Deep Learning Lab6: Generative Models

TAICA Student ID (NCU): 113522118
Name: 韓志鴻

1. Introduction

In Lab 6, the goal is to implement a Conditional DDPM. After receiving a set of label conditions (color and shape), the model must take those labels into account during generation, producing synthesized images that contain the specified objects. A DDPM has two principal stages:

A. **Forward process** – progressively adds random noise to a real image.

B. **Reverse process** – learns to remove that noise step by step, ultimately reconstructing a clear synthetic image from pure noise.

The experiment uses the i-CLEVR dataset. When given a color and shape condition (e.g., "red sphere," "yellow cube," "gray cylinder"), the model must generate a synthesized image containing those objects. In this lab we generate images according to the label conditions in test.json and new_test.json, then evaluate the model's accuracy. Based on the results, the DDPM achieved **97.22 % accuracy on test.json** and **94.05 % accuracy on new_test.json**.

2. Implementation details

Describe how you implement your model, what is this step, including your choice of DDPM, noise schedule.

The workflow of this lab is as follows: first, run **train.py** to train the model. Before training, **data_loader.py** downloads training labels and images. Next, **ConditionalDDPMModel**, built on a UNet architecture, is employed. During each training run, **evaluator.py** validates the model. Once training is complete, **inference.py** is executed to perform inference and produce resulting images.

A. Conditional DDPM Model (model.py)

The generative model in Lab 6 is a conditional diffusion model (Conditional DDPM), built upon the tutorials in [1] and [2] with slight modifications to its architecture. Its core structure is as follows:

i. Architecture:
- Adopts diffusers.UNet2DModel as the backbone model.
- This UNet model is used to learn the process of progressively restoring images from additive Gaussian noise (the denoising process).
- The model receives three inputs: noisy image x, time step t, and condition vector cond.

ii. Condition Embedding:
- The condition vector cond is a one-hot or multi-hot vector representing the target object or scene condition (e.g., "red sphere").
- It uses an nn.Linear(cond_dim, embedding_dim) layer to convert the condition into a UNet-compatible embedding vector.
- The embedding vector is passed to UNet as class embedding to enable conditional generation.

iii. UNet architecture design:
- Input and output channels: in_channels=3, out_channels=3, i.e., standard RGB image format.
- Image size: default is 64×64.
- Block design:

- The down-sampling path consists of six layers, including Attention blocks (AttnDownBlock2D) to enhance the model's perception of local and global structures.
- The up-sampling path also consists of six layers, performing symmetric reconstruction.

- Embedding method:
  - class_embed_type="identity"：Supports external custom class embeddings (i.e., the outputs of the aforementioned Linear layer).
  - time_embedding_type="positional"：Uses positional encoding to handle the time steps.

iv. Forward pass process:

- Convert the condition vector through a Linear layer into a class embedding. When fed into the UNet alongside the time step and the image, the model outputs the predicted noise, which is then used for the reverse inference (denoising step) in the diffusion process.

```
class ConditionalDDPMModel(nn.Module):
    def __init__(self, image_size=64, in_channels=3, out_channels=3, cond_dim=24, embedding_dim=256):
        super().__init__()
        self.image_size = image_size
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.cond_dim = cond_dim
        self.class_embedding = nn.Linear(cond_dim, embedding_dim)
        self.unet = UNet2DModel(
            sample_size=image_size,
            in_channels=in_channels,
            out_channels=out_channels,
            layers_per_block=2,
            block_out_channels=(64, 128, 256, 512, 256, 128),
            down_block_types=[
                "DownBlock2D",      # 64→128
                "DownBlock2D",      # 128→256
                "AttnDownBlock2D",  # 256→512 + Attention
                "DownBlock2D",      # 512→256
                "DownBlock2D",      # 256→128
                "DownBlock2D",      # 128→64
            ],
            up_block_types=[
                "UpBlock2D",        # 128→256 (skip from down4)
                "UpBlock2D",        # 256→512 (skip from down3)
                "AttnUpBlock2D",    # 512→256 + Attention
                "UpBlock2D",        # 256→128 (skip from down2)
                "UpBlock2D",        # 128→64  (skip from down1)
                "UpBlock2D",        # 64→out  (skip from input)
            ],
            class_embed_type="identity",
            time_embedding_type="positional"
```

```
        )


    def forward(self, x, t, cond):
        class_embed = self.class_embedding(cond.float())
        return self.unet(x, t, class_embed).sample
```

