DLP
LAB 6

謝侑哲

112550069

Github:

https://github.com/youzhe0305/NYCU-DLP

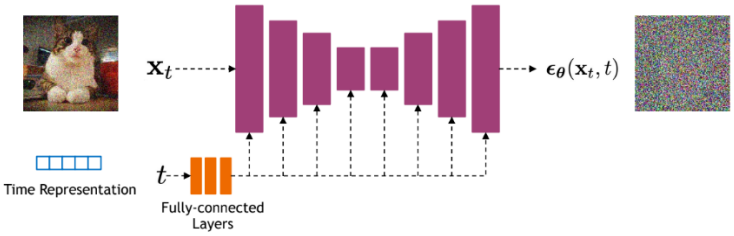
1. Introduction

本次 lab 為，使用 DDPM 根據 condition 的幾何體名字，生成幾何體的圖片，希望可以在 pre-train 的 Discriminator 中，取得盡可能高的準度，也就是生產出來的幾何體更貼近現實中的幾何體。

2. Implementation Details

A. Describe how you implement your model, including your choice of DDPM, noise schedule

DDPM 有很多不同的類型可以選擇，最基本的就是單純的撒 noise 在上面，再由 noise predictor 預測出 noise 之後，對於 noisy image 做 denoise。不過我的實作上採用更加主流的方式，把 UNet 掛載在 DDPM 上，如下圖:



透過掛載 UNet 的方式，可以讓模型做 noise predict 時，同時使用 high r esolution 的資訊以及提取精煉過的資訊。讓他能更好的預測 noise。

除了單純用 DDPM 之外，我也另外建立的一個 label encoder，用來把物件類別的資訊 encode 成更有意義的 embedding，再餵給 DDPM，讓他能對 condition 做更好的理解。

以下是我的實作，採用 hugging face 的 diffuser 的 UNet2DModel，建構也是參考他的 tutorial: https://github.com/huggingface/diffusion-models-class/blob/main/unit1/02_diffusion_models_from_scratch.ipynb

，可以透過給予參數簡單建立模型。至於為甚麼不採用 hugging face 的 UNet2DConditionModel 來處理這個 condition DDPM 的任務，是因為這個任務的 condition 只有 class，用簡單的 UNet2DModel 就可以處理，相對來說，UNet2DConditionModel 更適合用來處理有複雜 condition，也必須傳入 condition 的 hidden representation。這裡使用 UNet2DModel 就可以直接傳入在 class 參數中，不過就要另外寫簡單的 encoder 而不能用 ConditionModel 才有的 class projection。

Encoder 的建立上，我是簡單使用兩個 Linear 層做 encode，讓他能用二次曲線的相對平滑的方式做 encode。

DDPM 的層數設計則是建立了 6 層，前面先使用普通的 DownBlock，後面放一個 AttentionBlock 增加模型的 capacity。至於不多放 AttentionBLock 的原因是放太多的話，Attention 的計算量很大，會導致訓練速度非常慢，並且當前的 capacity 已經夠用，故只放一層。另外，其他大部分設定都依照預設的，向是一層由兩個 ResBlock 組成等。

Forward 則是先 encode label，在把圖片、timestep、embedding 都傳入 DDPM 中，讓他能依據這些做適合的 noise predict。

```
class cDDPM(nn.Module):
    def __init__(self, n_object_class):
        super().__init__()
        self.label_enocder = nn.Sequential( # 因為非condition的UNet2DModel不能內部設定projection，所以另外加一個encoder
            nn.Linear(n_object_class, 64),
            nn.Linear(64,512)
        )
        self.ddpm = diffusers.UNet2DModel( # 只處理time step跟class不需要用到ConditionModel，有更高級的cnodition才要。
            sample_size=(64,64),           # the target image resolution
            in_channels=3,            # the number of input channels, 3 for RGB images
            out_channels=3,           # the number of output channels
            class_embed_type='identity',
            layers_per_block=2,       # how many ResNet layers to use per UNet block
            block_out_channels=(128, 128, 256, 256, 512, 512), # Roughly matching our basic unet example
            down_block_types=(
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
                "DownBlock2D",
            ),
```

```
            up_block_types=(
                "UpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )

    def forward(self, img, t, label):
        encoded_label = self.label_enocder(label)
        return self.ddpm(sample=img, timestep=t, class_labels=encoded_label).sample
```

