DL_LAB6_REPORT

1.Introduction

在 CV 的 Condition-to-Image 任務中，Diffusion Probabilistic Models簡稱 DDPM是用來解這個任務的一種模型。其基本原理是通過一個 forward noising process，將真實圖像逐步添加random noise，直到變成純noise，然後再通過 reverse denoising process 根據學習到的參數，逐步去除噪聲，最終恢復出高品質的圖像。這兩階段的過程透過馬可夫鏈來逼近資料分佈，這樣才能夠生成很相似的output。相比於更老的GAN，DDPM 在訓練上比較穩定，且不易發生 mode collapse。此外，DDPM 在 image generation, image inpainting, super-resolution 等任務中表現也比較好，特別適合需要高質量且diverse的圖像輸出，雖然 sampling speed 相對較慢。在這次的lab中，我們給定條件（如 “red sphere”, “yellow cube”, “gray cylinder”），實做了完整的 DDPM pipeline，從 noise scheduler、模型架構設計到 sampling algorithm。我學到了如何調整 noise schedule、選擇 appropriate loss functions，收穫很多。

2. Implementation details

DDPM : ConditionalDDPM class 中我有使用 Diffusers 套件的 DDPMScheduler 負責添加 forwardnoise和backward denoise，用一個基於 Unet model當做 noise predictor；再加上 evaluation_model 做驗證；也有自己寫了 DataLoader 去讀取 iclevrDataset 的data。Optimizer 採用 Adam，並配合一個 cosine warmup learning-rate scheduler。在 train() 裡，對每個 batch 先用 scheduler add noise 、再讓 U-Net predict noise，計算 MSE loss，back propagate 後更新參數，同時調整 learning rate，並且在每個 epoch 結束時計算一次eval acc，根據表現儲存最佳模型，最後把訓練過程（loss、accuracy）和參數都寫入 record.txt 以便後續分析。程式碼如下圖所示。

```python
class ConditionalDDPM():
    def __init__(self, args):
        self.args = args
        self.device = args.device
        self.epoch = args.epoch
        self.lr = args.lr
        self.batch_size = args.batch_size
        self.num_train_timestamps = args.num_train_timestamps
        self.save_root = args.save_root
        self.label_embeding_size = args.label_embeding_size
        self.noise_scheduler = DDPMScheduler(num_train_timesteps=self.num_train_timestamps, beta_schedule="squaredcos_cap_v2")
        self.noise_predicter = Unet(labels_num=24, embedding_label_size=self.label_embeding_size).to(self.device)
        self.eval_model = evaluation_model()
        self.train_dataset = iclevrDataset(root="../dataset/iclevr", mode="train")
        self.train_loader = DataLoader(self.train_dataset, batch_size=self.batch_size, shuffle=True)
        self.optimizer = torch.optim.Adam(self.noise_predicter.parameters(), lr=self.lr)
        self.lr_scheduler = get_cosine_schedule_with_warmup(
            optimizer=self.optimizer,
            num_warmup_steps=args.lr_warmup_steps,
            num_training_steps=len(self.train_loader) * self.epoch,
            num_cycles=50
        )

```

```python
    def train(self):
        # print(f"train dataset length: {len(self.train_loader)}")
        loss_criterion = nn.MSELoss()
        # training
        train_loss = []
        test_acc = []
        bestacc = 0
        for epoch in range(1, self.epoch+1):
            epoch_loss = 0
            for x, y in tqdm(self.train_loader):
                x = x.to(self.device)
                y = y.to(self.device)
                noise = torch.randn_like(x)
                timestamp = torch.randint(0, self.num_train_timestamps - 1, (x.shape[0], ), device=self.device).long()
                noise_x = self.noise_scheduler.add_noise(x, noise, timestamp)
                perd_noise = self.noise_predicter(noise_x, timestamp, y)
                loss = loss_criterion(perd_noise, noise)
                loss.backward()
                nn.utils.clip_grad_value_(self.noise_predicter.parameters(), 1.0)
                self.optimizer.step()
                self.lr_scheduler.step()
                self.optimizer.zero_grad()
                self.lr = self.lr_scheduler.get_last_lr()[0]
                epoch_loss += loss.item()
            epoch_loss /= len(self.train_loader)
            train_loss.append(epoch_loss)
            print('Epoch ' + str(epoch) + ' Loss: ' + str(epoch_loss))
            # acc = self.evaluate(epoch, testing_dataset="test")
            acc = self.evaluate( testing_dataset="test")
            test_acc.append(acc)
            print('Epoch ' + str(epoch) + ' Accuracy: ' + str(acc))
            if acc > bestacc or acc > 0.8 :
                if acc > bestacc :
                    bestacc = acc
                self.save(os.path.join(self.args.ckpt_root, f"epoch={epoch}.ckpt"), epoch)
        f = open(os.path.join(self.args.ckpt_root, f"record.txt"), 'w')
        print('Arguments:', file=f)
        for arg, value in vars(self.args).items():
            print(f"{arg}: {value}", file=f)
        print(file=f)

        print('Loss', file=f)
        for i in range(self.epoch):
            print(train_loss[i], ',', end=' ', file=f)
            if i % 10 == 9:
                print(file=f)
        print(file=f)

        print('Accuracy', file=f)
        for i in range(self.epoch):
            print(test_acc[i], ',', end=' ', file=f)
            if i % 10 == 9:
                print(file=f)
        print(file=f)

        f.close()
```