B. Training (train.py)

i. Use argparse to configure parameters, including epochs, batch size, learning rate, and data paths.

```
def get_args():
    parser = argparse.ArgumentParser(description="Train Conditional DDPM on i-CLEVR dataset")
    parser.add_argument('--epochs', type=int, default=200)
    parser.add_argument('--batch_size', type=int, default=64)
    parser.add_argument('--lr', type=float, default=1e-4)
    parser.add_argument('--save_dir', type=str, default='./checkpoints')
    parser.add_argument('--image_dir', type=str, default='./iclevr')
    parser.add_argument('--train_json', type=str, default='./train.json')
    parser.add_argument('--objects_path', type=str, default='./objects.json')
    parser.add_argument('--test_json', type=str, default='./test.json')
    parser.add_argument('--new_test_json', type=str, default='./new_test.json')
    parser.add_argument('--test_batch_size', type=int, default=64)
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--device', type=str, default='cuda' if torch.cuda.is_available() else 'cpu')
    return parser.parse_args()
```

ii. During the training phase, after loading the training dataset and initializing the model, I use AdamW as the optimizer and manage the noise schedule with Hugging Face Diffusers' DDPMScheduler [1][2]. Here, I set num_train_timesteps = 1000, meaning both the forward and reverse processes use 1000 time steps. I also employ the cosine schedule beta_schedule = 'squaredcos_cap_v2', which allows the model to learn denoising more robustly during training and remove noise precisely during generation, improving final image quality. Finally, I use the evaluation_model evaluator to compute the classification accuracy of the generated images and apply a OneCycleLR scheduler for dynamic learning-rate adjustment.

```
def train(args):
    torch.manual_seed(args.seed)
    os.makedirs(args.save_dir, exist_ok=True)
    dataloader = DataLoader( # 建立訓練集 DataLoader，包含影像與標籤
        ICLEVRDataset(args.train_json, args.objects_path, args.image_dir, is_train=True),
        batch_size=args.batch_size, shuffle=True, num_workers=4
    )

    # 初始化模型及訓練相關工具
    model = ConditionalDDPMModel().to(args.device)
    optimizer = torch.optim.AdamW(model.parameters(), lr=args.lr)
    scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='squaredcos_cap_v2')
```

```
evaluator = evaluation_model()
lr_scheduler = OneCycleLR( # 動態學習率：OneCycleLR
    optimizer,
    max_lr=args.lr,
    steps_per_epoch=len(dataloader),
    epochs=args.epochs,
    pct_start=0.3,
    anneal_strategy='cos'
)
model.optimizer = optimizer
```

iii. Start training:

- **torch.randn_like(images)** returns a new tensor (noise) with the same shape as images, sampled from a standard normal distribution (mean 0, std 1).
- **torch.randint(0, 1000, (batch_size,))** randomly selects a time step t in [0, 1000) for each image in the batch, producing an integer tensor timesteps of length batch_size, which indicates the diffusion stage to simulate or denoise.
- **scheduler.add_noise(images, noise, timesteps)** injects the sampled noise into the original images according to each time step, yielding noisy_images at the corresponding noise levels for training the denoiser.
- Zero the gradients, compute the MSE loss between the model's predicted noise and the actual noise, then backpropagate and update model parameters with AdamW while adjusting the learning rate via OneCycleLR. Finally, accumulate the batch loss for monitoring and analysis.

```
loss_list, acc_list, lr_list = [], [], [] # 記錄各項指標
best_acc = 0.0
print("Start training...")
for epoch in range(1, args.epochs + 1):
    model.train()
    running_loss = 0.0
    for images, labels in tqdm(dataloader, desc=f"Epoch {epoch}/{args.epochs}", leave=False):
        images, labels = images.to(args.device), labels.to(args.device)

        # 隨機噪聲與時間步
        noise = torch.randn_like(images).to(args.device)
        timesteps = torch.randint(0, scheduler.config.num_train_timesteps, (images.size(0),),
                                  device=args.device).long()
        noisy_images = scheduler.add_noise(images, noise, timesteps) # 添加噪聲到影像
        optimizer.zero_grad()
        loss = F.mse_loss(model(noisy_images, timesteps, labels), noise) # 訓練目標：預測噪聲
        loss.backward()
        optimizer.step()
        lr_scheduler.step()
        running_loss += loss.item()
```