關於 noise scheduler 的選擇，我使用的是 squaredcos_cap_v2，這個 noise scheduler 使用 cos 的平方，避免在一開始的 noise 過大，導致生成的效果不好。相較於 linear 或純 cos，理論上會有更加平滑穩定的生成結果。實作是直接引用 hugging face 的 DDPMscheduler，如下。

```
self.noise_scheduler = DDPMScheduler( # 讓noisr可以直接加到
    num_train_timesteps=args.time_step,
    beta_start=0.0001, # beta代表t到t+1時，noise要加的量
    beta_end=0.02,
    beta_schedule='squaredcos_cap_v2'
)
```

B. Training & Testing Process

因為沒有 demo，所以這裡額外討論 training 跟 testing 的實作。

Training 的流程是，首先從 gaussian distribution 中 sample 出一張根源圖一樣大小的 noise。接著隨機抽一個 timestep t，因為 DDPM 理論中的，每一步撒 noise 的比例 beta 可以透過代換及常態分佈 variance 運算的特性，導出某個 t 對應的係數 alpha_t，因此只要隨機抽一個 t，並灑上對應的 noise 比例就行。而部分就由 DDPMscheduler 來實作。在圖上加躁之後，連同 label, t 丟給 model 做預測，嘗試預測出原來的 noise，並且用 MSE 計算 loss，至於為甚麼是 MSE，是因為 DDPM 本質上預測出來的是 noise distribution(常態分布)的 mean，也就是最高機率的那個 noise。所以用 MSE 會是個合理的 loss。最後做更新就完成一個 epoch 的 training，剩下就是迴圈而已。

```python
def train_one_epoch(self, ith_epoch):
    iterations = 0
    for idx, (img, label) in enumerate(progress_bar):
        img = img.to(device)
        label = label.to(device)

        noise = torch.randn(img.shape).to(device)
        t = torch.randint(low=0, high=self.args.time_step, size=(img.shape[0],)).long().to(device) # random time_
        noisy_img = self.noise_scheduler.add_noise(img, noise, t)
        predict_noise = self.model(noisy_img, t, label)

        loss = self.criterion(predict_noise, noise) # 雖然加的不是原本的noise，而是sqrt(1-alpha_t)*noise，但反正只差

        total_loss += loss
        iterations += 1
        loss.backward()
        self.optimizer_step()

        progress_bar.set_description(f'Training, Epoch {ith_epoch}:', refresh=False)
        progress_bar.set_postfix({
            'lr': f'{self.scheduler.get_last_lr()}',
            'averge loss':f'{round(total_loss.item() / iterations, 8)}'
        }, refresh=False)
        progress_bar.refresh()

    self.scheduler.step()
```

關於 testing 的部分，則是由大到小取得 t，並預測每一步應該要 denoise 的量，因為數學上有特殊的係數，所以同樣使用 DDPMscheduler 來輔助計算，使用 step 函數做圖片的更新。