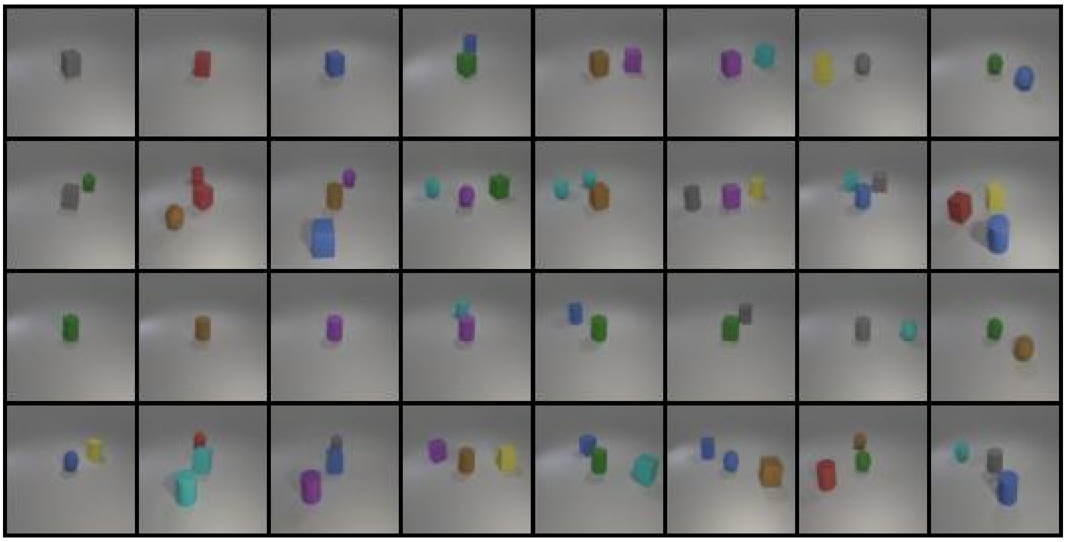
noise predictor 的部分我寫在unet.py中，如下圖。
程式碼中有使用diffuser的UNet2DModel。Unet的input 有RGB還有conditiion，所以對condition進行shapeexpand 來allign input image shape，之後再將他們concatenate，因此input channel會是3(RGB)+24(Condition)。timeembedding 則直接將 timestamp 輸入到Unet2DModel，model 便

會自動處理。我實作noise_predicter的程式碼如下頁 圖所示。

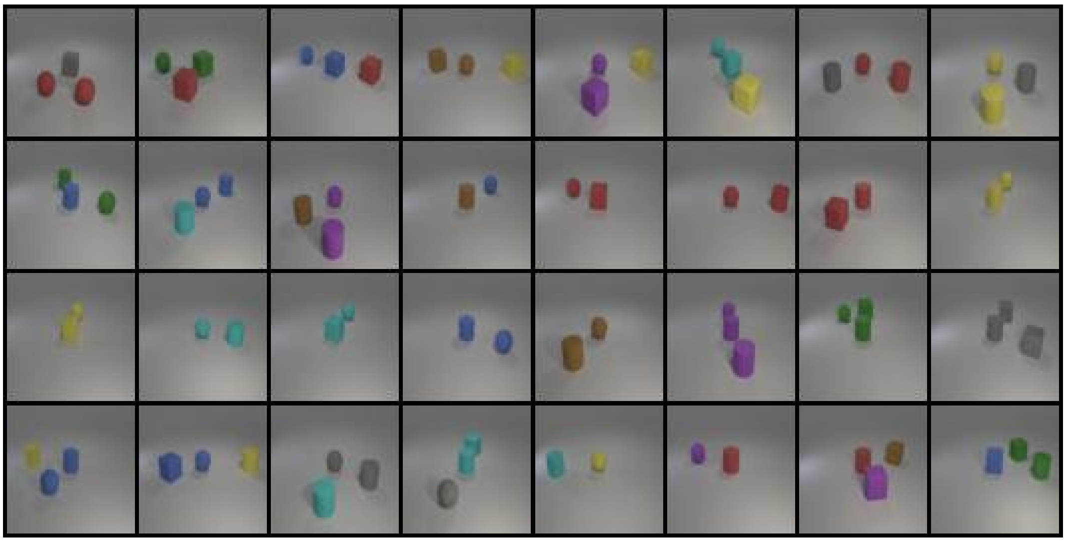
```python
class Unet(nn.Module):
    def __init__(self, labels_num = 24, embedding_label_size=4) -> None:
        super().__init__()
        self.label_embedding = nn.Embedding(labels_num, embedding_label_size)
        self.model = UNet2DModel(
            sample_size = 64,
            # in_channels = 3 + labels_num * embedding_label_size,
            in_channels = 3 + labels_num,
            out_channels = 3,
            time_embedding_type = "positional",
            layers_per_block = 2,
            block_out_channels = (128, 128, 256, 256, 512, 512),  # number of output channels for each UNet block
            down_block_types = (
                "DownBlock2D",  # a regular ResNet downsampling block
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
                "DownBlock2D",
            ),
            up_block_types = (
                "UpBlock2D",  # a regular ResNet upsampling block
                "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )
    def forward(self, x, t, label):
        bs, c, w, h = x.shape
        embeded_label = label.view(bs, label.shape[1], 1, 1).expand(bs, label.shape[1], w, h)
        unet_input = torch.cat((x, embeded_label), 1)
        unet_output = self.model(unet_input, t).sample
        return unet_output
```