iv. At the end of each epoch, compute and record that round's average training loss (avg_loss) and the current learning rate (current_lr), appending them to loss_list and lr_list, respectively. Then call evaluate on both test.json and new_test.json, append the returned average accuracy to acc_list, and—if it exceeds best_acc—update the best accuracy and save the model. After all epochs finish, use plot_metrics to plot the loss, accuracy, and learning rate curves over epochs.

```
        avg_loss = running_loss / len(dataloader)
        current_lr = lr_scheduler.get_last_lr()[0]
        loss_list.append(avg_loss)
        lr_list.append(current_lr)
        # 同時使用 test.json 與 new_test.json 評估模型
        acc = evaluate(model, scheduler, evaluator, args, epoch, avg_loss, current_lr)
        acc_list.append(acc)
        if acc >= best_acc:
            best_acc = acc
            save_best_model(model, args.save_dir, epoch)

    plot_metrics(range(1, args.epochs + 1), loss_list, acc_list, lr_list) # 繪製訓練曲線
    print("Training complete. Metrics plot saved to training_metrics.png")
```

v. The evaluate function works as follows: it runs the full "generate-from-noise → denoise → classifier-scoring" pipeline first on test.json and then on new_test.json, computes the multilabel accuracy for each dataset, and then averages those two accuracies to produce the overall evaluation score for that epoch.

```
def evaluate(model, scheduler, evaluator, args, epoch, avg_loss, current_lr):
    acc_test = evaluate_dataset(model, scheduler, evaluator, args, args.test_json)
    acc_new_test = evaluate_dataset(model, scheduler, evaluator, args, args.new_test_json)
    avg_acc = (acc_test + acc_new_test) / 2.0 # 計算兩者的平均準確度
    print(f"Epoch {epoch:03} | Loss: {avg_loss:.4f} | "
          f"Test Acc: {acc_test:.4f} | New Test Acc: {acc_new_test:.4f} | "
          f"Avg Acc: {avg_acc:.4f} | LR: {current_lr:.6f}")
    return avg_acc


def evaluate_dataset(model, scheduler, evaluator, args, json_path):
    loader = DataLoader( # 根據指定的 JSON 檔建立只含標籤的 DataLoader
        ICLEVRDataset(json_path=json_path, objects_path=args.objects_path, image_dir=args.image_dir,
                      is_train=False),
        batch_size=args.test_batch_size, shuffle=False, num_workers=4
    )

    model.eval()
    total_acc = 0.0
    # 遍歷每個 batch 的標籤，生成影像並計算準確度
    for labels in loader: # 對每批標籤生成影像並評估
        labels = labels.to(args.device)
```

```
        generated_images = torch.randn(labels.size(0), 3, 64, 64).to(args.device) # 從純噪聲開始生成影像
        # 依序執行反向去噪步驟，從最高 timestep 到最低
        for timestep in tqdm(scheduler.timesteps, desc="Denoising", leave=False):
            # 為整個 batch 建立一個值全為 timestep 的長度向量
            t_tensor = torch.full((labels.size(0),), timestep, device=labels.device, dtype=torch.long)
            with torch.no_grad():
                pred_noise = model(generated_images, t_tensor, labels)

            # 根據模型預測的噪聲更新樣本，得到上一個時間步的影像
            generated_images = scheduler.step(pred_noise, timestep, generated_images).prev_sample

        # 用 evaluator（ResNet18）對生成影像做多標籤預測，累加本 batch 的準確度
        total_acc += evaluator.eval(generated_images, labels)
    avg_acc = total_acc / len(loader) # 計算整個資料集的平均準確度並回傳
    return avg_acc
```

