DLP HW7

數據所 311554019 宋沛潔

• Introduction

在這個 lab 中，要用 conditional DDPM 的架構來根據多 label 條件生成圖像。因此在模型的設計中，要包含給定的條件去生成對應的圖像。那條件包含不同顏色不同形狀的物體，且一張照片可能包含多個物體，總共 label 有 24 個。最後再生成的圖像輸入到一個預訓練的 Restnet 來評估最後生成的結果。

Diffusion models 是一種生成模型，他主要是通過隨機過程中的 diffusion。在訓練時，模型從 input 中學習這種 diffusion 過程，然後在生成新樣本時，模型會去模擬這種 diffusion 過程，來生成出與 input 很像的照片。

• Implementation details

1. Describe how you implement your model

1.1 DDPM

採用 pixel domain diffusion models，diffusion 過程是直接應用於照片的 pixel。模型會在每個 t step 對 pixel 加 noise，在用模型訓練後，進行反向將 noise 移除，來根據給定的條件來生成出照片。

下面是對數據進行加 noise 並再去除，也就是 denoise。add noise 主要是將圖片加入一個隨機的 noise。模型的目標是讓真正的 noise 和模型預測的 noise 的差異越小越好。所以每個 t steps 模型都會更新參數。

```
if not args.test_only:
    for epoch in tqdm(range(args.epochs)):
        ddpm.train()
        for x, label in tqdm(train_dataloader):
            optimizer.zero_grad()
            x = x.to(args.device)
            label = label.to(args.device)
            current_step = torch.randint(0, args.t_steps, (x.shape[0],)).long()
            x_noised, noise = add_noise(x, current_step, args)
            pred = ddpm(
                x_noised.to(args.device), current_step.to(args.device), label.to(args.device).long()
            )
            loss = F.mse_loss(noise.to(args.device), pred)
            loss.backward()
            optimizer.step()
        train_loss.append(loss.item())
        print(f'[epoch: %02d] loss: %.5f \n' % (epoch, loss.item()))
```

至於在 test 時，會隨機生成 noise 與給定的條件一起送進模型訓練，模型會根據給定的 input 去還原圖片。

```python
def denoise_image(ddpm, test_data, args):
    ddpm.eval()
    pred_x = torch.randn(32, 3, 64, 64).to(args.device)
    for label in tqdm(test_data):
        image_x = denoise_with_steps(ddpm, pred_x, label, args)
    return image_x, label

def denoise_with_steps(ddpm, pred_x, label, args):
    current_x = pred_x.clone()
    for step in range(args.t_steps):
        step_time = torch.tensor([args.t_steps - step - 1] * 32).to(args.device)
        with torch.no_grad():
            pred_noise = ddpm(current_x, step_time, label.to(args.device).long())
            current_x = reverse(current_x, pred_noise, step_time, args)
    return current_x
```

```python
pred_x, test_label = denoise_image(ddpm, test_dataloader, args)
Accuracy = evaluation_model().eval(pred_x, test_label)
test_acc.append(Accuracy)
print(f'Test Accuracy: %.5f \n' % (Accuracy))
if Accuracy > best_acc:
    torch.save(ddpm.state_dict(), "Unet.pt")
    best_acc = Accuracy
    print(f'Save best test accuracy model! Best Accuracy: %.5f \n' % (Accuracy))
    image = make_grid(pred_x, nrow=8, normalize=True)
    save_image(image, "best_result.png")
```

