint_1000_step
s/ddpm_model_epoch_200.pth  --timesteps 1000
Average accuracy of annotation/test.json: 0.8489583333333334
Average accuracy of annotation/new_test.json: 0.8020833333333334
Average accuracy of annotation/extra.json: 1.0
```

Table 2: Results with 1000 training/evaluation steps