Lab6 - Generative Models Experiment Report

313551055 柯柏旭

Report part:

Introduction

在這份報告中，探討了基於UNet架構的條件生成模型在多標籤圖像生成任務中的實現與評估。實驗中使用了一個定制的數據集 (iclevr) 來訓練去噪擴散概率模型（DDPM），目的是生成特定物體標籤條件下的圖像。

報告詳細介紹了模型架構、訓練過程和數據處理的技術細節，並討論了實驗中遇到的挑戰，如模型複雜度和學習穩定性。並且還討論了為了提升性能所進行的模型改進和訓練策略，最後對結果進行了詳細分析，評估了這些改進的有效性。

Implementation details

Usage

`train.py` 用來訓練 ddpm 模型

`sample.py` 會使用訓練好的 ddpm 模型與 label(`*test.json`) 去生成圖片

`evaluate.py` 會使用作業提供的 `evaluator.py` 與 `*test.json` 來評估 `sample.py` 生成的圖片

Dataloader

```
class CLEVRDataset(Dataset):
    def __init__(self, json_file, root_dir, object_mapping,
transform=None):
        self.root_dir = root_dir
        self.transform = transform

        # 讀取 json 文件
        with open(json_file, 'r') as f:
            self.data = json.load(f)
```

```
        # 獲取所有圖片名稱
        self.image_names = list(self.data.keys())

        # 將物件名稱轉換為索引
        self.object_mapping = object_mapping

    def __len__(self):
        return len(self.image_names)

    def __getitem__(self, idx):
        # 獲取圖片名稱和其對應的標籤
        img_name = self.image_names[idx]
        label_names = self.data[img_name]

        # 讀取圖片
        img_path = os.path.join(self.root_dir, img_name)
        image = Image.open(img_path).convert("RGB")

        # 如果有transform，進行圖片轉換
        if self.transform:
            image = self.transform(image)

        # 將物件名稱轉換為 one-hot 向量
        labels = torch.zeros(len(self.object_mapping))  # 創建一個長度為
24 的零向量
        for label in label_names:
            labels[self.object_mapping[label]] = 1  # 將對應的索引位置設
置為 1

        return image, labels

def custom_collate_fn(batch):
    images, labels = zip(*batch)

    # 将图片堆叠为一个张量
    images = torch.stack(images, 0)
```

```
    # 将标签堆叠为一个张量
    labels = torch.stack(labels, 0)

    return images, labels

def get_dataloader(json_file, root_dir, object_mapping,
batch_size=32, num_workers=4, shuffle=True):
    transform = transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    dataset = CLEVRDataset(json_file=json_file, root_dir=root_dir,
object_mapping=object_mapping, transform=transform)

    dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=shuffle, num_workers=num_workers,
collate_fn=custom_collate_fn)

    return dataloader
```

`Dataloader.py` 定義了一個名為 CLEVRDataset 的自訂 PyTorch 資料集類別，用來處理特定的影像和標籤資料。程式碼首先通過讀取一個 JSON 檔案來載入圖片名稱及其對應的物件標籤。

每個標籤是用一個物件名稱到索引的映射（object_mapping）轉換成 one-hot 向量。

`__getitem__` 函數會根據索引讀取圖片，進行轉換，並返回圖片及其對應的 one-hot 編碼標籤。程式碼還定義了一個自訂的 collate_fn 函數，用於在批次處理時將圖片和標籤堆疊成張量。最後，使用 get_dataloader 函數創建一個 DataLoader，用於迭代加載資料。