1.2 UNet architectures

UNet 的設計如下。UNet 主架構分為三個部分，包含 2 層的 down、1 層的 double convolution、2 層的 up。Input 進來的維度是 3x64x64，出去也是 3x64x64。down 主要是先對輸入的特徵進行抓取，所以維度會從 256→512。再來是 1 層的 double convolution，維度到 512。up 的階段會將 x 進行特徵映射並將空間的資訊恢復到原本的大小 512→256。模型在 up 還加入了轉為向量的 label 與 time steps，被用來好好調整準備要恢復特徵，也就是把條件加入，幫助模型恢復到原本的照片。讓模型有層次學習，從低維度特徵到高維度特徵。

```python
class Unet(nn.Module):
    def __init__(self, in_c=3, out_c=3, classes=24):
        super(Unet, self).__init__()

        self.in_c = in_c
        self.out_c = out_c
        self.initial = ConvblockRes(in_c, 256, used_residual=True)

        self.down1 = DownConvUnet(256, 256)
        self.down2 = DownConvUnet(256, 512)

        self.h = nn.Sequential(nn.AvgPool2d(7), nn.GELU())

        self.layert1 = embedding(1, 512)
        self.layert2 = embedding(1, 256)
        self.layerc1 = embedding(classes, 512)
        self.layerc2 = embedding(classes, 256)

        self.conv = nn.Sequential(
            nn.ConvTranspose2d(512, 512, 8, 8),
            nn.GroupNorm(8, 512),
            nn.ReLU(),
        )

        self.up1 = UpConvUnet(1024, 256)
        self.up2 = UpConvUnet(512, 256)
        self.out = nn.Sequential(
            nn.Conv2d(512, 256, 3, 1, 1),
            nn.GroupNorm(8, 256),
            nn.ReLU(),
            nn.Conv2d(256, self.out_c, 3, 1, 1),
        )

    def forward(self, x, t, c):
        label1 = self.layerc1(c)
        time1 = self.layert1(t)
        label2 = self.layerc2(c)
        time2 = self.layert2(t)

        x = self.initial(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        hidden = self.h(down2)

        conv = self.conv(hidden)
        up1 = self.up1(label1 * conv + time1 , down2)
        up2 = self.up2(label2 * up1 + time2 , down1)
        out = self.out(torch.cat((up2, x), 1))
        return out
```

```python
class DownConvUnet(nn.Module):
    def __init__(self, in_c, out_c):
        super(DownConvUnet, self).__init__()
        self.upconv = nn.Sequential(ConvblockRes(in_c, out_c), nn.MaxPool2d(2))
    def forward(self, x):
        return self.upconv(x)


class UpConvUnet(nn.Module):
    def __init__(self, in_c, out_c):
        super(UpConvUnet, self).__init__()

        self.downconv = nn.Sequential(
            nn.ConvTranspose2d(in_c, out_c, 2, 2),
            ConvblockRes(out_c, out_c),
            ConvblockRes(out_c, out_c),
        )

    def forward(self, x, skip):
        x = torch.cat((x, skip), 1)
        x = self.downconv(x)
        return x
```

```python
class ConvblockRes(nn.Module):
    def __init__(self, in_c, out_c, used_residual=False):
        super().__init__()
        self.same_channels = False
        if in_c == out_c:
            self.same_channels = True
        self.used_residual = used_residual
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_c, out_c, 3, 1, 1),
            nn.BatchNorm2d(out_c),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_c, out_c, 3, 1, 1),
            nn.BatchNorm2d(out_c),
            nn.GELU(),
        )

    def forward(self, x):
        if self.used_residual:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            if self.same_channels:
                out = x + x2
            else:
                out = x1 + x2
            return out / 1.5
        else:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            return x2
```

ConvblockRes 包含了兩個 convolution，每個 convolution 後面都 BatchNorm2d 和 GELU，這樣可以加速模型收斂。residual connection 的設計是防止梯度消失。DownConvUnet 包含 ConvBlockRes 和 MaxPool2d。UpConvUnet 有 ConvTranspose2d 和兩個 ConvBlockRes。ConvTranspose2d 會將輸入進行 upsampling 。

1.3 Noise schedule

首先 add noise& reverse 都會去計算各 t steps 的 beta 值，讓他介於給定的 [start_beta, end_beta] 之間，然後算各 t steps 的 alpha 值以及 log alpha，其中 alpha 是 1 - beta。再來算 alphabar_t，他是所有 t steps 的 log alpha 的累加總和。並計算後面再加 noise 與 denoise 會用到用到值。add noise 是將原始照片 x_clear 添加 t steps 的 noise。reverse 是用來 denoise，把 xt 恢復到前一個 xt-1。

```python
def add_noise(x, current_step, args):
    beta_range = (args.end_beta - args.start_beta) * torch.arange(0, args.t_steps + 1, dtype=torch.float32)  / args.t_steps + args.start_beta
    alpha = 1 - beta_range
    log_alpha = torch.log(alpha)
    cumulated_alpha = torch.cumsum(log_alpha, dim=0).exp()
    alpha_bar = torch.sqrt(cumulated_alpha).to(args.device)
    alpha_bar_sqrt = torch.sqrt(1 - cumulated_alpha).to(args.device)

    noise = torch.randn_like(x).to(args.device)

    x_with_noise = (
        alpha_bar[current_step, None, None, None] * x
        + alpha_bar_sqrt[current_step, None, None, None] * noise
    )
    return x_with_noise, noise

def reverse(x_t, noise, current_step, args):
    beta_range = (args.end_beta - args.start_beta) * torch.arange(0, args.t_steps + 1).float() / args.t_steps + args.start_beta
    alpha = 1 - beta_range
    log_alpha = torch.log(alpha)
    cumulated_alpha = torch.cumsum(log_alpha, dim=0).exp()

    alpha_over_sqrt_one_minus_alphabar = ((1 - alpha) / torch.sqrt(1 - cumulated_alpha)).to(args.device)
    one_over_alpha = 1 / torch.sqrt(alpha).to(args.device)
    beta_sqrt = torch.sqrt(beta_range).to(args.device)

    epsilon_coefficient = alpha_over_sqrt_one_minus_alphabar[current_step, None, None, None]
    epsilon= torch.randn(x_t.shape, device=args.device)
    x_t_minus_one = (one_over_alpha[current_step, None, None, None]) * (x_t - epsilon_coefficient * noise) + (beta_sqrt[current_step, None, None, None]) * epsilon
    return x_t_minus_one
```