C. Inference (inference.py)

i. Execute the run() function, in which **test_model_on_conditions** runs separate tests on **test.json** and **new_test.json** and generates the corresponding images. Finally, **sample_denoising_process** visualizes the denoising process for the label set **["red sphere", "cyan cylinder", "cyan cube"]**.

```
def run(self):
    # 對兩組 JSON 執行測試
    for json_path, prefix in [(self.args.test_json, "test"), (self.args.new_test_json, "new_test")]:
        dataset = ICLEVRDataset(json_path=json_path, objects_path=self.args.objects_json,
                                image_dir="", is_train=False)
        dataloader = DataLoader(dataset, batch_size=self.args.batch_size, shuffle=False)
        self.objects_map = dataset.objects_map
        self.test_model_on_conditions(dataloader, prefix) # 執行條件生成測試：生成影像、儲存與評估
    self.sample_denoising_process() # 所有測試完成後，執行去噪過程範例並儲存中間結果
```

ii. The test_model_on_conditions function is used to verify the model's generative performance under different conditioning inputs. It sequentially generates images for the entire test dataset and evaluates their accuracy using the evaluator. Additionally, all generated images are saved both as a combined grid and as individual image files.

```
def test_model_on_conditions(self, dataloader, prefix):
    # 以指定條件 (prefix) 測試模型並儲存結果
    print(f"\n生成 {prefix} 圖片中...")
    out = os.path.join(self.images_root, prefix)
    os.makedirs(out, exist_ok=True)

    # 同時跑完所有 batch 並收集生成的影像與對應標籤
    batches = [(self.sample_images(b.to(self.device)), b.to(self.device))
            for b in tqdm(dataloader, desc=f"Generating {prefix}")]
```

```
    imgs, labels = torch.cat([i for i,_ in batches]), torch.cat([l for _,l in batches])

    # 存整組合併 grid
    gp = os.path.join(self.args.sample_dir, f"{prefix}_grid.png")
    save_image(make_grid(imgs * 0.5 + 0.5, nrow=8), gp)
    print(f"已儲存 {prefix} 合成圖至 {gp}")

    # 存每張小圖
    for i, img in enumerate(imgs):
        save_image(img * 0.5 + 0.5, f"{out}/{i}.png")

    # 使用 evaluator 計算準確度並回傳
    acc = self.evaluator.eval(imgs, labels)
    print(f"Evaluator Accuracy on {prefix}: {acc:.4f}")
    return acc
```

iii. The sample_images function is the core generation step of the diffusion model. It begins from Gaussian noise and iteratively restores the image using the model's predicted noise (reverse diffusion) until a condition-compliant image is produced. This implementation supports both DDPM and DDIM samplers and applies different denoising strategies based on the user's choice.

```
def sample_images(self, cond_batch):
    # 根據條件向量批次生成影像張量
    bsz = cond_batch.size(0)
    gen_img = torch.randn(bsz, 3, self.args.image_size, self.args.image_size,
                          device=self.device)
    # 逐步去噪
    for t in self.scheduler.timesteps:
        t_tensor = torch.full((bsz,), t, device=self.device, dtype=torch.long)
        with torch.no_grad():
            noise = self.model(gen_img, t_tensor, cond_batch)
            out = self.scheduler.step(noise, t, gen_img, eta=self.args.eta) \
                if self.args.sampler == 'ddim' else self.scheduler.step(noise, t, gen_img)
            gen_img = out.prev_sample
    return gen_img.clamp(-1, 1)
```