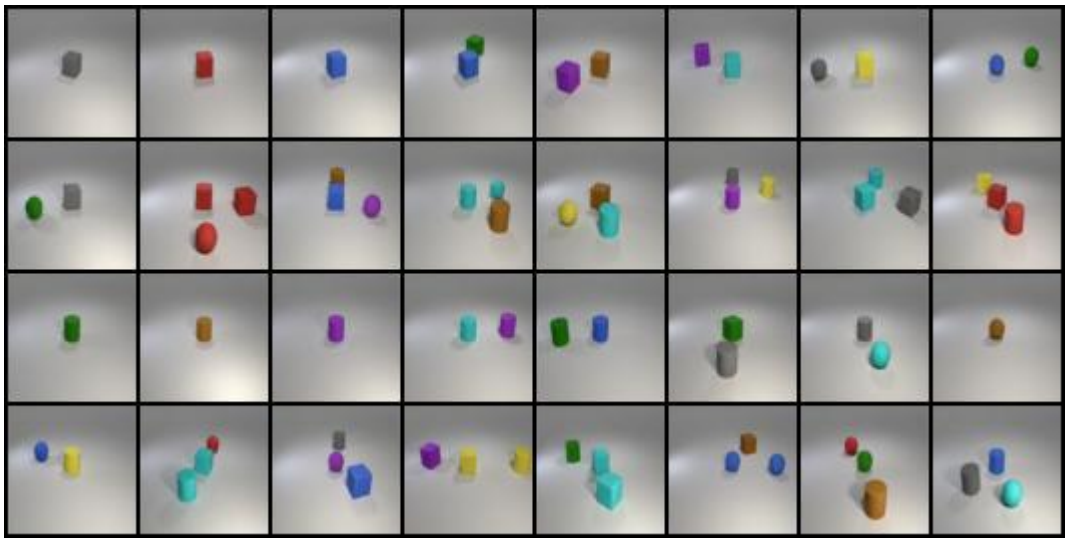
```python
for idx, label in enumerate(progress_bar):
    process_img = []
    label = label.to(device)
    noise = torch.randn(1, 3, 64, 64).to(device)
    img = noise
    for i, t in enumerate(self.noise_scheduler.timesteps):
        # print(t)
        predict_noise = self.model(img, t, label)
        img = self.noise_scheduler.step(predict_noise, t, img).prev_sample # 生成x_{t-1}
        if i % 100 == 0:
            process_img.append(img.squeeze(0))

    acc = self.evaluator.eval(img, label)
    print(acc)
    total_acc += acc
    process_img.append(img.squeeze(0))
    process_img = torch.stack(process_img) # (n,c,h,w)
    grid = make_grid(process_img, normalize=True)
    save_image(grid, f'output/processing_{idx}.jpg')
print('averge accuracy: ',total_acc / len(self.dataset))
```

3. Results and Discussion

A. Show your synthetic image grids and a denoising process image

Test Image:



New Test Image:

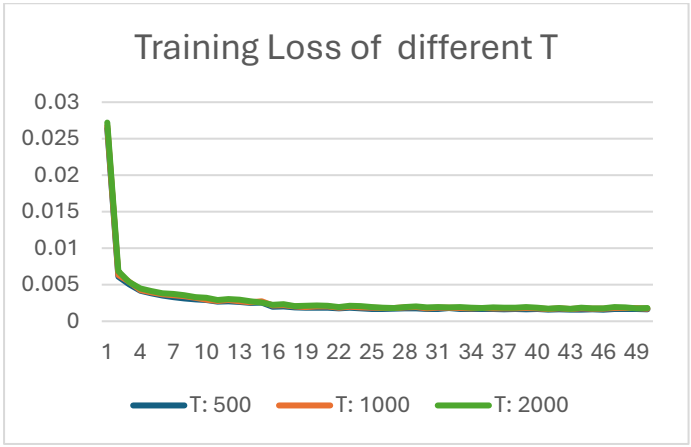
Denoising process – ["red sphere", "cyan cylinder", "cyan cube"] (因為有跟 Noise 做 normalization，所以看起來灰灰的):

B. Discussion of your extra implementations or experiments

除了上述實驗外，我也額外實驗了不同的 timestep 會對於準度造成何種結果，分別對於 timestep 500, 1000, 2000 訓練 50 個 epoch，並且在 test 資料上做 inference，查看他的平均 accuracy。使用的指令如下:

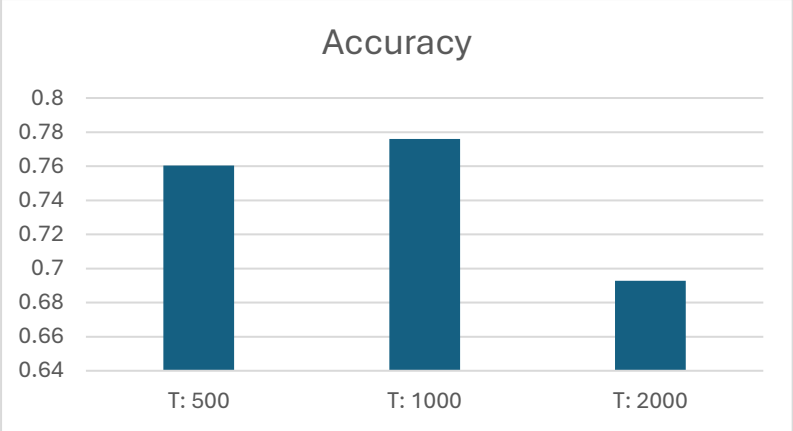
python3 train.py --batch_size 32 --num_epoch 50 --lr 0.0001 --data_root ../iclevr --save_root ../model_weight --time_step $

Training Loss:



從圖中可以看到，對於不同的 timestep，他們 training loss 的走向幾乎一樣，甚至可以說三條線是幾乎疊合在一起，並且都進入了收斂的狀態。因此對於依賴原圖撒上 noise 的 denoise 能力，三者可以說是差不多。不過，看到產出圖片的 accuracy 就有明顯的差別了。

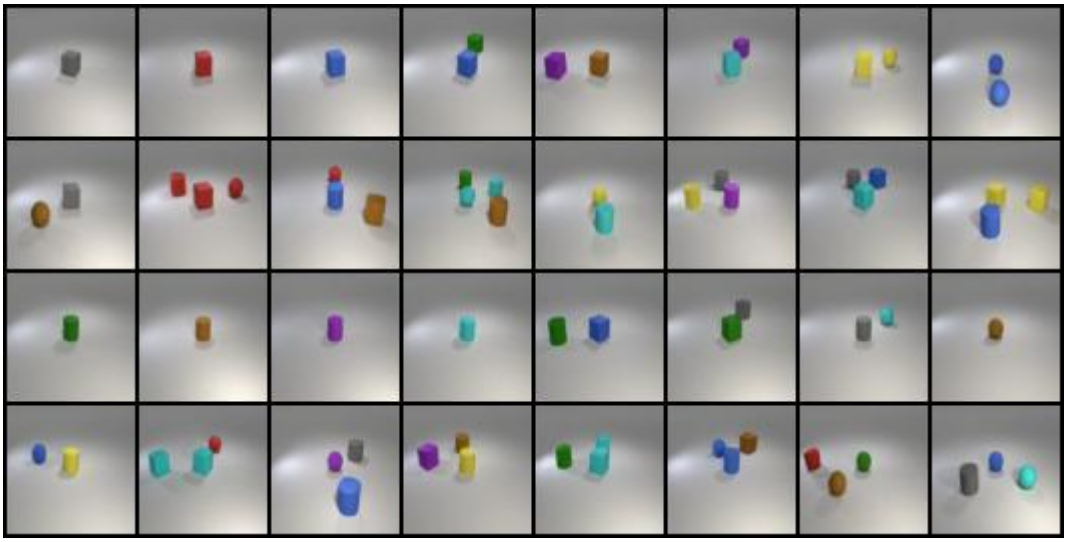
Accuracy:



可以看到說 Timestep 1000 的 accuracy 最高，次之是 500，最低的是 2000。先從 2000 的看起，可以看出當 timestep 過高時，會導致訓練出來的成效不好。我認為有兩種可能原因。