Model

參考 https://github.com/huggingface/diffusion-models-class 的模型部分並做改良 (提高複雜度、調整輸入維度)

```
class MultiLabelConditionedUnet(nn.Module):
    def __init__(self, num_classes=24, class_emb_size=4):
```

```
        super().__init__()

        # 定義一個嵌入層，將類別標籤映射為大小為 class_emb_size 的嵌入向量
        # 這裡的 num_classes 是類別的總數（24個），class_emb_size 是嵌入向量的維度
        self.label_embedding = nn.Embedding(num_classes, class_emb_size)

        # # 使用全連接層將 one-hot 編碼向量轉換為 class embedding
        # self.class_embedding = nn.Sequential(
        #     nn.Linear(num_classes, class_emb_size * 64 * 64),  # 將 one-hot 編碼展開到整個影像維度
        #     nn.ReLU()
        # )

        # 定義UNet模型，這裡的UNet用於圖像生成，接受額外的類別條件信息
        self.model = UNet2DModel(
            sample_size=64,           # 設定輸入影像的解析度為64x64
            in_channels=3 + num_classes,  # 設定輸入通道數，3個RGB通道加上one-hot類別條件信息的通道數
            out_channels=3,           # 設定輸出通道數，這裡輸出為RGB圖像，所以是3個通道
            time_embedding_type="positional",  # 使用位置嵌入來處理時間步長
            layers_per_block=2,       # 每個UNet區塊使用2層ResNet
            block_out_channels=(128, 128, 256, 256, 512, 512),  # 定義每個UNet區塊的輸出通道數
            down_block_types=(
                "DownBlock2D",  # 定義下采樣區塊類型，這裡是普通的ResNet下采樣
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",  # 添加帶有空間自注意力的ResNet下采樣區塊
                "DownBlock2D",
            ),
```

```
            up_block_types=(
                "UpBlock2D",  # 定義上采樣區塊類型，這裡是普通的ResNet上采樣
                "AttnUpBlock2D",  # 添加帶有空間自注意力的ResNet上采樣區塊
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )

    def forward(self, x, t, class_labels):
        # x 的形狀：批次大小 (bs), 通道數 (ch), 寬度 (w), 高度 (h)
        bs, ch, w, h = x.shape

        # 將one-hot類別標籤轉換為與影像相同大小的張量，用於條件輸入
        # class_labels的形狀為(bs, num_classes)，這裡先調整維度再擴展成(bs, num_classes, w, h)
        class_cond = class_labels.view(bs, class_labels.shape[1], 1, 1).expand(bs, class_labels.shape[1], w, h)

        # 將影像x與條件信息class_cond在通道維度上拼接
        # 拼接後的net_input形狀為(bs, 3 + num_classes, 64, 64)
        net_input = torch.cat((x, class_cond), 1)

        # 將拼接後的輸入傳入UNet模型進行前向傳播，並返回生成的影像
        return self.model(net_input, t).sample  # 返回形狀為(bs, 3, 64, 64)的輸出影像
```

`model.py` 定義了一個名為 MultiLabelConditionedUnet 的 PyTorch 模型類別，用於多標籤條件下的圖像生成。

該模型使用了一個嵌入層將類別標籤映射為固定大小的嵌入向量，並通過一個改良的 UNet 模型來生成影像。在這個過程中，模型會將類別標籤作為條件信息與輸入影像拼接，然後通過 UNet 進行處理。

UNet 中的不同區塊使用了不同的下采樣和上采樣策略，包括普通的 ResNet 區塊以及帶有空間自注意力的區塊。最終模型會生成與輸入尺寸相同的影像，這些影像包含了條件類別的相關信息。

詳細內容呈現於上面程式碼的註解中。

Training

```
# 創建噪聲調度器
noise_scheduler = DDPMScheduler(num_train_timesteps=1000,
beta_schedule='squaredcos_cap_v2')

train_dataloader = get_dataloader(train_json, dataset_path,
object_mapping, batch_size=32, shuffle=True)

# 設定訓練迭代次數
n_epochs = 250

# 創建模型並將其移動到設備上
net = MultiLabelConditionedUnet(num_classes=24,
class_emb_size=4).to(device)

# 定義損失函數
loss_fn = nn.MSELoss()

# 定義優化器
opt = torch.optim.Adam(net.parameters(), lr=5e-4)

# 定義學習率調度器
scheduler = StepLR(opt, step_size=8, gamma=0.8)

# 保留損失值以便稍後查看
losses = []

# 訓練迴圈
for epoch in range(n_epochs):
    running_loss = 0.0
    for x, y in tqdm(train_dataloader, desc=f"Epoch
{epoch+1}/{n_epochs}"):

        # 獲取數據並準備添加噪聲的版本
        x = x.to(device)  # 將數據移動到 GPU
        y = y.to(device)
```

```
            noise = torch.randn_like(x)
            timesteps = torch.randint(0, 999,
(x.shape[0],)).long().to(device)
            noisy_x = noise_scheduler.add_noise(x, noise, timesteps)

            # 獲得模型的預測
            pred = net(noisy_x, timesteps, y)  # 注意我們傳入了標籤 y

            # 計算損失
            loss = loss_fn(pred, noise)  # 輸出與噪聲的接近程度

            # 反向傳播並更新參數
            opt.zero_grad()
            loss.backward()
            opt.step()

            # 儲存損失值以便稍後查看
            losses.append(loss.item())
            running_loss += loss.item()

            # 更新進度條中的當前損失
            tqdm.write(f"Batch loss: {loss.item():.4f}")

        # 每個 epoch 結束後更新學習率
        scheduler.step()
```

