國立陽明交通大學 深度學習實驗 LAB 6

313553036 多工所 1 葉軒宇

i. Introduction

在實驗六中我們實作條件去噪擴散概率模型（cDDPM），它是一種新興的生成模型，用於基於給定 label 創建合成圖片。這種模型的核心是一個逐步的擴散過程，它能將隨機 noise 逐漸轉變為結構化的圖片。cDDPM 的獨特之處在於其條件生成能力，即模型可以根據提供的 label 資訊來引導圖片生成過程，確保生成的圖片符合指定的特徵。在訓練階段，模型學習如何 reverse 這個擴散過程，同時考慮圖片內容和 label 訊息，從而能夠從 noise 中重建原始圖片。當用於生成新圖片時，cDDPM 從純 noise 開始，通過多次迭代逐步去除 noise，最終產生符合給定 label 的清晰圖片。這種方法的一個主要優勢是能夠生成 high quality、diverse 的圖片，同時在訓練穩定性方面表現良好，為各種應用提供了強大而靈活的圖片生成工具。

ii. Implementation details

DDPM:

在 DDPM 模型架構的部分，我使用了 diffusers library 內的 UNet2DModel 來建立一個 UNet 的模型。它使用 t+1 時間的

noise image 以及 label 當作輸入，來預測圖片的 noise。然後將 t+1 時間的 noise 從圖片中刪除來獲得 t 時間的圖片。重複 timestamps 的次數後即可獲得最終的圖片。

```python
class DDPM_NoisePredictor(nn.Module):
    def __init__(self, n_classes=24, label_embed_size=4096):
        super().__init__()

        self.label_embedding = nn.Embedding(n_classes, label_embed_size)

        self.model = UNet2DModel(
            sample_size=64,
            in_channels=3 + n_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",
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
                "DownBlock2D",
            ),
            up_block_types=(
                "UpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )

    def forward(self, images, timestep, labels):
        batch_size, c, w, h = images.shape

        label_embed = self.label_embedding(labels)

        _, n_classes, label_embed_size = label_embed.shape
        label_embed = label_embed.view(batch_size, n_classes, w, h)

        inputs = torch.cat( tensors: (images, label_embed), dim=1)

        outputs = self.model(inputs, timestep).sample

        return outputs
```

Training step:

先用 IclervDataset、Dataloader 來載入訓練數據，其中每個 epoch 都會打亂數據順序。

```python
train_dataset = IclevrDataset(
    mode="train",
    json_root="data",
    image_root="data/iclevr",
    num_cpus=8,
)

train_loader = DataLoader(
    train_dataset,
    batch_size=args.batch_size,
    shuffle=True,
    num_workers=4,
    pin_memory=True if args.device == 'mps' else False
)
```

接著創建一個 DDPM 的模型、設定 optimizer 與 learning rate scheduler、設定 DDPM scheduler 以及使用 MSE 當作 loss funcion。

```python
ddpm = DDPM_NoisePredictor(n_classes=args.n_classes).to(args.device)
optimizer = torch.optim.AdamW(ddpm.parameters(), lr=args.lr, weight_decay=args.weight_decay)

lr_scheduler = get_cosine_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=args.lr_warmup_steps,
    num_training_steps=(min(len(train_loader), args.max_steps_per_epoch) * args.epochs),
)

ddpm_scheduler = DDPMScheduler(
    num_train_timesteps=args.num_train_timesteps, beta_schedule="squaredcos_cap_v2"
)

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

這部分就是開始訓練循環。先將數據準備好後生成隨機 noise、隨機 timesteps，然後 forward 傳播、backward 傳播以及計算損失，最後再更新模型的 parameter、lr。

```python
for i, (images, labels) in enumerate(progress_bar):
    if i >= args.max_steps_per_epoch:
        break

    batch_size = images.size(0)
    images, labels = images.to(args.device), labels.to(args.device)

    noises = torch.randn_like(images).to(args.device)

    timesteps = torch.randint(
        0,
        ddpm_scheduler.config.num_train_timesteps,
        (batch_size,),
        dtype=torch.int64,
    ).to(args.device)

    noisy_images = ddpm_scheduler.add_noise(images, noises, timesteps)

    pred_noises = ddpm(noisy_images, timesteps, labels)

    loss = criterion(pred_noises, noises)
    loss.backward()

    epoch_loss += loss.item()

    optimizer.step()
    lr_scheduler.step()
    optimizer.zero_grad()

    progress_bar.set_description(
        f"Epoch {epoch}/{args.epochs} [Batch {i}/{min(len(train_loader), args.max_steps_per_epoch)}] [Loss: {loss.item():.4f}]"
    )

writer.add_scalar( tag: "Loss/DDPM", epoch_loss / min(len(train_loader), args.max_steps_per_epoch), epoch)
```

此處主要是定期去對模型進行評估，並將結果記錄到 tensorboard，還有定期保存模型的部分。

