1. Introduction

在本次實驗報告中會講述如何設計 DDPM 的 model，並且如何加入 condition 的 label 到 DDPM 中，如何使用 noise schedule 和如何做 training 的部分，也會稍微帶到 DDPM 的原理和最後 inference 的結果，在實驗數據中也有使用不同的 embedding dimension 做測試，去比較不同 class embedding dimension 中的關係。

2. Implementation Details

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

在實作上都採用 Diffusers 所提供的所有套件，包括 UNet2DModel、noise schedule function、optimizer scheduler (get_cosine_schedule_with_warnmup) ，在這段中會介紹如何設計一個 Condition DDPM，還有他的基底 ClassConditionedUnet。

先來介紹 ClassConditionedUnet 的實作方式，下 condition 的方式都跟 diffusers 教學內容一致，會先將要下的 class condition 給轉成一個 embedding domain，在進入 UNet 之前會先將 24 個 one-hot encoding 的 label(classes)轉成 embedding size 大小的 embedding domain，然後才將這個 embedding 的資料連接到 UNet 去做訓練。

```python
10 class ClassConditionedUnet(nn.Module):
11     def __init__(self, num_classes=10, class_emb_size=4):
12         super().__init__()
13 
14         # The embedding layer will map the class label to a vector of size class_emb_size
15         self.class_emb = nn.Linear(num_classes, class_emb_size)
16         self.class_emb_size = class_emb_size
17         # Self.model is an unconditional UNet with extra input channels to accept the conditioning information (the class embedding)
18         self.model = UNet2DModel(
19             sample_size=64,           # the target image resolution
20             in_channels=3+class_emb_size, # Additional input channels for class cond.
21             out_channels=3,           # the number of output channels
22             layers_per_block=2,       # how many ResNet layers to use per UNet block
23             #block_out_channels=(32, 64, 128,256,512),  #
24             # block_out_channels=(128,128,256,256,512),
25             block_out_channels=(128,128,256,256,512,512),
26             down_block_types=(
27                 "DownBlock2D",        # a regular ResNet downsampling block
28                 "DownBlock2D",    # a ResNet downsampling block with spatial self-attention
29                 "DownBlock2D",        # a regular ResNet downsampling block
30                 "DownBlock2D",        # a regular ResNet downsampling block
31                 "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
32                 "AttnDownBlock2D",
33             ),
34             up_block_types=(
35                 "AttnUpBlock2D",
36                 "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
37                 "UpBlock2D",          # a regular ResNet upsampling block
38                 "UpBlock2D",          # a regular ResNet upsampling block
39                 "UpBlock2D",
40                 "UpBlock2D",          # a regular ResNet upsampling block
41               ),
42         )
43 
44     # Our forward method now takes the class labels as an additional argument
45     def forward(self, x, t, class_labels):
46         # Shape of x:
47         bs, ch, w, h = x.shape
48         # class conditioning in right shape to add as additional input channels
49         class_cond = self.class_emb(class_labels) # Map to embedding dimension
50         class_cond = class_cond.view(bs, self.class_emb_size, 1, 1).expand(bs, self.class_emb_size, w, h)
51         # Net input is now x and class cond concatenated together along dimension 1
52         net_input = torch.cat((x, class_cond), 1)
53         # Feed this to the UNet alongside the timestep and return the prediction
54         return self.model(net_input, t).sample
```

在第 15 行會先將輸入的 class label 轉成 class_emb_size 大小的 embedding domain，第 18 行道 41 行則是定義整個網路架構，有嘗試過很多種設計方式，最後發現直接以 6 層的 UNet 效果最好，在 6 層的 dowmsample 後的大小會是 1*1*512，因為受到輸入圖片大小(sample_size)的限制，因此設定 6 層為上限，而初始的就以 128 為初始，為了更好的學習深層的 feature 和 condition 之間的關係，會針對深層的網路增加注意力機制(AttnDownBlock2D)，且沒有快速的加倍 channel 數量，直接以 128、128、256、256、512、512 為最終的網路模型，其中可以看到下 Condition 的方式是直接以 UNet 的輸入加上 class_emb_size 的數量，因此在後續的實驗中會依照不同的 class_emb_size 做探討，如果挑選過大的 class_emb_size 會不會造成訓練不起來等情況。

在 forward 的設計上，輸入的部分會有原圖 x、noise 的 timestep 時間點 t、24 個 one-hot class 的 label (class_labels)，會先將 label 轉成 embedding domain，然後再跟原圖合併(將 condition 的資訊加在原圖 x 上)，然後再給專門為 Diffusion 設計的 UNet 的模型(UNet2DModel)當作輸入，最後