第一，因為 DDPM 的訓練方式是隨機抽一個 t，再去做 noise 預測，但是 timestep 太大的時候，每個 t 被分配到的訓練次數就會減少，在相同 epoch 數量的情況下，訓練出來的成果就會比較不好。不過 3 者的 training loss 較低，所以這個可能性還有待進一步的訓練確認。

第二，因為 DDPM 在 timestep 大的情況下，可能會對於 noise 有 overfitting 的狀況，也就是過度的處理像 noise 這類的 high frequency 的細節，結果導致生出來的圖片細節不足，讓生成的品質下降。下圖是 2000 個 timestep 生成出來的圖。跟最佳的圖比對，可以看出有些該是方柱的地方，卻變得有點像是圓柱，符合上面的敘述。不過也可以看出他有很多顏色的生成錯誤，或者是少生成之類的，所以確切原因也還有待進一步的實驗。

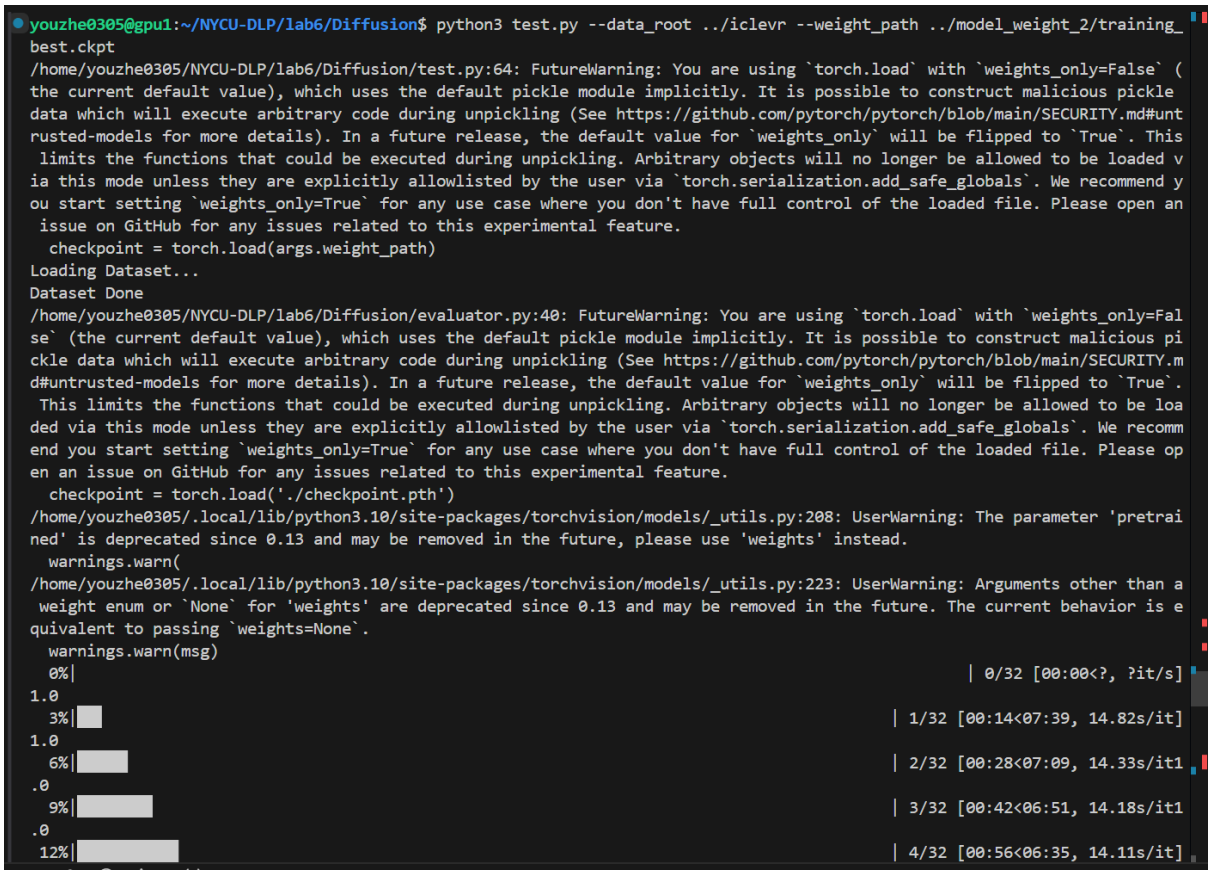


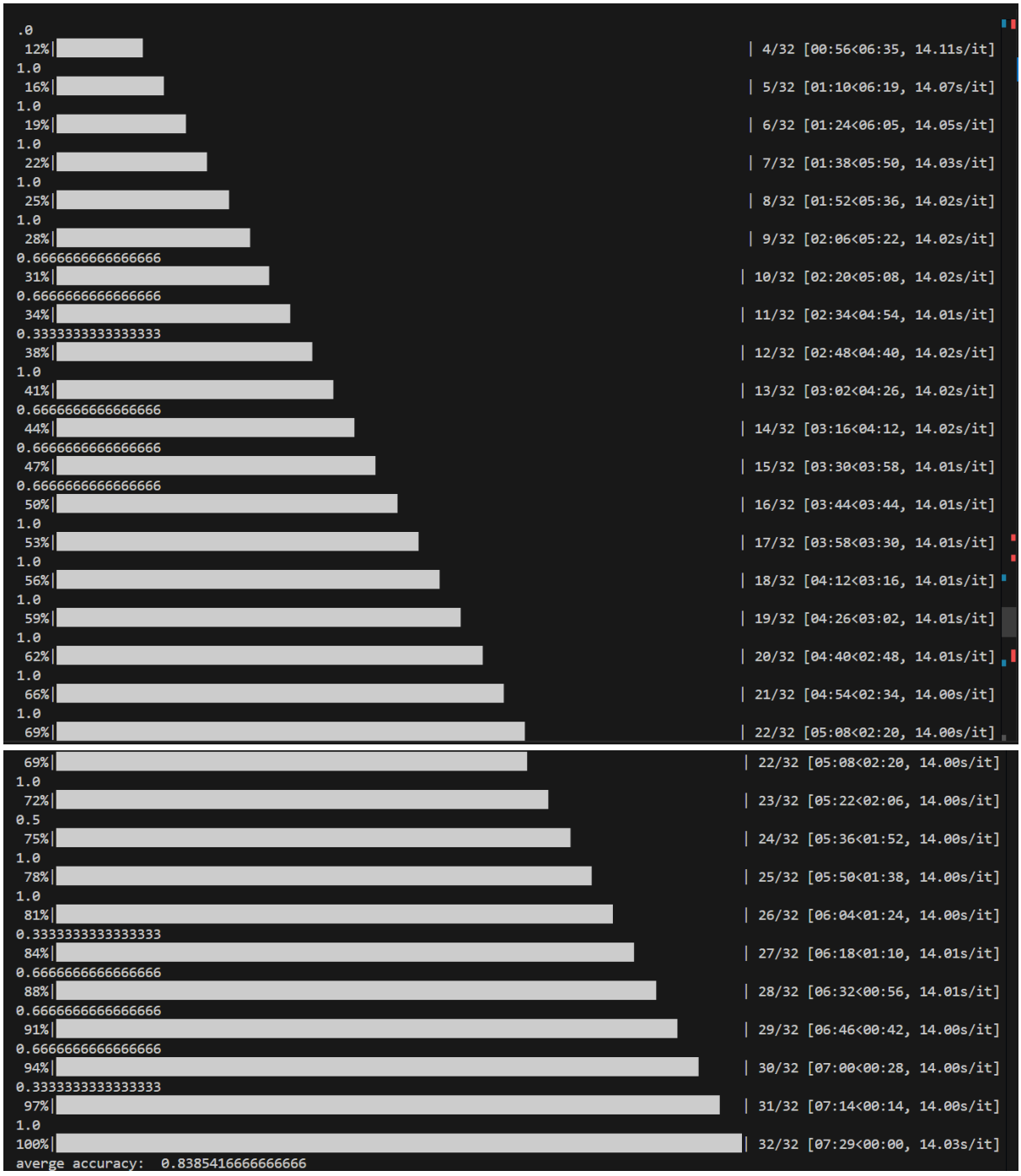
關於 timestep 比較少的 500 步，其實他跟標準的 1000 步並沒有差到太多，accuracy 差不到 2%，幾乎可以視為是誤差了。不過要分析原因的話，有可能是因為 timestep 太小時，會導致 denoise 跨度較大比較難處理細節的東西。而剛好這次做的都是簡單的幾何體，對於細節的要求較少，所以減少 timestep 對於 accuracy 的影響比較少。

