Lab7: Let’s Play DDPM

張千祐
Department of Computer Science
National Yang Ming Chiao Tung University
qianyou.cs11@nycu.edu.tw

1 Introduction

在這次 lab 中，目標是希望能夠實作 conditional Denoising Diffusion Probabilistic Models (DDPM)，用來作影像的生成，也就是說，在訓練時，除了給予影像以外，還會給予該影像的 label 作爲 condition input，在測試時，用 label 作爲條件輸入，產生對應這個 label 的影像。資料集是 iclevr，圖片包含各種顏色和形狀的物體。實作部分需要自己調整包含 noise schedule, UNet structure 等設計來增進生成圖片的準確率。

2 Implementation details

2.1 Choice of DDPM

DDPM 我選擇使用 diffusers 的 UNet2DModel，原因是在 hugging face 上有充足的教程資源以及相關代碼。

```python
from diffusers import UNet2DModel
```

Listing 1: DDPM

2.2 UNet architectures

UNet 部分我總共用了 3 個 block，其中兩個是 attention block，剩下是一般的 resnet block，每個 block 有 2 層 layer。加入 condition 的方式是我將 input channels 數量由原先的 3 改成 3+24，每一個多出來的 channel 都由 labels 的 one hot encoding value 組成。

```python
class ClassConditionedUnet(nn.Module):
  def __init__(self, args, num_classes=24):
    super().__init__()
    # Self.model is an unconditional UNet with extra input channels to
     accept the conditioning information (the class embedding)
    if args.pretrained:
      path = '{}/model.pt'.format(args.model_dir)
      self.model = torch.load(path)
    else:
      self.model = UNet2DModel(
        sample_size=64,           # the target image resolution
        in_channels=3 + num_classes, # Additional input channels for
    class cond.
        out_channels=3,           # the number of output channels
        layers_per_block=2,       # how many ResNet layers to use per
    UNet block
        block_out_channels=(128, 256, 256),
        down_block_types=(
          "DownBlock2D",        # a regular ResNet downsampling block
```

```python
          "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
        ),
      )

  # Our forward method now takes the class labels as an additional argument
  def forward(self, x, t, class_labels):
    # Shape of x:
    bs, ch, w, h = x.shape
    class_cond = class_labels.view(bs, class_labels.shape[1], 1, 1).expand(bs, class_labels.shape[1], w, h)
    # Net input is now x and class cond concatenated together along dimension 1
    net_input = torch.cat((x, class_cond), 1) # (bs, 7, 64, 64)
    # Feed this to the unet alongside the timestep and return the prediction
    return self.model(net_input, t).sample # (bs, 1, 64, 64)
```

Listing 2: UNet architectures

2.3 Noise schedule

Noise schedule 我同樣是採用 diffusers 的 DDPMScheduler，beta schedule 是採用預設的 linear 的方式。

```python
# Create a scheduler
noise_scheduler = DDPMScheduler(num_train_timesteps=args.timesteps)
noise_scheduler.set_timesteps(num_inference_steps=40)
```

Listing 3: Noise schedule

2.4 Loss functions

我的 model 是預測的是 noise，因此 loss function 是 predicted noise 跟 ground truth noise 之間的 MSELoss。

```python
def train(model, args):
  # Redefining the dataloader to set the batch size higher than the demo of 8
  train_dataset = iclevr_dataset(args, mode='train')
  test_dataset = iclevr_dataset(args, mode='test')
  train_dataloader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
  test_dataloader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

  # How many runs through the data should we do?
  n_epochs = args.epoch

  # Our loss finction
  loss_fn = nn.MSELoss()

  # The optimizer
  opt = torch.optim.AdamW(model.parameters(), lr=args.lr)

  # Keeping a record of the losses for later viewing
```

```python
losses = []

highest_acc = 0

# The training loop
for epoch in range(n_epochs):
  for x, y in tqdm(train_dataloader):

    # Get some data and prepare the corrupted version
    x = x.to(device) # Data on the GPU (mapped to (-1, 1))
    y = y.to(device)
    noise = torch.randn_like(x)
    timesteps = torch.randint(0, args.timesteps-1, (x.shape[0],)).long().to(device)
    noisy_x = noise_scheduler.add_noise(x, noise, timesteps)

    # Get the model prediction
    pred = model(noisy_x, timesteps, y) # Note that we pass in the labels y

    # Calculate the loss
    loss = loss_fn(pred, noise) # How close is the output to the noise

    # Backprop and update the params:
    opt.zero_grad()
    loss.backward()
    opt.step()

    # Store the loss for later
    losses.append(loss.item())

  # Print our the average of the last 100 loss values to get an idea of progress:
  avg_loss = sum(losses[-100:])/100
  writer.add_scalar('Train/Avg loss', avg_loss, epoch)
  with open('./{}/train_record.txt'.format(args.log_dir), 'a') as train_record:
    train_record.write(f'Finished epoch {epoch}. Average of the last 100 loss values: {avg_loss:05f}')
  print(f'Finished epoch {epoch}. Average of the last 100 loss values: {avg_loss:05f}')

  cur_acc = test(model, epoch, test_dataloader, args)
  if cur_acc > highest_acc:
    model.save(args)
    highest_acc = cur_acc
```

