DLP_LAB6

— 313551133 陳軒宇

1. Introduction (5%)

本次實驗的目的為實現一個條件式去噪擴散概率模型（conditional Denoising Diffusion Probabilistic Model, DDPM），用於生成符合多標籤條件的合成圖像。例如，當輸入「紅色球體」、「黃色立方體」和「灰色圓柱體」等標籤時，模型應能生成包含這些物體的合成圖像。

我們將使用 `iclevr` 資料集進行訓練和測試，其中包含了 18009 筆訓練資料，以及 32 筆測試資料。

最後，我們將使用 `test.json` 和 `new_test.json` 進行測試，並使用預訓練的評估器評估生成的圖像。

2. Implementation details (25%)

> Describe how you implement your model, including your choice of DDPM, noise schedule. (There is no demo in this lab, so please write in detail.)

DDPM 的核心原理是通過一個逐步去噪的過程來生成圖像。其主要架構為下：

1. Noise Scheduler：負責向原始圖像逐步添加高斯噪聲，模擬從純噪聲到清晰圖像的反向過程。
2. Noise Predictor：通常是一個 U-Net 結構的神經網絡，用於在每個時間步預測添加的噪聲。
3. Condition Embedding：將標籤信息嵌入到模型中，以實現條件生成。

在 training 過程中，模型學習如何從噪聲圖像中逐步恢復原始圖像。具體步驟如下：

1. 使用 Noise Scheduler 向原始圖像添加不同程度的隨機噪聲。
2. Noise Predictor 嘗試預測每個時間步驟中添加的噪聲。
3. 計算預測噪聲與實際添加噪聲之間的損失。
4. 通過反向傳播更新模型參數，優化 Noise Predictor 的預測能力。

在 inference 階段，模型從純噪聲開始，通過反覆去噪過程逐步生成符合給定條件的圖像。

具體的實現如下：

- Noise predictor
  - 使用了 `diffusers` 中的 `UNet2DModel` 作為 `U-Net` 的架構，並加入 `Time Embedding` 和 `Condition Embedding`。

```
class ConditionalUNet2DModel(nn.Module):
    def __init__(self, num_classes=24, embedding_size=4):
        super(ConditionalUNet2DModel, self).__init__()

        self.label_embedding = nn.Embedding(num_classes, embedding_size)
        self.model = UNet2DModel(
            sample_size=64,
            in_channels=3 + num_classes,
            out_channels=3,
            time_embedding_type="positional",
            layers_per_block=2,
            block_out_channels=(128, 128, 256, 256, 512, 512),
            down_block_types=(
                "DownBlock2D",  # a regular ResNet downsampling block
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",  # a ResNet downsampling block with sp
                "DownBlock2D",
            ),
            up_block_types=(
                "UpBlock2D",  # a regular ResNet upsampling block
                "AttnUpBlock2D",  # a ResNet upsampling block with spatia
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )

    def forward(self, x, t, label):
        b, c, w, h = x.shape
        embeded_label = label.view(b, label.shape[1], 1, 1).expand(
            b, label.shape[1], w, h)
        x = torch.cat((x, embeded_label), 1)  # unet input
        x = self.model(x, t).sample  # unet output
        return x
```

- Noise scheduler

- 使用了 `diffusers` 的 `DDPMScheduler` 作為噪聲調度器
- 時間步數（num_train_timesteps）設為 1000
- 使用 "squaredcos_cap_v2" 作為 beta 調度方法

```
self.noise_scheduler = DDPMScheduler(num_train_timesteps=self.num_timeste
```

- Condition Embedding
  - 使用 `nn.Embedding` 作為條件嵌入，條件標籤被擴展到與輸入圖像相同的空間維度，並在通道維度上與噪聲圖像連接。

```
self.label_embedding = nn.Embedding(num_classes, embedding_size)
```

- Time Embedding
  - 使用 `diffusers` 中 `UNet2DModel` 的 "positional" 類型，這是一種基於正弦餘弦函數的位置編碼方法。

```
time_embedding_type="positional"
```

- Loss function
  - 使用 Mean Squared Error (MSE) 作為損失函數。

```
self.criterion = nn.MSELoss()
```

- Optimizer
  - 使用 `Adam` 作為優化器

```
self.optimizer = torch.optim.Adam(self.noise_predicter.parameters(), lr=s
```

- Learning rate Scheduler
  - 使用 Cosine Annealing 的學習率調度策略，並包含了預熱階段：

```
self.lr_scheduler = get_cosine_schedule_with_warmup(
   optimizer=self.optimizer,
   num_warmup_steps=args.lr_warmup_steps,
   num_training_steps=len(self.train_loader) * self.epochs,
   num_cycles=50
)
```