再從這個給定的 timestep t 去 sample 最後的結果。

在 train ddpm 時設計了一個 TrainDDPM 的 class，在建構子時會先定義上述所設計的 model (ClassConditionedUNet) 還有 training 的 dataset 和 validation 的 dataset，並且在 dataset 的設計上可以去調整一次 training 的 partial 和 val 的比例(後續都改成 partial=1 與 val_split = 0，也就是使用權部的資料及，不分割測試資料集)，圖片再經過 dataset 時會先做一次的 transform，如第二張圖的 217 行，會先將圖片轉成 64*64 的大小，然後轉成 PyTorch 的 Tensor，圖的像素就會從[0,255]轉道[0,1]的範圍，並且對圖片進行標準化，針對每個 RGB channel 進行標準化，使平均值為 0.5，標準差為 0.5，這樣能將圖片的像素從[0,1]的範圍轉換為[-1,1]，利於網路中的訓練。

```python
58  class TrainDDPM:
59  
60      def __init__(self,args,train_epochs):
61          self.args = args
62          print(self.args.class_emb_size)
63          self.model = ClassConditionedUnet(num_classes=24,class_emb_size=self.args.class_emb_size).to(self.args.device)
64          self.train_loader, self.val_loader = create_dataloaders('iclevr', 'train.json', 'objects.json',
65                                      batch_size=args.batch_size,
66                                      partial=args.partial,
67                                      num_workers=args.num_workers,
68                                      transform=transform,
69                                      val_split=val_split)
70  
71  
72  
73          self.train_epochs = train_epochs
74  
75          self.optimizer = torch.optim.AdamW(self.model.parameters(), lr=self.args.lr)
76          self.scheduler =  get_cosine_schedule_with_warmup(
77              optimizer=self.optimizer,
78              num_warmup_steps=0,
79              num_training_steps= len(self.train_loader) * 500,
80          )
81          self.noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='squaredcos_cap_v2')
82          self.enable_amp = True if args.device.count("cuda") >= 1 else False
83          self.scaler = amp.GradScaler(enabled=self.enable_amp)
84          print(self.enable_amp)
85          pytorch_total_params = sum(p.numel() for p in self.model.parameters())
```

```python
217     transform = transforms.Compose([
218         transforms.Resize((64, 64)),
219         transforms.ToTensor(),
220         transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))])
221     val_split = 0
222     if args.test:
223         val_split = 0.01
224 
225     root = f"ddpm_ckpt_{args.class_emb_size}" # try 128 (avoid bottleneck) 512 1024
226 
227     ddpmTrainer = TrainDDPM(args,args.epochs)
228 
229     if (args.ckpt_path != ""):
230         ddpmTrainer.load_model(args.ckpt_path)
231     if args.test:
232         ddpmTrainer.valid_one_epoch(-1)
233     else:
234         min_loss = 1e10
235         for epoch in range(args.start_epoch,args.epochs+1):
236             loss,lr = ddpmTrainer.training_one_epoch(epoch)
237             record_training_result(epoch, lr,loss,root)
238             #ddpmTrainer.valid_one_epoch(epoch)
239             if (loss < min_loss):
240                 min_loss = loss
241                 ddpmTrainer.save_model(epoch,root,suffix=f"loss={loss:.3f}")
242             elif (epoch  % args.save_per_epoch == 0):
243                 ddpmTrainer.save_model(epoch,root)
```

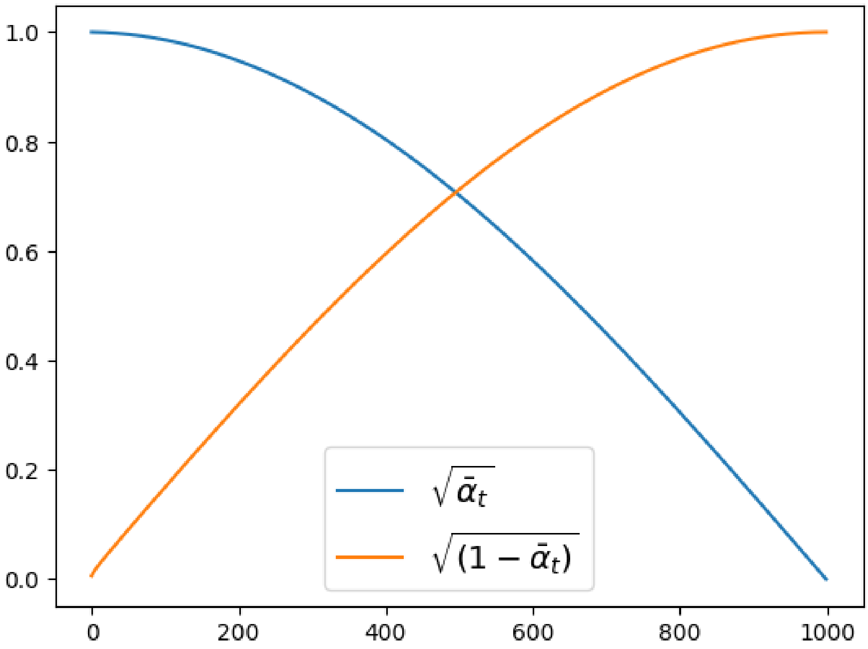
Training 所使用的 Optimizer 為 AdamW，learning rate 的 Scheduler 則是使用 Diffuser 所提供的 get_cosine_schedule_with_warmup，noise_scheduler 也是使用 Diffuser 所提供的 DDPMScheduler，在 DDPM 論文中，在每一個 timestep 中都會加入少量的高斯噪音 noise，而 Diffuser 所提供的套件可以透過 timestep 的步長去設定噪音量，如下方公式所表示的：