Listing 4: Loss functions

2.5 Hyperparameters

- Lr=5e-4
- Batch size=64
- Epoch = 100
- Timesteps = 1000
- Optimizer = AdamW

3 Results and discussion

3.1 Results

總共跑了 100epoch，以下是最高準確率的一次結果

	Test	NewTest
Acc(%)	68.06	78.57

Table 1: Highest Accracy

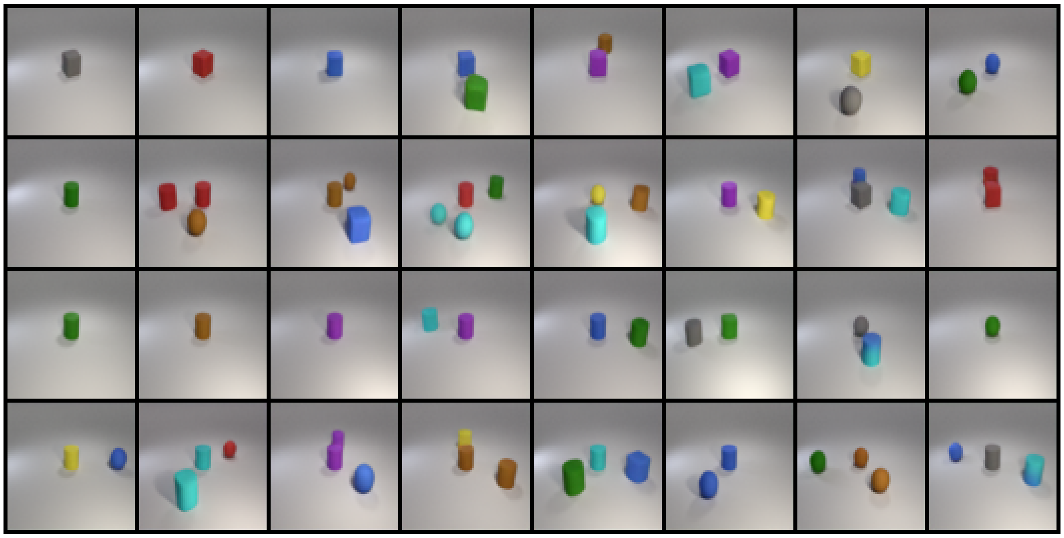

Figure 1: Generated images of test.json

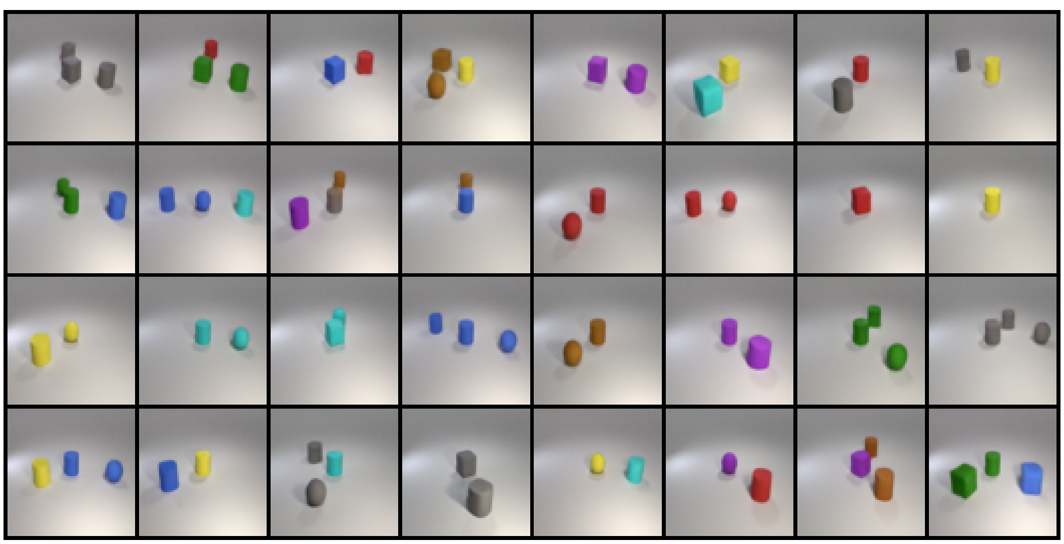

Figure 2: Generated images of new_test.json

3.2 Discussion

我發現 block_out_channels 影響我的 model 表現很大，我一開始是設 block_out_channels=(64, 128, 256)，跑了 50 個 epoch 完大概只有各不到 20% 的準

確率，而且生成圖片的顏色始終都像是帶有某種顏色的濾鏡一樣，不會是乾淨的白色背景，我改成 block_out_channels=(128, 256, 256) 後才變好。