1.4 Loss functions

Loss function 使用的是 MSE。

2. **Specify the hyperparameters**

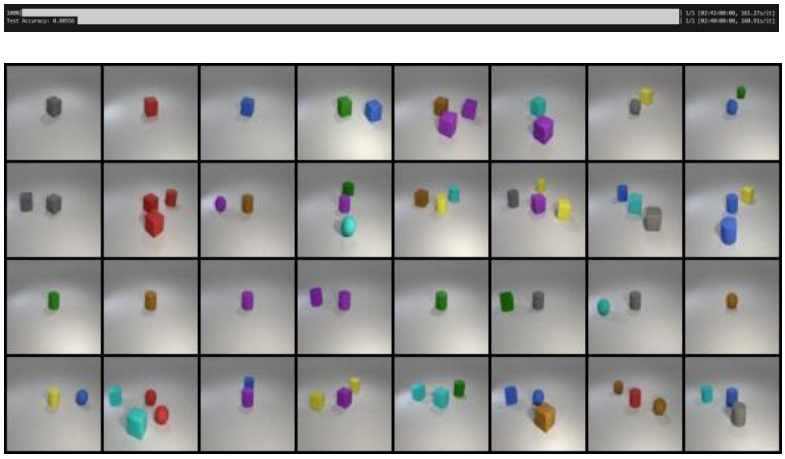
Hyperparameters	
Batch size	32
Learning rate	5e-4
T steps	1000
Epochs	550

其中，batch size 越大模型越快收斂，用更少的 epochs 更快達 80%以上。Learning rate 不能設太小要不然無法收斂。有加入 scheduler，但是效果都不好。T steps 增加對模型的準確度並沒有甚麼幫助。