$$q(\mathbf{x}t|\mathbf{x}t-1) = \mathcal{N}(\mathbf{x}t; \sqrt{1-\beta_t}\mathbf{x}t-1, \beta_t\mathbf{I}) \quad q(\mathbf{x}_1|\mathbf{x}0) = \prod^{T} t = 1 q(\mathbf{x}t|\mathbf{x}t-1)$$

給定某個 timestep x_{t-1} 可以得到下一個加噪音的版本 x_t，從公式中可以看出將x_{t-1}按照$\sqrt{1-\beta_t}$做縮放，並加入按β_t的噪音，也就是說β是根據特定的 timestep 來定義的，並且決定每個 timestep 中添加的 noise 數量，但是想要直接從原圖去預測在 t 的時間點的噪音，假設今天要求x_{500}，為了避免直接執行 500 次，可以使用另一個公式

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$$

其中$\bar{\alpha}t = \prod i = 1^T \alpha_i$且 $\alpha_i = 1 - \beta_i$ 由下方的圖中可以看出藍色曲線$\sqrt{\bar{\alpha}_t}$表示的是原始圖片在每一個 timestep 中保留的部分，隨著 timestep 越大(t 越大)，$\sqrt{\bar{\alpha}_t}$會越來越小，相反的$\sqrt{1-\bar{\alpha}_t}$則是噪音的占比，最終會形成全部都是噪音的圖。



```python
    def training_one_epoch(self,epoch):
        self.current_epoch = epoch
        criterion = nn.MSELoss()
        total_loss = 0
        t = 0
        self.model.train()
        for (image,label) in (pbar := tqdm(self.train_loader, ncols=120)):

            image = image.to(self.args.device,dtype=torch.float32)
            label = label.squeeze()
            label = label.to(self.args.device,dtype=torch.float32)
            noise = torch.randn_like(image)
            timesteps = torch.randint(0, 999, (image.shape[0],)).long().to(self.args.device)
            noisy_image = self.noise_scheduler.add_noise(image, noise, timesteps)

            # Get the model prediction

            with amp.autocast(enabled=self.enable_amp):
                pred = self.model(noisy_image, timesteps, label) # Note that we pass in the labels y
                loss = criterion(pred, noise) # How close is the output to the noise

            self.scaler.scale(loss).backward()

            total_loss += loss
            # Backprop and update the params:
            self.scaler.step(self.optimizer)
            self.scaler.update()

            self.optimizer.zero_grad()
            self.scheduler.step()
            self.optimizer.step()


            t += 1
            avg_loss = total_loss / t

            # Store the loss for later
            self.tqdm_bar('train', pbar, loss=loss.detach().cpu(),avg_loss=avg_loss, lr=self.scheduler.get_last_lr()[0])

        return avg_loss.item(),self.scheduler.get_last_lr()[0]
```