- Training step
  - 在每個訓練步驟中，我們隨機選擇 timestep，添加噪聲，然後讓模型預測噪聲：

```
t = torch.randint(0, self.num_timesteps, (x.shape[0], ), device=self.dev
noise = torch.randn_like(x)
noise_x = self.noise_scheduler.add_noise(x, noise, t)
noise_pred = self.noise_predicter(noise_x, t, y)
```

- Inference step
  - 在推理階段，我們從純噪聲開始，逐步去噪以生成最終圖像：

```
x = torch.randn(32, 3, 64, 64).to(self.device) # sample noise

for t in tqdm(self.noise_scheduler.timesteps, desc=f"({test_mode}) Ep
    pred_noise = self.noise_predicter(x, t, y)
    x = self.noise_scheduler.step(pred_noise, t, x).prev_sample
```

  - 最後，我們使用評估模型來計算生成圖像的準確度：

```
acc = self.eval_model.eval(images=x.detach(), labels=y)
```

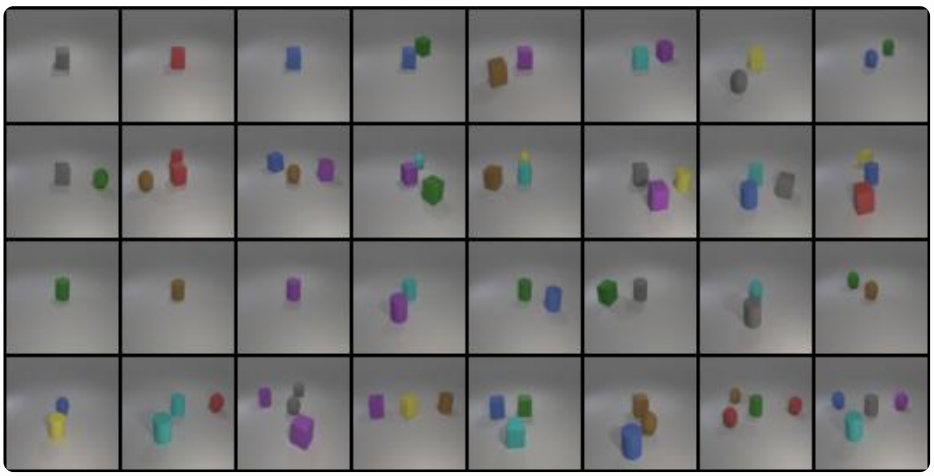
3. Results and discussion (30%)

> Show your synthetic image grids (total 16%: 8% * 2 testing data) and a denoising process image (4%)

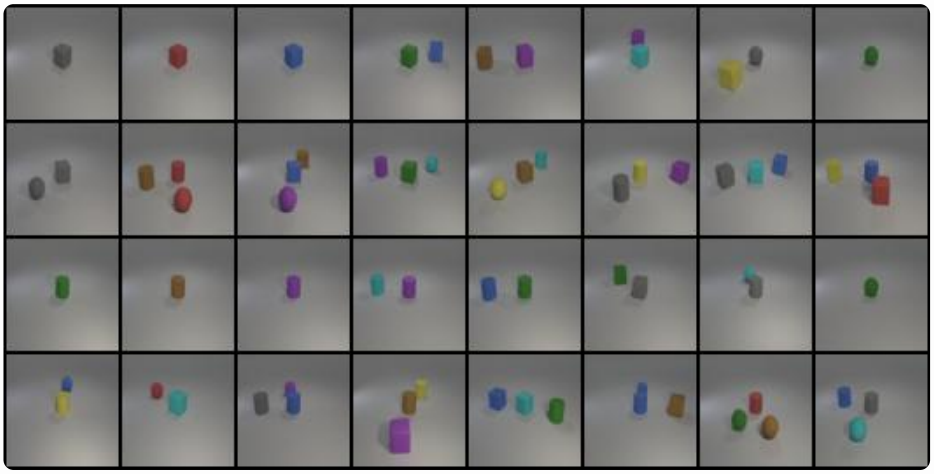
這裡展示了 $epoch = 130$ 和 $epoch = 116$ 的生成圖像，上傳到繳交區為 $epoch = 130$ 的結果，這兩種結果的得分都 $> 80\%$ 。至於為甚麼會展示兩種結果，會在 3.3 節說明。

3.1 Synthetic image grids (16%)

- Results for *test.json* (8%)