iv. In the sample_denoising_process section, for the Lab 6 requirement of **["red sphere", "cyan cylinder", "cyan cube"]**, evenly spaced intermediate denoising outputs are extracted. These intermediates are then assembled into an image grid to visualize the model's step-by-step denoising capability.

```
def sample_denoising_process(self):
    # 構造指定物件條件向量，選擇 red sphere、cyan cylinder、cyan cube
    cond = torch.zeros(len(self.objects_map), device=self.device)
    for name in ["red sphere", "cyan cylinder", "cyan cube"]:
        cond[self.objects_map[name]] = 1.0
```

```
cond = cond.unsqueeze(0)  # shape: [1, cond_dim]

# 初始化純隨機噪聲樣本 x，shape: [1, 3, H, W]
x = torch.randn(1, 3, self.args.image_size, self.args.image_size, device=self.device)
# 選擇去噪步驟函式
if self.args.sampler == 'ddim':
    step_fn = lambda noise, t, x: self.scheduler.step(noise, t, x, eta=self.args.eta)
else:
    step_fn = lambda noise, t, x: self.scheduler.step(noise, t, x)

# 擷取等間隔的中間去噪結果
timesteps = self.scheduler.timesteps
step_size = max(1, len(timesteps) // 10)
images = []
for i, t in enumerate(timesteps):
    t_tensor = torch.full((1,), t, device=self.device, dtype=torch.long)
    with torch.no_grad():
        noise = self.model(x, t_tensor, cond)
        x = step_fn(noise, t, x).prev_sample
    if i % step_size == 0 or i + 1 == len(timesteps):
        images.append(x.clamp(-1, 1))

grid = make_grid(torch.cat([img * 0.5 + 0.5 for img in images]), nrow=len(images))
save_image(grid, os.path.join(self.args.sample_dir, "denoising_process.png"))
print(f"去噪過程儲存至 {self.args.sample_dir}/denoising_process.png")
```

3. Results and discussion

A. Show your synthetic image grids and a denoising process image with the label set ["red sphere", "cyan cylinder", "cyan cube"].

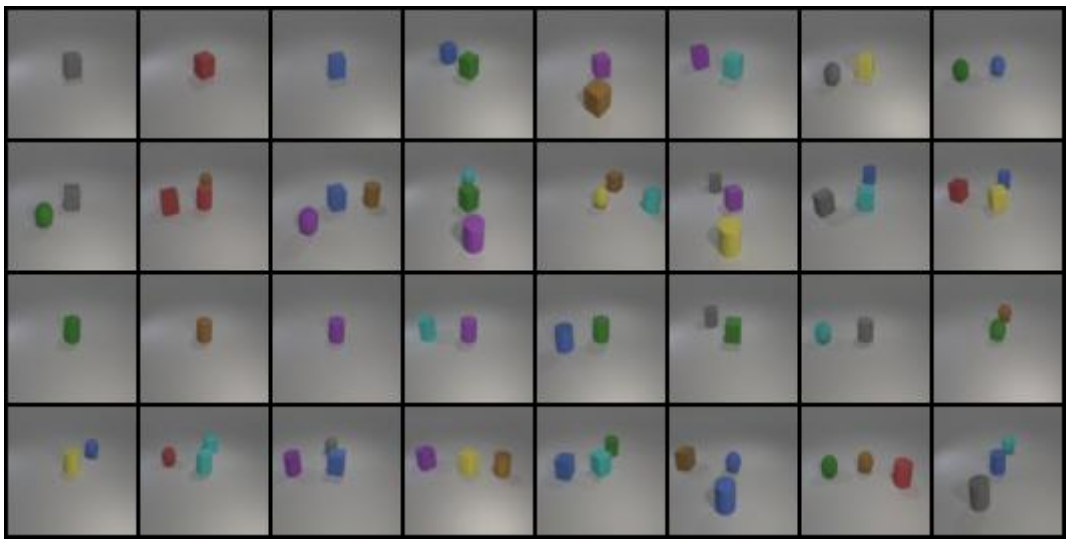

Fig.1 synthetic image grids (test)