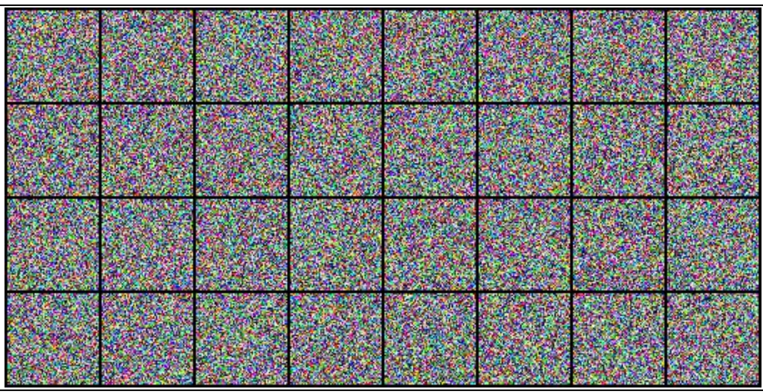
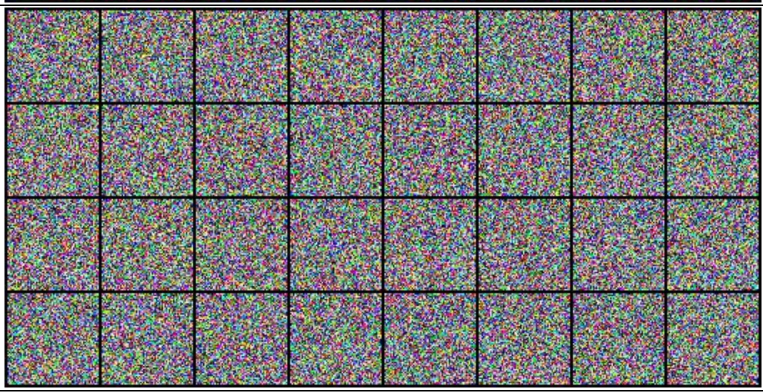
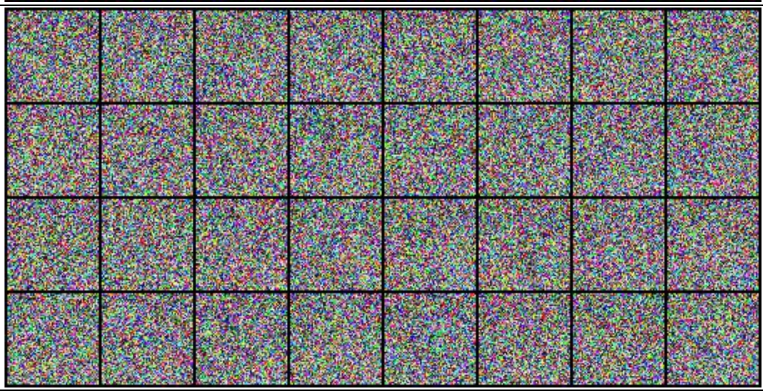
在 training_one_epoch 的程式中會隨著 noise_scheduler 和挑選到的 timestep 去對原圖片增加噪音，然後給 model 去做訓練，使用 MSE 針對噪音去求 loss 是為了能更精確的去學習如何去除噪音。