3. Results and discussion

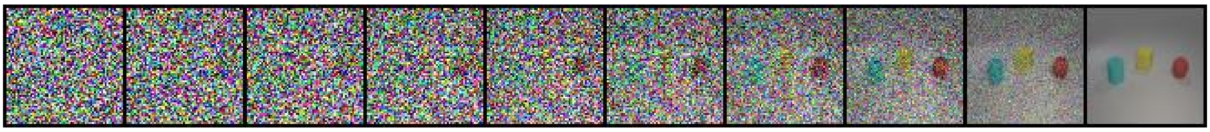
- Show your synthetic image grids (total 16%: 8% * 2 testing data)
    - test.json



- new_test.json

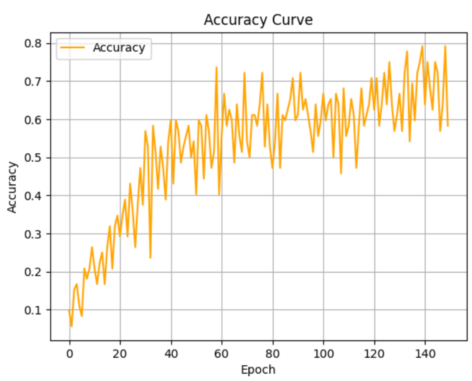
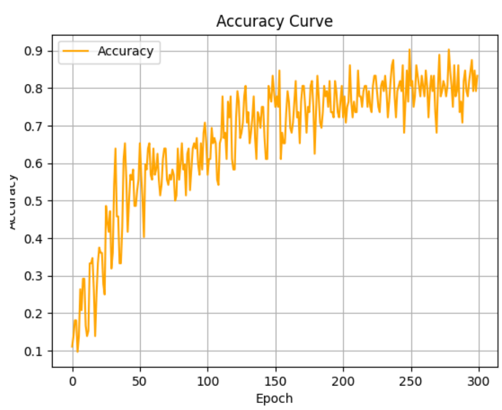


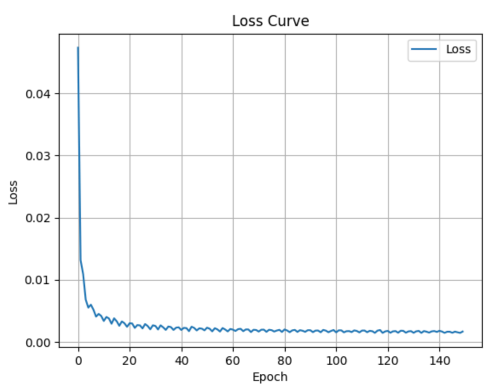
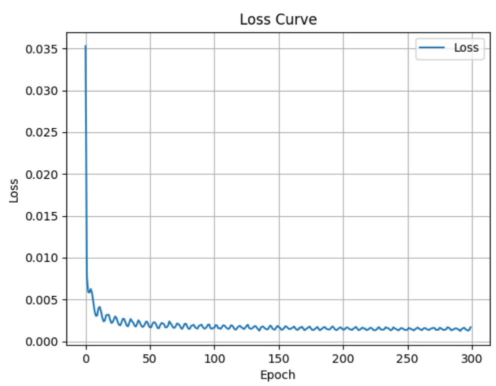
- denoising process image with the label set ["red sphere", "cyan cylinder", "cyan cube"]



- Discussion of your extra implementations or experiments (10%)

有測試了兩組hyperparameters，其結果如下面四張圖可以看到,
基本上epoch 150 已經能夠達到還不錯的accuracy ,只是再繼續train到300其實大約就能從Acc 0.7左右提升至Acc 0.8 左右，可以觀察到DDPM其實相較於GAN很好train，不會因為hyper parameters 的一點點差異就導致model collapse 。

lr	1e-4	2e-4
epoch	150	300
acc curve		

loss curve		

4.Experimental results

test.json acc : 0.819

new_test.json acc : 0.905

```
(base) winston@gpu9:/project2/winston/lab/lab6/DDPM$ python main.py --test True --ckpt_path ./epoch=220.ckpt --ckpt_root ./save_model/2
 In testing mode : True
 testing_dataset :  test
 1000it [00:37, 26.99it/s]
 DDPM test.json Accuracy :  0.819
 testing_dataset :  new_test
 1000it [00:36, 27.36it/s]
 DDPM new_test.json Accuracy :  0.905
 1000it [00:10, 94.39it/s]
```