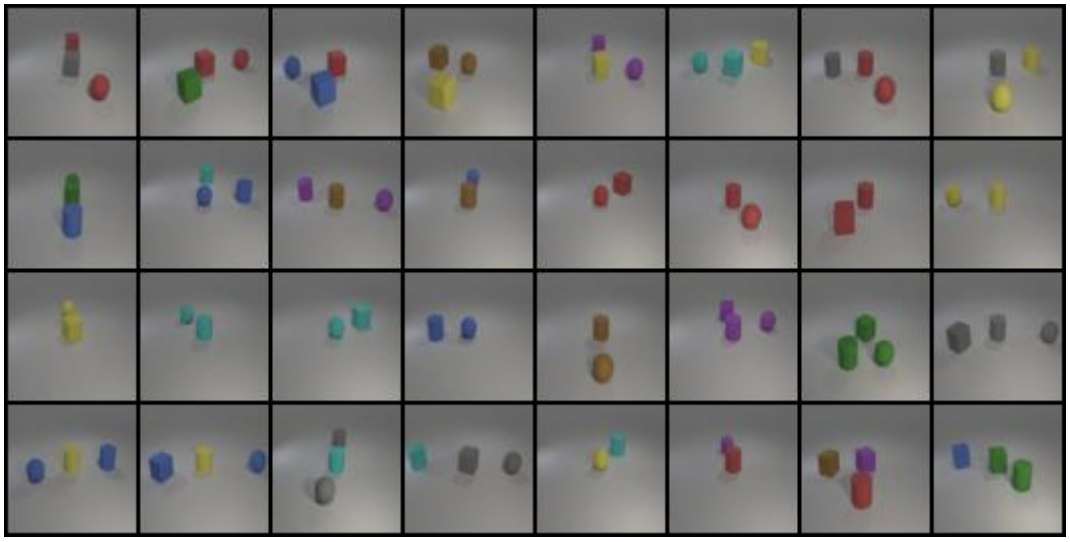

Fig.2 synthetic image grids (new_test)

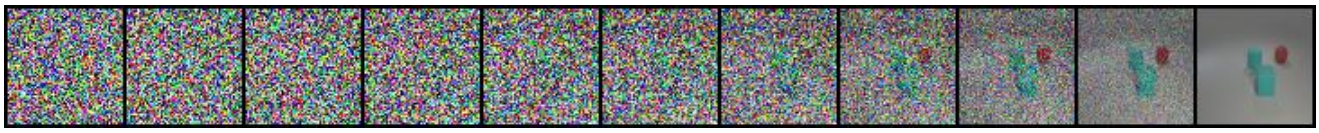

Fig.3 denoising process image (red sphere, cyan cylinder, cyan cube)

B. Discussion of your extra implementations or experiments.

i. Use the OneCycleLR dynamic learning rate during training:

To help the model avoid local minima and achieve more stable convergence and improved performance, I selected the OneCycleLR scheduler. As shown in the code below, during the first 30% of the epochs, OneCycleLR gradually increases the learning rate, which assists the model in escaping local minima and exploring a broader parameter space. Then, over the remaining 70% of the epochs, it follows a cosine annealing schedule to gradually decrease the learning rate, enabling the model to converge more steadily toward an optimal solution.

```
lr_scheduler = OneCycleLR(
    optimizer,
    max_lr=args.lr,
    steps_per_epoch=len(dataloader),
    epochs=args.epochs,
    pct_start=0.3,
    anneal_strategy='cos'
)
```

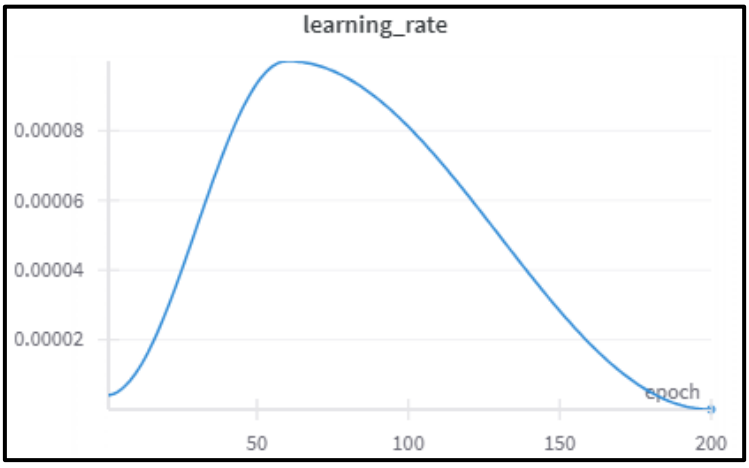

Fig. 4 OneCycleLR learning-rate schedule curve

ii. <u>During inference, I try using both DDPM and DDIM:</u>

I added a DDIMScheduler in inference.py to support DDIM, allowing me to experiment with the differences between DDPM and DDIM. The commands to run are as follows (using beta_schedule = linear as an example):