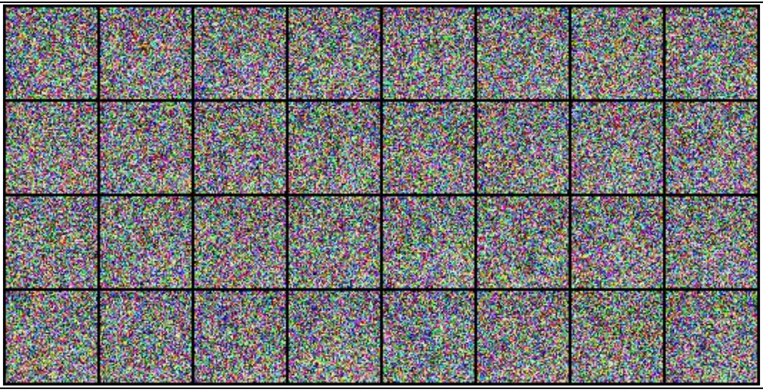
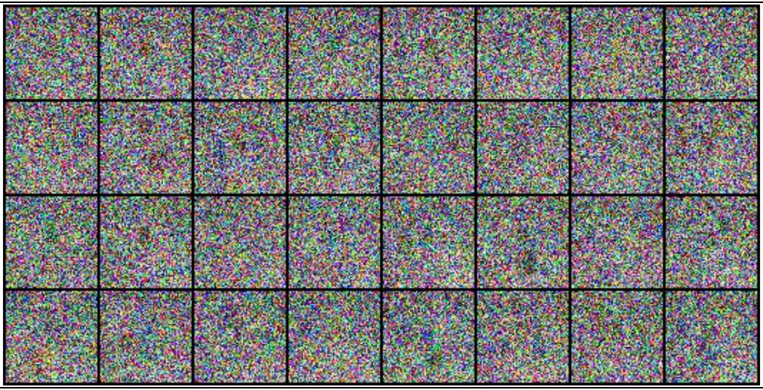
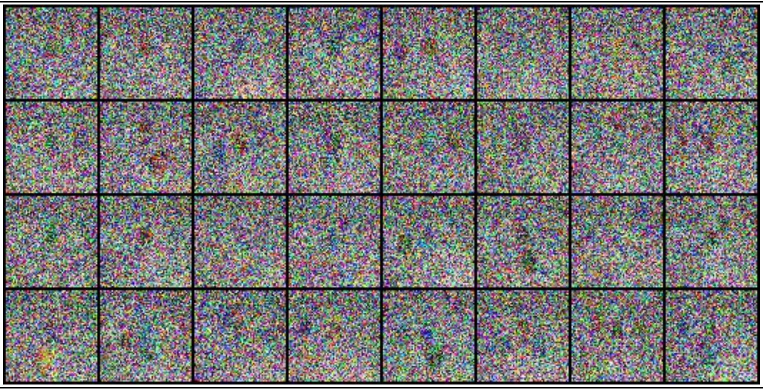
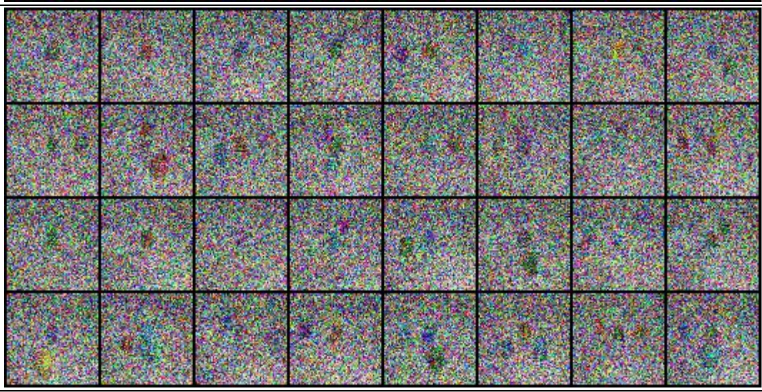
```python
    def valid_one_epoch(self,test_file,object_file,show_noise=False,all_test=False):
        transform=transforms.Compose([
            transforms.Normalize((0, 0, 0), (2, 2, 2)),
            transforms.Normalize((-0.5, -0.5, -0.5), (1, 1, 1)),
        ])
        label = load_test_data(test_file, object_file)


        # sampling
        with torch.inference_mode():
            batch_size = label.shape[0]
            x = torch.rand(batch_size,3,64,64).to(self.args.device)
            if (batch_size != 1):
                label = label.squeeze()
            label = label.to(self.args.device,dtype=torch.float32)

            for i, t in tqdm(enumerate(self.noise_scheduler.timesteps)):
                with torch.no_grad():
                    residual = self.model(x,t,label)

                x = self.noise_scheduler.step(residual,t,x).prev_sample


            ret = self.evaluate.eval(x, label)
            if (not all_test):
                print("ACC:",ret)
                img = transform(x)
                self.save_images(img, name="final_test")
        return ret
```