```python
if epoch % args.eval_interval == 0:
    with torch.no_grad():
        test_dataset = IclevrDataset(
            mode="test",
            json_root="data",
            image_root="data/iclevr",
            num_cpus=8,
        )
        accuracy, generated_images = evaluate_DDPM(
            args, ddpm, ddpm_scheduler, test_dataset
        )

        image_visualizations = make_grid(generated_images, nrow=8)

        writer.add_scalar( tag: "Evaluation Accuracy", accuracy, epoch)
        writer.add_image( tag: "Generated Images", image_visualizations, epoch)

if epoch % 10 == 0:
    torch.save(
         obj: {
            "ddpm": ddpm.state_dict(),
            "ddpm_scheduler": ddpm_scheduler,
        },
         f: f"{args.output_dir}/ddpm_epoch{epoch}.pth",
    )
elif epoch == args.epochs - 1:
    torch.save(
         obj: {
            "ddpm": ddpm.state_dict(),
            "ddpm_scheduler": ddpm_scheduler,
        },
         f: f"{args.output_dir}/ddpm.pth",
    )
```

在訓練的最後根據 test、new test 兩種模式來測試結果並印出準確率以及具有網格的圖片。

```
for mode in ["test", "new_test"]:
    test_dataset = IclevrDataset(
        mode=mode,
        json_root="data",
        image_root="data/iclevr",
        num_cpus=8,
    )

    accuracy, generated_images = evaluate_DDPM(
        args, ddpm, ddpm_scheduler, test_dataset
    )

    print(f"Accuracy for {mode}: {accuracy}")

    image_visualizations = make_grid(generated_images, nrow=8)

    save_image(image_visualizations, fp: f"{args.output_dir}/{mode}_ddpm_result.png")
```

Test(evaluate) step:

在 evaluate 裡面，先採樣一個 random noise 並獲取 label 作為 diffusion model 的 input。後面就跟 training stage 相同的步驟來預測 noise，並逐步應用 DDPM model 來去除 noise 並生成圖片。

```python
def evaluate_DDPM(args, ddpm, ddpm_scheduler, test_dataset):
    evaluator = evaluation_model()

    test_loader = DataLoader(
        test_dataset, batch_size=args.batch_size, shuffle=False, num_workers=8
    )

    generated_images = []
    total_accuracy = 0.0

    for i, labels in enumerate(test_loader):
        batch_size = labels.size(0)
        labels = labels.to(args.device)

        images = torch.randn(batch_size, 3, 64, 64).to(args.device)
        for i, timestep in enumerate(ddpm_scheduler.timesteps):
            with torch.no_grad():
                pred_noise = ddpm(images, timestep, labels)
            images = ddpm_scheduler.step(pred_noise, timestep, images).prev_sample

        accuracy = evaluator.eval(images, labels)
        total_accuracy += accuracy

        denormalized_images = (images / 2 + 0.5).clamp( min: 0,  max: 1)
        generated_images.extend(denormalized_images)

    accuracy = total_accuracy / len(test_loader)

    return accuracy, generated_images
```

這個部分是根據作業的要求去設定特定的 label(["red sphere", "cyan cylinder", "cyan cube"])。然後跟前面的步驟相同來生成圖片，其中每 100 步保存一次結果。

```python
def show_DDPM_denoising_process(args, ddpm, ddpm_scheduler):
    # 指定標籤 ["red sphere", "cyan cylinder", "cyan cube"]
    label_names = ["red sphere", "cyan cylinder", "cyan cube"]

    with open(os.path.join(args.data_path, "objects.json"), "r") as f:
        objects_dict = json.load(f)

    label = torch.zeros(24, dtype=torch.long)
    for name in label_names:
        label[objects_dict[name]] = 1

    label = label.unsqueeze(0).to(args.device)

    batch_size = 1
    total_timesteps = len(ddpm_scheduler.timesteps)
    denoising_process_images = []

    current_image = torch.randn(batch_size, 3, 64, 64).to(args.device)