- **DDPM**

```
python inference.py --checkpoint checkpoints/model_latest.pth --beta_schedule linear --sampler ddpm
```

- **DDIM**

```
python inference.py --checkpoint checkpoints/model_latest.pth --beta_schedule linear --sampler ddim --eta 1
```

Based on the experimental results, **both DDPM and DDIM achieved an accuracy of 0.9444 on test.json and 0.9048 on new_test.json**. However, these accuracies can be higher or lower with different random seeds, and the performance of DDPM and DDIM isn't always exactly the same, but in all cases the accuracy remains above 0.9.

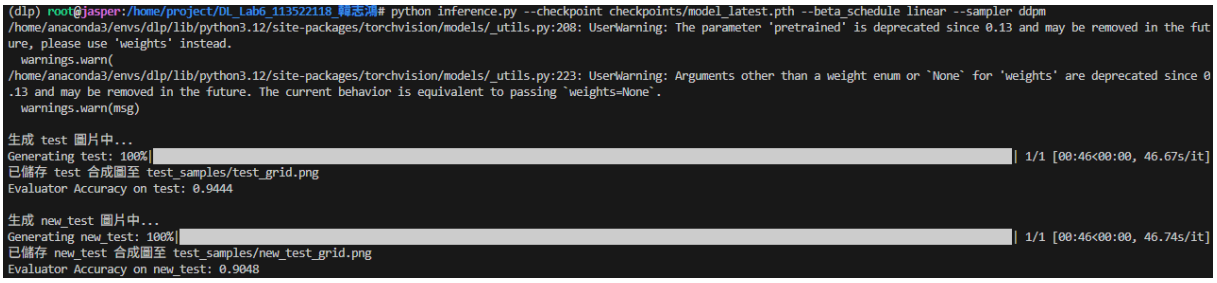

Fig. 5 Accuracy on test.json and new_test.json for DDPM (beta_schedule = linear)

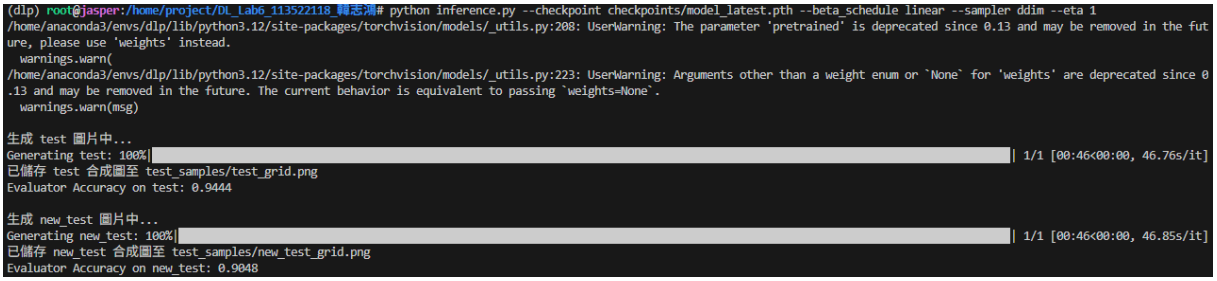

Fig. 6 Accuracy on test.json and new_test.json for DDIM (beta_schedule = linear)

iii. <u>Compare the linear and cosine (squaredcos_cap_v2) beta_schedules in DDPMScheduler:</u>

For DDPM training, I ran two experiments—one with a linear schedule and one with a cosine schedule—keeping all other parameters identical. From the training accuracy curves shown below, we can see that while the model using the linear schedule increases accuracy more quickly in the early stages, the model using the cosine schedule is more stable in the middle and late stages and ultimately achieves a higher average accuracy. This result suggests that the cosine schedule, with its smoother noise-injection process, helps the model learn denoising more stably during training and more effectively remove noise during generation, thereby enhancing final image quality.