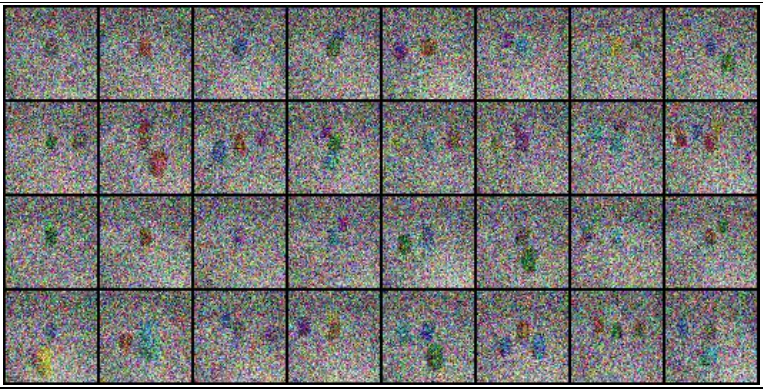
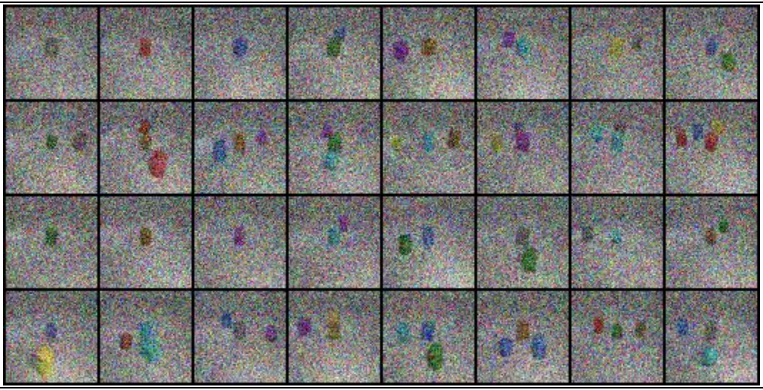
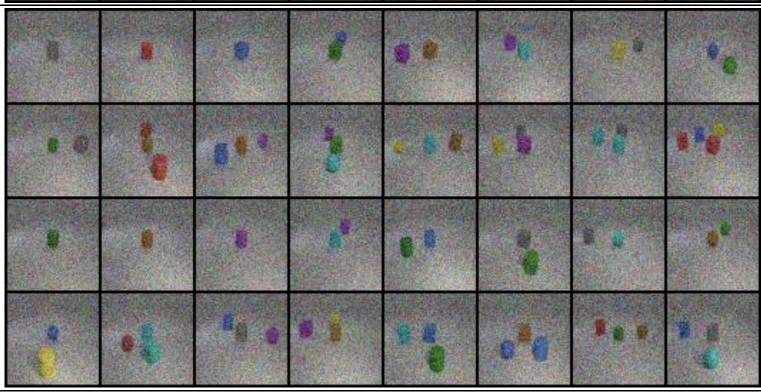
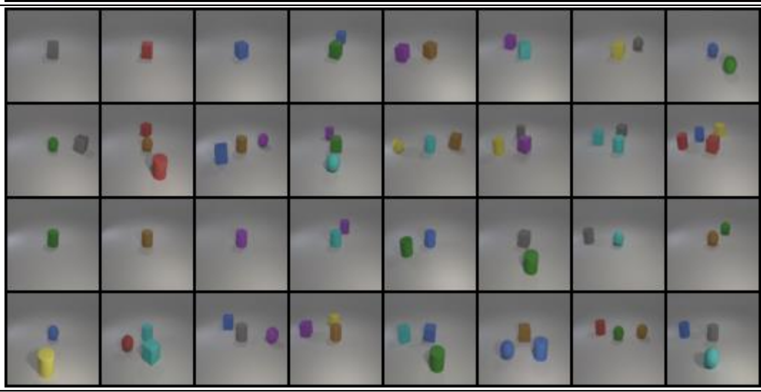
最後在 valid 時候直接以完全 noise 的圖(第 122 行)進行 timestep 次數的 sample，最後得到一張完整的圖，但是前面因為有做過 transform，因此還要將圖片轉換回原本的[0,1](第 137 行)。

3. Results and discussion

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

Timestep = 0	
Timestep = 100	
Timestep = 200	

Timestep = 300	
Timestep = 400	
Timestep = 500	
Timestep = 600	

Timestep = 700	
Timestep = 800	
Timestep = 900	
Timestep = 1000	

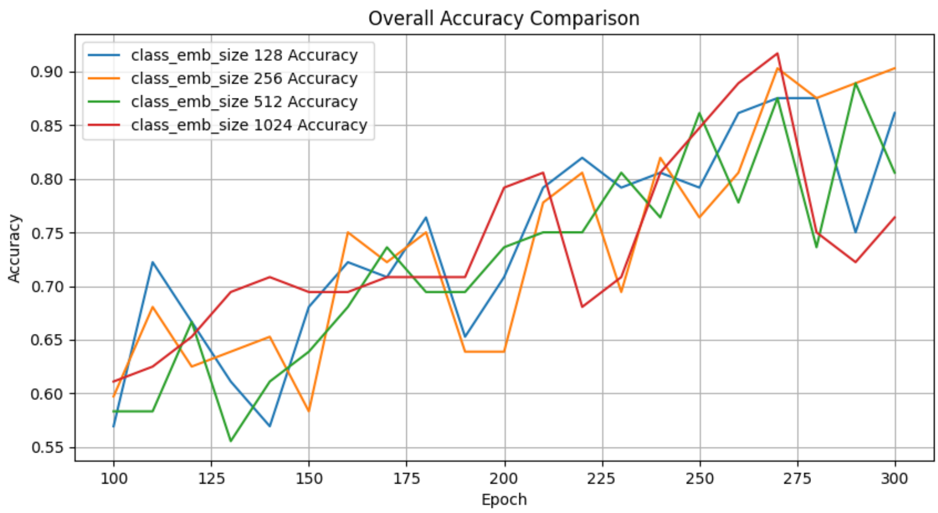
```
root@79b987ff2204:/workspace# python3 tester.py --ckpt_path ddpm_ckpt_512/ddpm_300_loss\=0.001.pth --gpu_ids 3 --class_e
mb_size 512 --batch_size 1
/workspace/evaluator.py:40: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
ecute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models fo
 functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this m
tart setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an i
  checkpoint = torch.load('./checkpoint.pth')
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is depre
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum
nt to passing `weights=None`.
  warnings.warn(msg)
tester.py:155: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which use
ry code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more detail
at could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless the
`weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHu
  self.model.load_state_dict(torch.load(path))
1000it [02:15,  7.36it/s]
ACC: 0.8472222222222222
```