`train.py` 用於訓練一個使用條件生成網絡 (UNet) 的多標籤圖像生成模型。在這裡，程式碼首先設置了 device 和 noise scheduler，並讀取訓練數據。模型結構為 MultiLabelConditionedUnet，訓練過程使用均方誤差 (MSE) 作為損失函數，並通過 Adam 優化器進行參數更新。

學習率調度器 (StepLR) 用於每8個epoch將學習率減少20%。訓練過程中，每個 epoch 都會計算並儲存損失，並且每10個 epoch 保存一次模型的權重。最後，訓練結束後，程式碼會繪製並保存損失曲線。

在上面的程式碼中，使用了**DDPM (Denoising Diffusion Probabilistic Model)** 來進行圖像生成。

- Forward process: 在前向過程中，模型從一張圖逐步添加高斯噪聲，使得最終的圖像變成純噪聲。
- Backward process: 逆向過程則是模型的生成過程，目標是從純噪聲逐步去除噪聲，重建出清晰的圖像。

至於 noise scheduler 使用了 diffusers 的 DDPMScheduler 用來控制在每個時間步中添加或去除的噪聲量。

- num_train_timesteps=1000：這表示將訓練過程分成 1000 個時間步，每一步都會添加或去除一定的噪聲。
- beta_schedule='squaredcos_cap_v2'：這是一種特殊的 \beta 值調度方式，使用一個餘弦平方函數來控制噪聲的增減，這種方式通常能夠更穩定地生成高質量的圖像。

Sampling

```
seed = 42

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.random.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True

# 設置設備
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 加載已訓練的模型
net = MultiLabelConditionedUnet(num_classes=24,
class_emb_size=4).to(device)
net.load_state_dict(torch.load('/home/hentci/code/NYCU-
DLP/lab6/64x64_saved_models/model_epoch_250.pth'))
net.eval()

# 創建噪聲調度器
noise_scheduler = DDPMScheduler(num_train_timesteps=1000,
beta_schedule='squaredcos_cap_v2')
```

```
# 準備生成樣本
generated_images = []

# 創建保存去噪過程的文件夾
os.makedirs('denoise_process', exist_ok=True)

for idx, labels in enumerate(tqdm(test_data, desc="Generating
samples")):
    print(labels)

    # 將標籤轉換為 one-hot 編碼
    y = torch.zeros((1, len(object_mapping)), device=device)  # 初始化
one-hot 向量
    for label in labels:
        label_idx = object_mapping[label]
        y[0, label_idx] = 1

    print(y)

    # 準備隨機噪聲作為起點
    x = torch.randn(1, 3, 64, 64).to(device)  # 根據你的圖片尺寸設定

    denoise_images = []  # 儲存去噪過程中的中間結果

    # 生成迴圈
    for i, t in enumerate(noise_scheduler.timesteps):
        with torch.no_grad():
            residual = net(x, t, y)  # 傳入 one-hot 編碼的標籤 y

        # 更新樣本
        x = noise_scheduler.step(residual, t, x).prev_sample

        # 每隔一定的步驟保存中間結果
        if i % 100 == 0 or i == len(noise_scheduler.timesteps) - 1:
            denoise_images.append(x.clone())
```

```
    # 將生成的圖像轉換為PIL格式並保存
    generated_image = ((x + 1) / 2).clamp(0,
1).cpu().squeeze().permute(1, 2, 0).numpy() * 255
    generated_image =
Image.fromarray(generated_image.astype('uint8'))
    generated_images.append(generated_image)

    # 保存生成的最終圖片
    generated_image.save(f'generated_images/sample_{idx}.png')

    # 保存去噪過程圖像
    denoise_images = [(img + 1) / 2 for img in denoise_images]  # 從
[-1, 1] 映射回 [0, 1]
    denoise_images = torch.cat(denoise_images, dim=0)
    grid = make_grid(denoise_images, nrow=len(denoise_images))
    grid = grid.permute(1, 2, 0).cpu().numpy() * 255
    grid_image = Image.fromarray(grid.astype('uint8'))
    grid_image.save(f'denoise_process/denoise_process_{idx}.png')

    print('save image: ', idx)


print("Image generation and denoise process saving completed.")
```