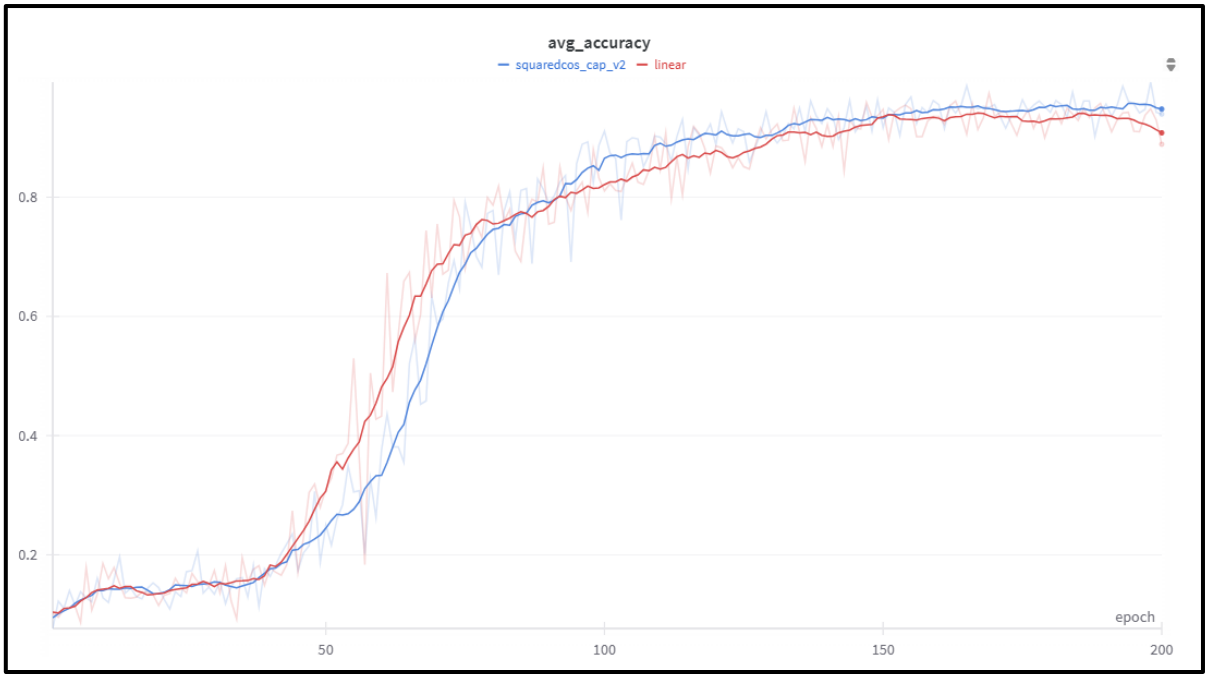

Fig. 7 Training average accuracy using linear and cosine schedules (Curves are smoothed with a running average over 10 points)

4. Experimental results

Classification accuracy on test.json and new test.json. Show your accuracy screenshots.

```
生成 test 圖片中...
Generating test: 100%| | 1/1 [01:40<00:00, 100.43s/it]
已儲存 test 合成圖至 test_samples/test_grid.png
Evaluator Accuracy on test: 0.9722

生成 new_test 圖片中...
Generating new_test: 100%| | 1/1 [01:41<00:00, 101.24s/it]
已儲存 new_test 合成圖至 test_samples/new_test_grid.png
Evaluator Accuracy on new_test: 0.9405
去噪過程儲存至 test_samples/denoising_process.png
```

Fig. 8 Training average accuracy using cosine schedules

5. Reference

[1] Hugging Face Diffusion Models Course, https://github.com/huggingface/diffusion-models-class

[2] Denoising Diffusion Probabilistic Models As a Defense Against Adversarial Attacks, https://hackmd.io/@meow2meow/HyvRZjxBh

[3] 【扩散模型】 DDPM 和 DDIM 讲解, https://blog.csdn.net/aaatomaaa/article/details/133580069

[4] DDIM 简明讲解与 PyTorch 实现：加速扩散模型采样的通用方法, https://zhouyifan.net/2023/07/07/20230702-DDIM/