4. Experiment Results

A. Classification accuracy on test.json and new test.json

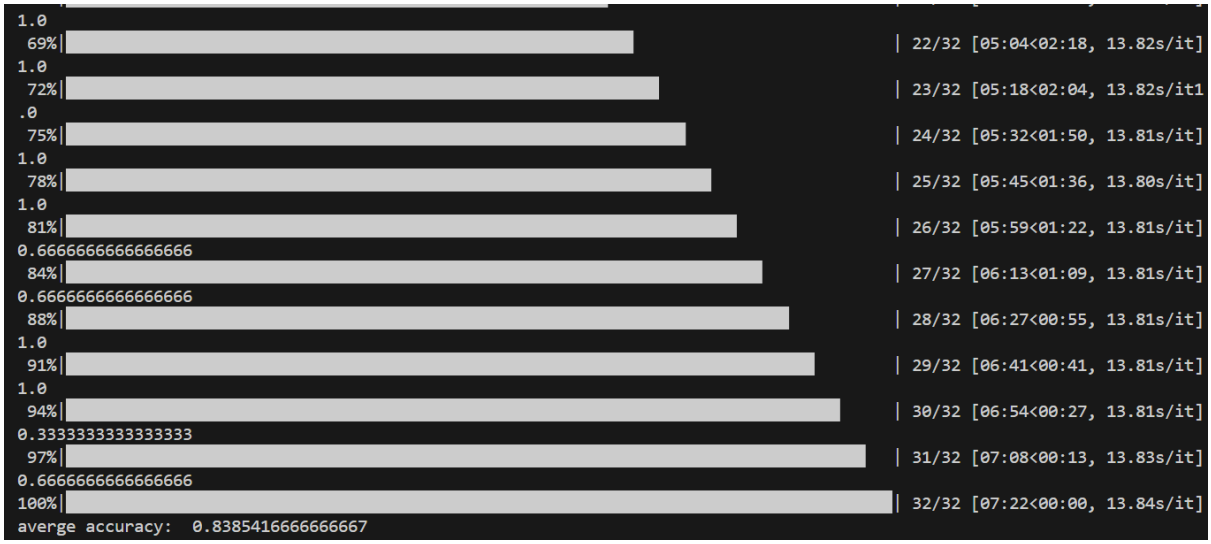
以下為 test.json object 的結果，得到的平均準度為 0.8385