`sample.py` 實現了利用事先訓練好的多標籤條件式 UNet 模型進行圖像生成的過程。

首先，設置了隨機種子來確保結果的可重現性，並選擇適當的計算設備（如 GPU）。接著，從本地加載已經訓練好的模型權重，並設定模型進行評估模式。然後，使用 DDPMScheduler 創建一個噪聲調度器，它會控制圖像生成過程中的噪聲添加和去除的步驟。

在生成過程中，程式為每組標籤創建一個隨機噪聲的初始圖像，並且在每個時間步長中，模型會根據當前的圖像和標籤預測去噪後的殘差。這個殘差會被用來更新圖像，最終生成一張較為清晰的圖像。

Evaluating

```
seed = 42

random.seed(seed)
```

```
np.random.seed(seed)
torch.manual_seed(seed)
torch.random.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True

# Function to denormalize the images for visualization
def denormalize(tensor):
    return tensor * torch.tensor((0.5, 0.5, 0.5)).view(3, 1, 1) +
torch.tensor((0.5, 0.5, 0.5)).view(3, 1, 1)

# Load test.json and define object_mapping
with open('./new_test.json', 'r') as f:
    test_data = json.load(f)

# Function to convert labels to one-hot encoding
def labels_to_one_hot(labels, object_mapping):
    one_hot = torch.zeros(len(object_mapping))
    for label in labels:
        one_hot[object_mapping[label]] = 1
    return one_hot

# Directory containing the generated images
image_dir = '/home/hentci/code/NYCU-
DLP/lab6/new_test_generated_images'

# Load and preprocess images
images = []
labels = []

transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

```
# Iterate over the test data and corresponding image files
for idx, label_list in enumerate(test_data):
    print(idx, ':',label_list)
    img_name = f"sample_{idx}.png"
    img_path = os.path.join(image_dir, img_name)

    if os.path.exists(img_path):
        img = Image.open(img_path).convert("RGB")
        img = transform(img)
        images.append(img)

        one_hot_label = labels_to_one_hot(label_list, object_mapping)
        labels.append(one_hot_label)
    else:
        print(f"Image {img_name} not found in {image_dir}.")

# Convert lists to tensors
images = torch.stack(images)
# # images = images.clip(-1, 1)
labels = torch.stack(labels)

print(images)