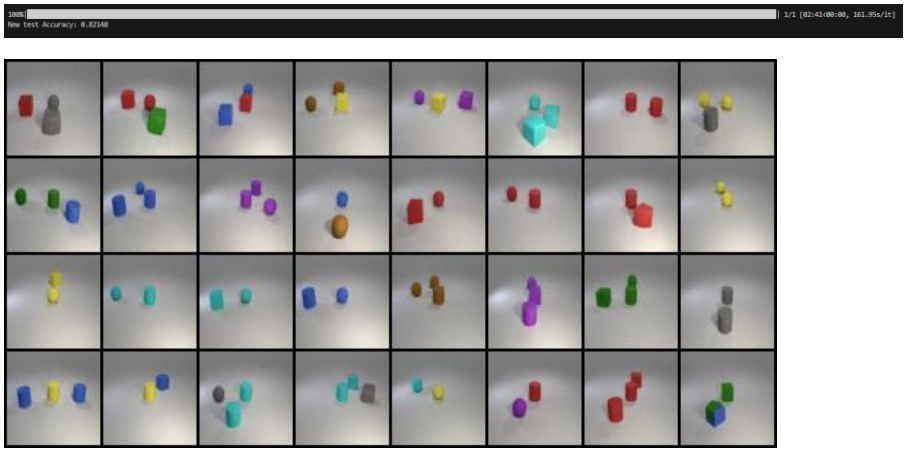
- ## Results and discussion

1. **Show your results based on the testing data:**

A. 在 test.json 的準確度為：0.80556



B. 在 new_test.json 的準確度為：0.8214



2. **Discussion:**

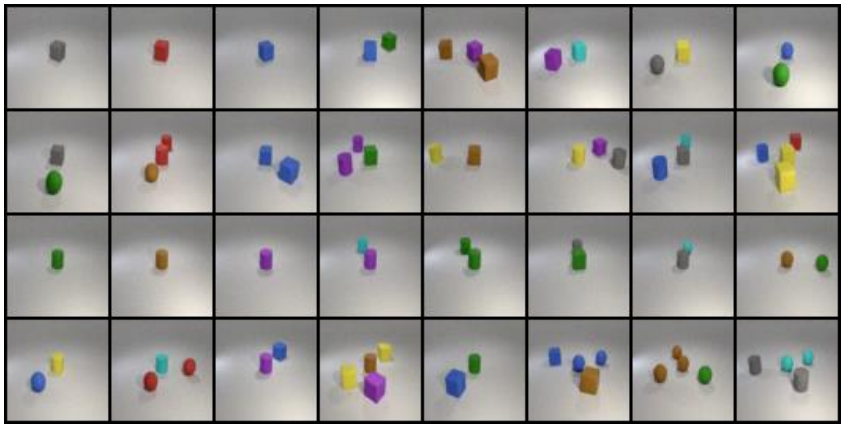
- 加深模型

  有嘗試將 up&down 都多加 2 層，並在 down 的抓取特徵過程的維度增加，雖然收斂效果變好，但是訓練時間變得很慢，因此只有

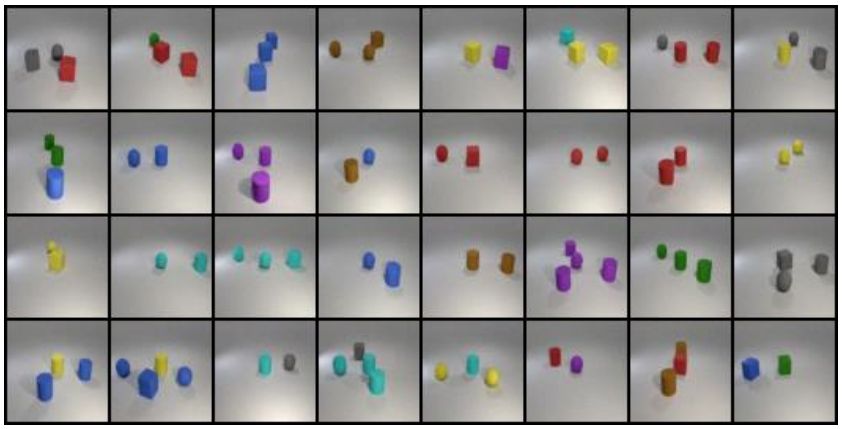
跑到 250 epochs，不過準確度就有到 75%左右。

- 去除 time embeddings
  時間資訊對於模型的訓練來說，可能沒有這麼重要，所以訓練出來的效果跟有加 time embeddings 相差不大，只是效果差了一點點。

  A. 在 test.json 的準確度為：0.79167



  B. 在 new_test.json 的準確度為：0.8036



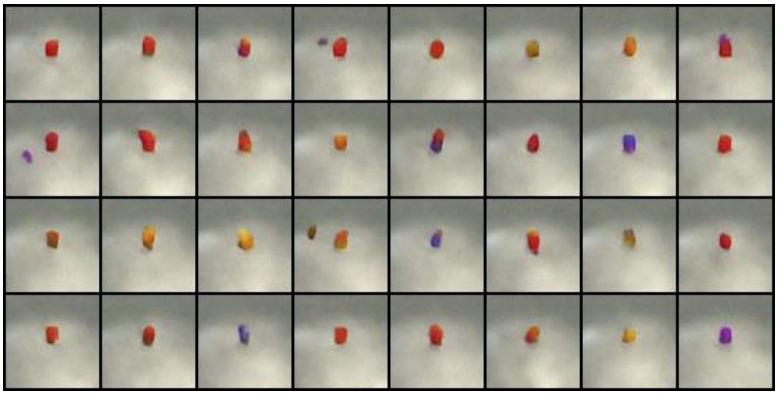
- 去除 Label condition embeddings
  label condition embeddings 拿掉是可以訓練出來算清晰的照片，但是因為沒有 label condition 的條件照片在生成的時候沒有根據，無法計算出 accuracy。
- Label condition 其他結合方式:
  把 label 及 time 在一開始相加並跟每一層 down 做結合。
  將 label 及 time 相加，然後加到每一層的輸入中。表示所有資訊在每一層影響效果都一樣。但是 label 及 time 在模型的不同層可能具有不同的重要性。所以用這個方法無法有效學到 label 及 time 各自帶有的資訊。

```python
def forward(self, x, t, c):
    # label1 = self.layerc1(c)
    # time1 = self.layert1(t)
    label2 = self.layerc2(c)
    time2 = self.layert2(t)
    f = label2 + time2

    x = self.initial(x)
    down1 = self.down1(x + f)
    down2 = self.down2(down1 + f)
    hidden = self.h(down2)

    conv = self.conv(hidden)
    up1 = self.up1(conv , down2)
    up2 = self.up2( up1 , down1)
    out = self.out(torch.cat((up2, x), 1))
    return out
```

A. 在 test.json 的準確度為：0.16667



B. 在 new_test.json 的準確度為：0.15476