    for i, timestep in enumerate(ddpm_scheduler.timesteps):
        with torch.no_grad():
            pred_noise = ddpm(current_image, timestep, label)
        current_image = ddpm_scheduler.step(
            pred_noise, timestep, current_image
        ).prev_sample

        if i % 100 == 0 or i == total_timesteps - 1:
            denormalized_image = (current_image[0] / 2 + 0.5).clamp(0, 1)
            denoising_process_images.append(denormalized_image)

    return denoising_process_images
```

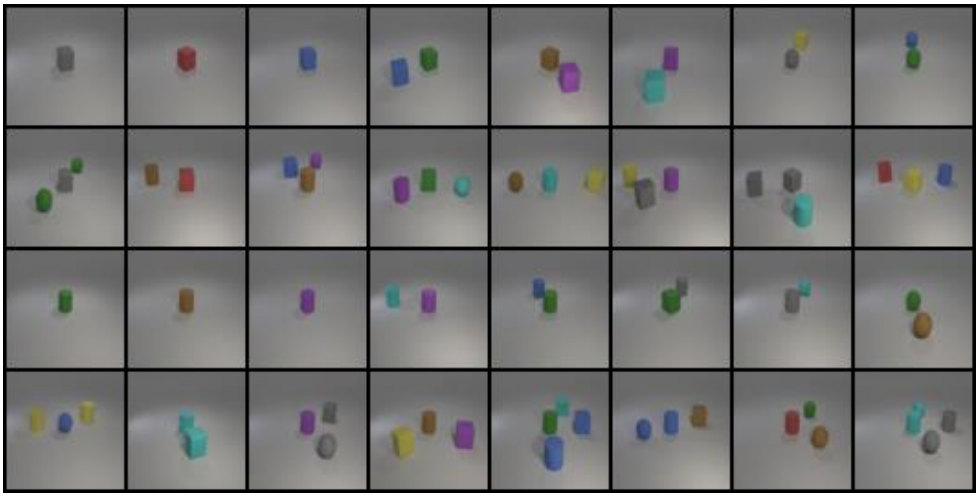
Test 即對兩種不同的測試集去進行評估

```python
def test_DDPM(args, ddpm, ddpm_scheduler):
    for mode in ["test", "new_test"]:
        test_dataset = IclevrDataset(
            mode=mode,
            json_root=args.data_path,
            image_root=os.path.join(args.data_path, "iclevr"),
            num_cpus=8,
        )

        accuracy, generated_images = evaluate_DDPM(
            args, ddpm, ddpm_scheduler, test_dataset
        )

        print("---------------------------------")
        print(f"Accuracy for {mode}: {round(accuracy * 100, 2)}%")

        image_visualizations = make_grid(generated_images, nrow=8)

        save_image(image_visualizations, fp: f"{args.output_dir}/{mode}_ddpm_result.png")

    denoising_process_images = show_DDPM_denoising_process(args, ddpm, ddpm_scheduler)

    denoising_process_images_grid = make_grid(
        denoising_process_images, nrow=len(denoising_process_images)
    )

    save_image(
        denoising_process_images_grid, fp: f"{args.output_dir}/ddpm_denoising_process.png"
    )
```

iii. Results and discussion

Synthetic image grids:

Test

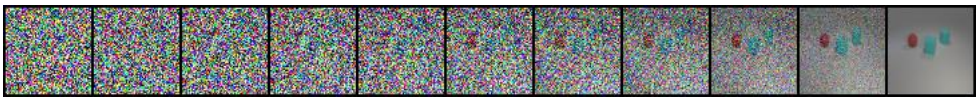


New_test

Denoising process image:

可以看到最終生成了如 label 相同的

Red sphere, cyan cylinder, cyan cube



Test accuracy:

```
Checkpoint keys: dict_keys(['ddpm', 'ddpm_scheduler'])
Loaded DDPM state from checkpoint.
Loaded scheduler from checkpoint.
--------------------------------
Accuracy for test: 79.17%
--------------------------------
Accuracy for new_test: 76.19%
```

Discussion of your extra implementation or experiments:

我嘗試 timestep = 1000 了之後，發現其實在準確率並沒有很大的差別。

```
Checkpoint keys: dict_keys(['ddpm', 'ddpm_scheduler'])
Loaded DDPM state from checkpoint.
Loaded scheduler from checkpoint.
--------------------------------
Accuracy for test: 77.78%
--------------------------------
Accuracy for new_test: 79.76%
```

Reference:

https://huggingface.co/docs/diffusers/en/api/schedulers/ddpm

https://huggingface.co/docs/diffusers/en/api/models/unet2d