# Instantiate the evaluator model
evaluator = evaluation_model()

# Evaluate the accuracy
accuracy = evaluator.eval(images.cuda(), labels.cuda())
print(f"Accuracy of the synthetic images: {accuracy * 100:.2f}%")
```

`evaluate.py` 設定隨機種子以確保結果的可重現性，然後從 .json 文件中讀取測試數據並定義一個 object_mapping，將物件名稱映射到對應的索引。隨後定義了一個函數將標籤轉換為 one-hot 編碼格式。

程式會從指定的目錄中載入生成的圖像，對其進行預處理（包括調整大小和正則化），並將其與相應的 one-hot 編碼標籤一起儲存到列表中。這些圖像和標籤最終會被轉換為張量格式。

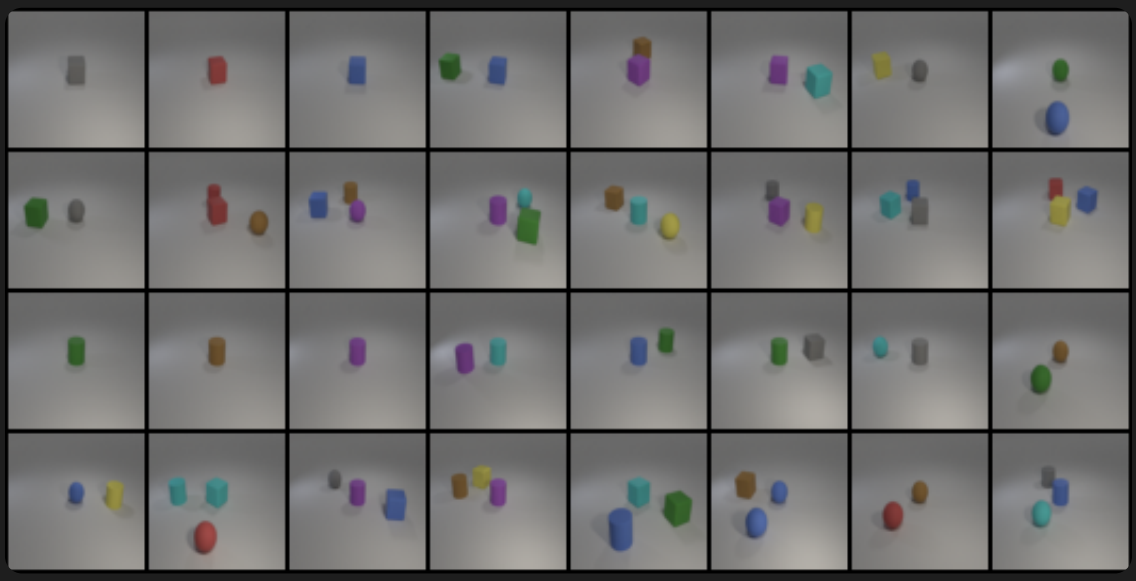
最後，程式使用已定義的評估模型(`evaluator.py`)來計算生成圖像的準確性，並將圖像去正則化後以網格形式保存到一個 PNG 文件中，並打印出準確度結果和保存訊息。

Result and discussion

Show your synthetic image grids and a denoising process image

synthetic image grids

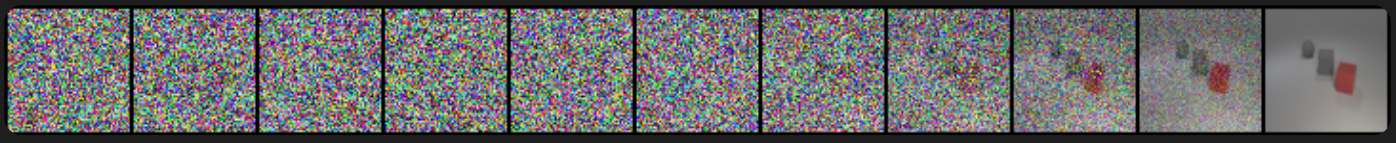
- test.json



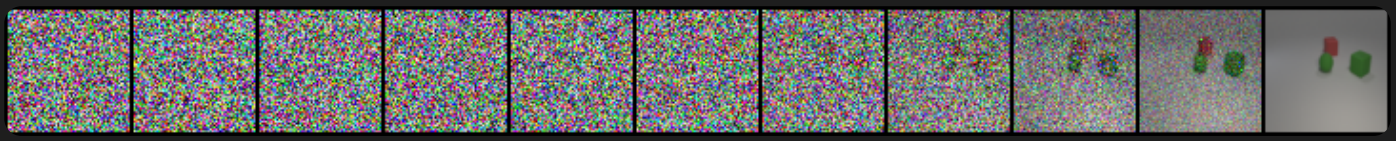
- new_test.json

denoising process image

- ["gray cube", "red cube", "gray sphere"]



- ["green cube", "red cube", "green sphere"]



Discussion of your extra implementations or experiments

在一開始寫這個作業時，原本是直接參考https://github.com/huggingface/diffusion-models-class 的模型部分去實作。殊不知，單純更改模型的維度去 fit iclevr dataset 是完全不夠的，這樣的話會生出一堆五彩繽紛的結果如下:

- gray cube

然而比起這個，更嚴重的問題是模型根本沒學到 condition，他不會去管 condition。即便我有做 label embedding。後來發現，單純就是模型的複雜度太低了，因此沒有辦法 fit 這個 task。畢竟我 reference 的 model 是拿來生成 MNIST 的資料的，完全比不上。

解決的方法就是非常直覺，爆改模型架構然後爆 train 個 100 epochs。

更改前:

```
block_out_channels=(32, 64, 64),
down_block_types=(
    "DownBlock2D",        # a regular ResNet downsampling block
    "AttnDownBlock2D",    # a ResNet downsampling block with spatial
self-attention
    "AttnDownBlock2D",
),
up_block_types=(
    "AttnUpBlock2D",
    "AttnUpBlock2D",      # a ResNet upsampling block with spatial
self-attention
    "UpBlock2D",          # a regular ResNet upsampling block
  ),
```

更改後:

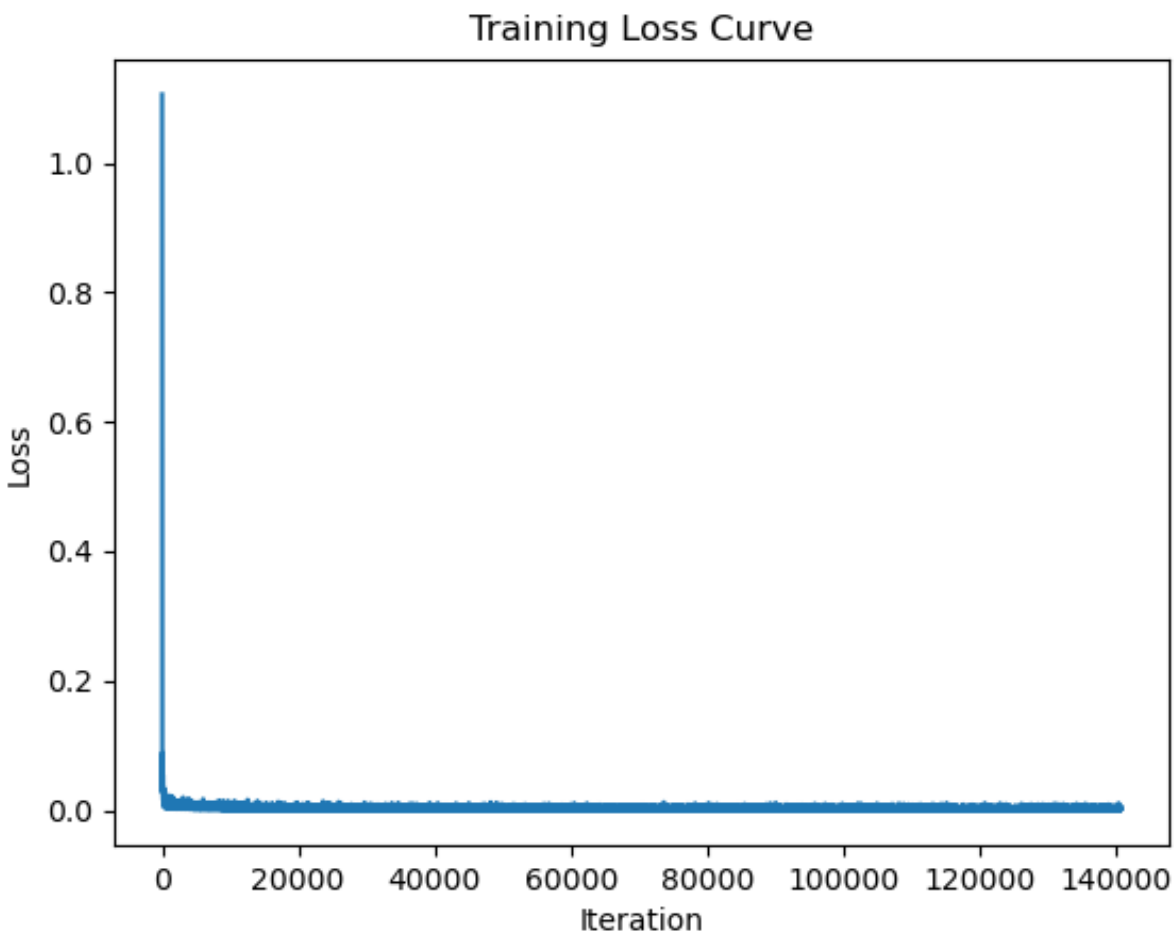
```
block_out_channels=(128, 128, 256, 256, 512, 512),  # 定義每個UNet區塊的輸出通道數
down_block_types=(
    "DownBlock2D",  # 定義下采樣區塊類型，這裡是普通的ResNet下采樣
    "DownBlock2D",
    "DownBlock2D",
    "DownBlock2D",
    "AttnDownBlock2D",  # 添加帶有空間自注意力的ResNet下采樣區塊
    "DownBlock2D",
),
up_block_types=(
    "UpBlock2D",  # 定義上采樣區塊類型，這裡是普通的ResNet上采樣
    "AttnUpBlock2D",  # 添加帶有空間自注意力的ResNet上采樣區塊
    "UpBlock2D",
    "UpBlock2D",
    "UpBlock2D",
    "UpBlock2D",
),
```

模型大小從 `33MB` 變成 `433MB` ，但結果也變得更加準確了:

- gray cube

除此之外，我也加上了 lr scheduler，因為如果讓 lr 一直維持 1e-4，每次到大概 100 epochs 的時候 loss 都會突然起飛到 1. 多，所以我使用了學習率調度器 (StepLR) 讓每 8 個 epoch 將學習率減少20%，loss 也變了穩定不少。

- loss curve



Experimental results part:

Classification accuracy on test.json and new test.json

Show your accuracy screenshots

- test.json (**acc: 81.94%**)