此圖示以 test.json 作為 test 的目標，最後上傳的 model 以下方 Experiment Results 為準。

從 timestep 的過程中可以看到大概從 t=500 時就可以看出物品的輪廓了，接著到 t=1000 時最為明顯。

B. Discussion of your extra implementations or experiments

在介紹 class_emb_size 時候有提到，本實驗使用了多個不同的數據，有 128、256、512、1024 等，由於時間限制都訓練在 300 個 epoch 上下，由此圖可以很明顯地看的出來彼此之間的差異。



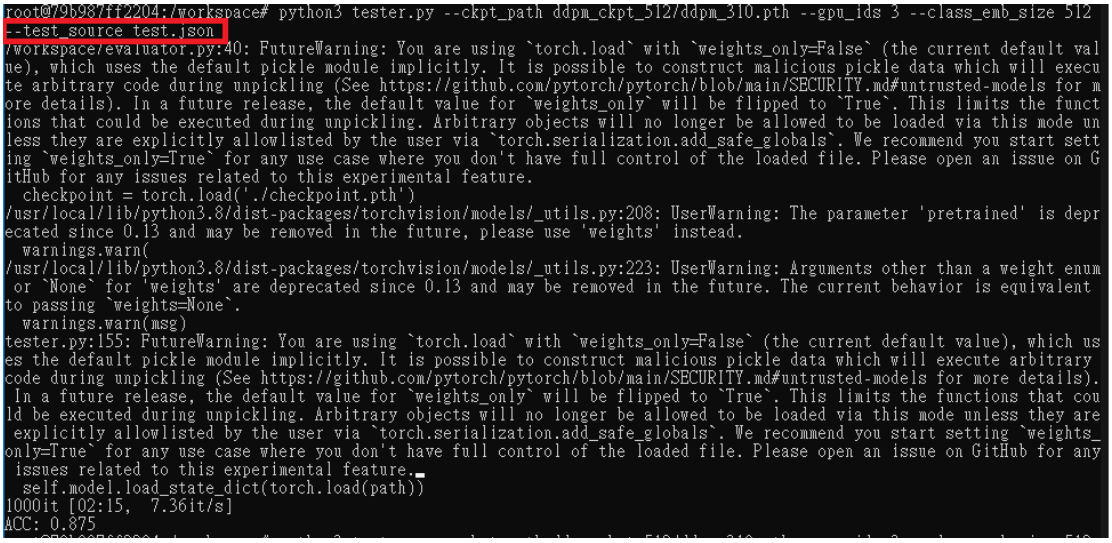
每個測試的種子碼都是固定的(seed=48763)，會發現隨著 epoch 的增長 accuracy 會越來越高，但是還是沒有辦法很穩定的去產生圖片，此圖也是依據(test.json)去做比較的圖，只取 100 至 300 個 epoch 之間的 accuracy 差異。

4. Experiment Results

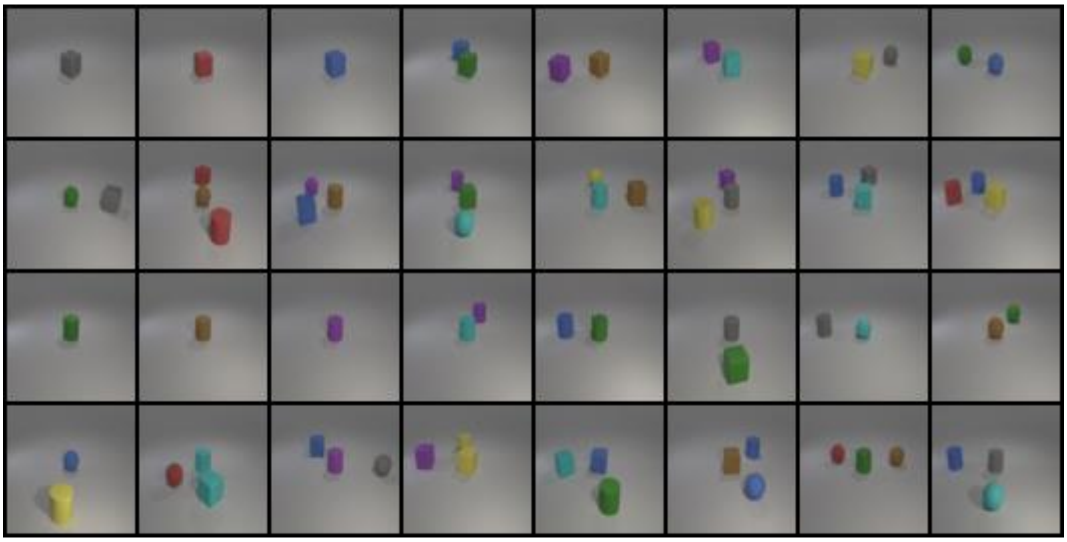
A. Classification accuracy on test.json and new test.json