以下為 test_new.json object 的結果，得到的平均準度為 0.8385 (雖然平均準度剛好跟 test 一樣，但就純粹巧合，他的過程中每個的準度是不同的，沒有丟同一份資料)

```
youzhe0305@gpu1:~/NYCU-DLP/lab6/Diffusion$ python3 test.py --data_root ../iclevr --weight_path ../model_weight_2/training_
best.ckpt --test_type new_test
/home/youzhe0305/NYCU-DLP/lab6/Diffusion/test.py:64: FutureWarning: You are using `torch.load` with `weights_only=False` (
the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle
data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#unt
rusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This
 limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded v
ia this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend y
ou start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an
 issue on GitHub for any issues related to this experimental feature.
  checkpoint = torch.load(args.weight_path)
Loading Dataset...
Dataset Done
/home/youzhe0305/NYCU-DLP/lab6/Diffusion/evaluator.py:40: FutureWarning: You are using `torch.load` with `weights_only=Fal
se` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pi
ckle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.m
d#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`.
 This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loa
ded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recomm
end you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please op
en an issue on GitHub for any issues related to this experimental feature.
  checkpoint = torch.load('./checkpoint.pth')
/home/youzhe0305/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrai
ned' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/youzhe0305/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a
 weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is e
quivalent to passing `weights=None`.
  warnings.warn(msg)
  0%|                                                                                                | 0/32 [00:00<?, ?it/s]
1.0
  3%|▉                                                                                     | 1/32 [00:14<07:30, 14.52s/it]
0.6666666666666666
  6%|█▊                                                                                    | 2/32 [00:28<07:02, 14.07s/it]
0.6666666666666666
  9%|██▊                                                                                   | 3/32 [00:42<06:45, 13.97s/it]
1.0
 12%|███▋                                                                                  | 4/32 [00:55<06:29, 13.90s/it]
0.6666666666666666
 16%|████▋                                                                                 | 5/32 [01:09<06:13, 13.85s/it0
.6666666666666666
 19%|█████▌                                                                                | 6/32 [01:23<05:59, 13.82s/it0
.6666666666666666
 22%|██████▌                                                                               | 7/32 [01:37<05:45, 13.80s/it]
0.6666666666666666
 25%|███████▌                                                                              | 8/32 [01:50<05:30, 13.79s/it]
0.6666666666666666
 28%|████████▍                                                                             | 9/32 [02:04<05:17, 13.81s/it]
0.3333333333333333
 31%|█████████▍                                                                           | 10/32 [02:18<05:03, 13.81s/it]
1.0
 34%|██████████▎                                                                          | 11/32 [02:32<04:50, 13.82s/it]
1.0
 38%|███████████▎                                                                         | 12/32 [02:46<04:36, 13.81s/it]
1.0
 41%|████████████▎                                                                        | 13/32 [03:00<04:22, 13.83s/it]
1.0
 44%|█████████████▏                                                                       | 14/32 [03:13<04:08, 13.82s/it]
1.0
 47%|██████████████▏                                                                      | 15/32 [03:27<03:55, 13.84s/it]
1.0
 50%|███████████████                                                                      | 16/32 [03:41<03:41, 13.83s/it]
0.5
 53%|████████████████                                                                     | 17/32 [03:55<03:27, 13.83s/it]
1.0
 56%|█████████████████                                                                    | 18/32 [04:09<03:13, 13.82s/it]
1.0
 59%|█████████████████▉                                                                   | 19/32 [04:23<02:59, 13.83s/it]
1.0
 62%|██████████████████▉                                                                  | 20/32 [04:36<02:45, 13.81s/it]
1.0
 66%|███████████████████▊                                                                 | 21/32 [04:50<02:32, 13.83s/it]
1.0
```



A. The command for inference process for both testing data

使用 inference 的 command 如下:

1. 把 model weight 存到 model_weight_best 的 folder 中
2. 首先移動到 Diffusion 資料夾底下:
   cd Diffusion
3. 關於 test data 的 inference:
   python3 test.py --data_root ../iclevr --
   weight_path ../model_weight_best/DL_lab6_112550069_謝侑哲.pth --test_type test
4. 關於 new_test data 的 inference:
   python3 test.py --data_root ../iclevr --
   weight_path ../model_weight_best/DL_lab6_112550069_謝侑哲.pth --test_type
   new_test