```
The test label .json is:  ./test.json
Accuracy of the synthetic images: 81.94%
Synthesized images saved as "synthesized_images_grid.png"
(DLP_lab) (base) hentci@gpu4:~/code/NYCU-DLP/lab6$
```

- new_test.json (**acc: 90.48%**)

```
The test label .json is:  ./new_test.json
Accuracy of the synthetic images: 90.48%
Synthesized images saved as "synthesized_images_grid.png"
(DLP_lab) (base) hentci@gpu4:~/code/NYCU-DLP/lab6$
```

The command for inference process for both testing data

我的環境為 `cuda version 12.5`, `python 3.9` ，除此之外有附 `requirement.txt` 在檔案中。

由於我的 sample 圖片部分與 evaluate 部分是分開的，助教需要先去跑 `sample.py` 生成圖片後再使用 `evaluate.py` 去評估生成的圖片QQ。

首先對於 `sample.py` :

需要先改動兩個 part:

1. 模型路徑

```
net = MultiLabelConditionedUnet(num_classes=24, class_emb_size=4).to(device)
net.load_state_dict(torch.load('/home/hentci/code/NYCU-DLP/lab6/DL_lab6_313551055_柯柏旭.pth'))
net.eval()
```

2. test json file 路徑

```
# 讀取 test.json
with open('./new_test.json', 'r') as f:
    test_data = json.load(f)
```

接著 run:

```
python3 sample.py
```

便會 sample 圖片到 './generated_images' 的資料夾。

接著需要跑 `evaluate.py` :

需要先改動兩個 part:

1. test json file 路徑

```
test_label_file = './test.json'
```

2. sample 圖片路徑

```
# Directory containing the generated images
image_dir = '/home/hentci/code/NYCU-DLP/lab6/generated_images'
```

接著 run:

```
python3 evaluate.py
```

便會跑出 accuracy 與生成 synthetic images 如下:

```
The test label .json is:  ./new_test.json
Accuracy of the synthetic images: 90.48%
Synthesized images saved as "synthesized_images_grid.png"
(DLP_lab) (base) hentci@gpu4:~/code/NYCU-DLP/lab6$
```

以上，感謝助教批改！