Show your accuracy screenshots

Test.json (0.875 (87.5%))

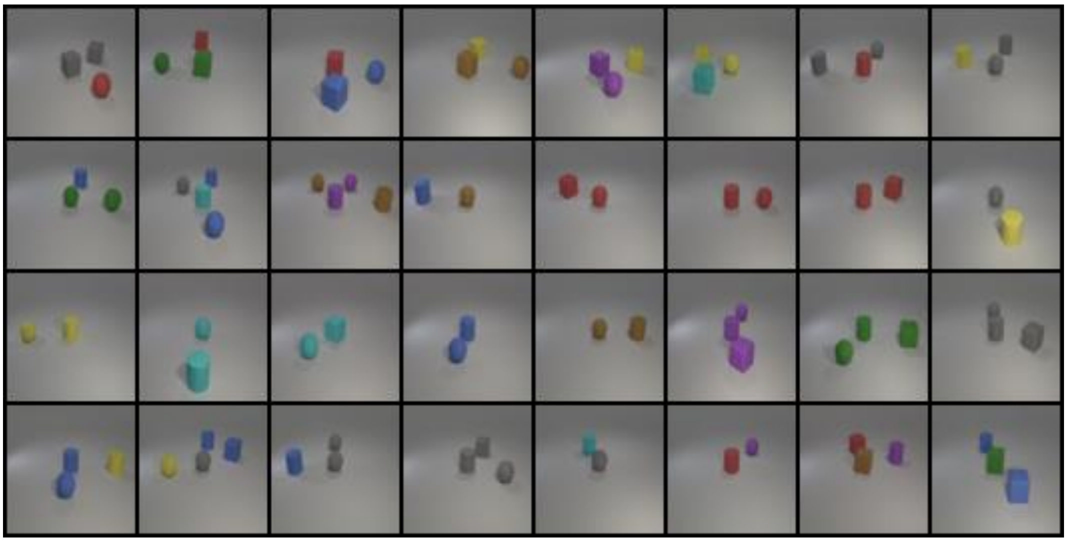


```
root@79b987ff2204:/workspace# python3 tester.py --ckpt_path ddpm_ckpt_512/ddpm_310.pth --gpu_ids 3 --class_emb_size 512
--test_source test.json
/workspace/evaluator.py:40: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
ue), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execu
te arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for m
ore details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the funct
ions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode un
less they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start sett
ing `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on G
itHub for any issues related to this experimental feature.
  checkpoint = torch.load('./checkpoint.pth')
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is depr
ecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.8/dist-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 equivalent
to passing `weights=None`.
  warnings.warn(msg)
tester.py:155: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which us
es 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#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 cou
ld be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are
 explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you 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.
  self.model.load_state_dict(torch.load(path))
1000it [02:15,  7.36it/s]
ACC: 0.875
```



New_test.json (0.892 (89.2%))

```
root@79b987ff2204:/workspace# python3 tester.py --ckpt_path ddpm_ckpt_512/ddpm_310.pth --gpu_ids 3 --class_emb_size 512
--test_source new_test.json
/workspace/evaluator.py:40: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
ue), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execu
te arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for m
ore details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the funct
ions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode un
less they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start sett
ing `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on G
itHub for any issues related to this experimental feature.
  checkpoint = torch.load('./checkpoint.pth')
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is depr
ecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.8/dist-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 equivalent
to passing `weights=None`.
  warnings.warn(msg)
tester.py:155: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which us
es 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#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 cou
ld be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are
 explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you 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.
  self.model.load_state_dict(torch.load(path))
488it [01:05,  7.24it/s]
1000it [02:15,  7.38it/s]
ACC: 0.8928571428571429
```



The command for inference process for both testing data

在 inference(testing)時可以使用此指令：

```
python3 tester.py --gpu_ids 0 --ckpt_path {pth_model_path} --class_emb_size 512 --test_source {path_to_test.json|path_to_new_test.json}
```

對其做詳細的講解為

- --gpu_ids 設定 gpu 的 ID
- --ckpt_path 設定要載入的 checkpoint.pth 位置
- --class_emb_size 設定對應大小 (因為上傳的 model 是 512 因此固定 512)
- --test_source 看是要選擇使用 test.json 或者 new_test.json 資料

當然得在下載之前可能需要下載所需套件，於程式碼中有提供 requirements.txt (如果因為套件原因無法執行的話)

5. Reference

https://github.com/huggingface/diffusers