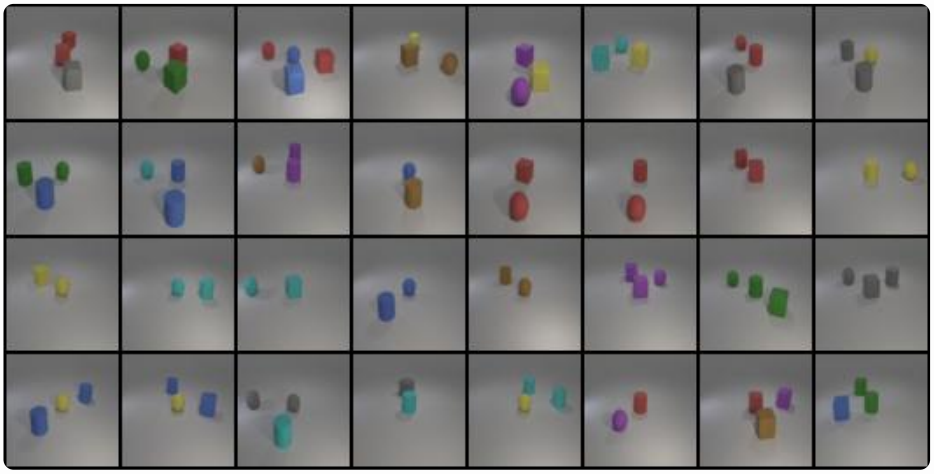


epoch=130, seed=116, on Google Colab



epoch=116, seed=116, on Local machine

- Results for *new_test.json* (8%)

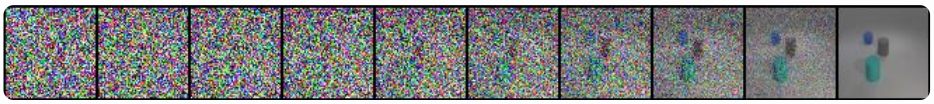


epoch=130, seed=116, on Google Colab

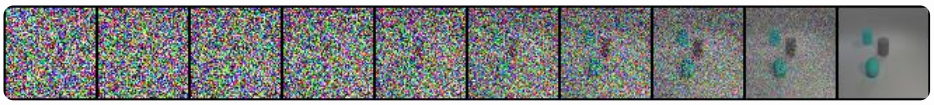
epoch=116, seed=116, on Local machine

3.2 Denoising process image (4%)

- Denoising process for ["blue cylinder", "gray cylinder", "cyan sphere"]



epoch=130, seed=130, on Google Colab



epoch=116, seed=116, on Local machine

3.3 Discussion of extra implementations or experiments (10%)

雖然我在 $epoch = 130$ 得到較好的結果，但實際檢視生成的圖像時，發現生成的圖像很容易有額外的物體。例如在 *test.json* 和 *new_test.json* 中，最多只會有 3 種物體，但生成的圖像卻會有出現 4 種物體的情況。

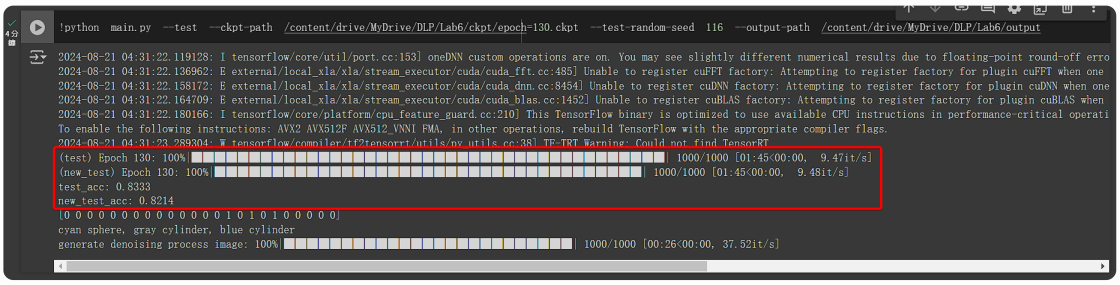
在查看 `evaluator.py` ，發現在計算 accuracy 時，若標籤只有 k 個物體，只會計算前 k 個物體的準確度，因此即便生成的圖像有 4 種物體，只要前 3 種物體的準確度是 100% ，整體的準確度就會是 100% 。

4. Experimental results (40%)

4.1 Classification accuracy (40%)

epoch=130

- Accuracy on test.json: 0.8333
- Accuracy on new_test.json: 0.8214



由於 DDPM 模型的隨機性，生成的圖像可能會有所不同，在固定亂數種子後，才能夠確保每次生成的圖像相同。

```
1 torch.manual_seed(args.test_random_seed) # fix random seed on test single epo
```

但我發現即便使用相同的亂數種子，在不同的裝置上生成的圖像仍然可能會有所不同，因此實驗結果只能在 Colab 上複現。

4.2 Inference process

> (Please make sure TA can understand how to run your inference code and have your synthetic images)

- local

```
python .\main.py --test --ckpt-path .\ckpt\epoch=116.ckpt --test-random-seed
```

- Google Colab

```
!python main.py --test --ckpt-path /content/drive/MyDrive/DLP/Lab6/ckpt/epoch
```