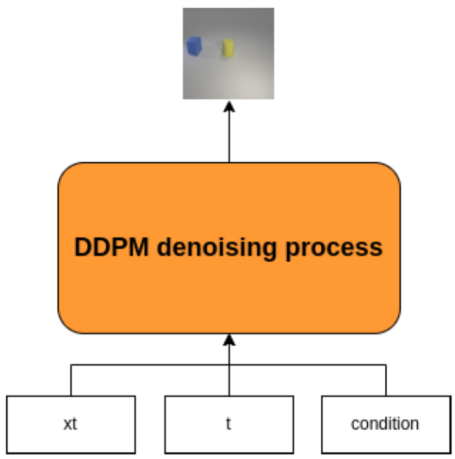
Lab6: Let’s Play DDPM

系級:智能系統　學號:312581006　姓名:張宸瑋

1. Report

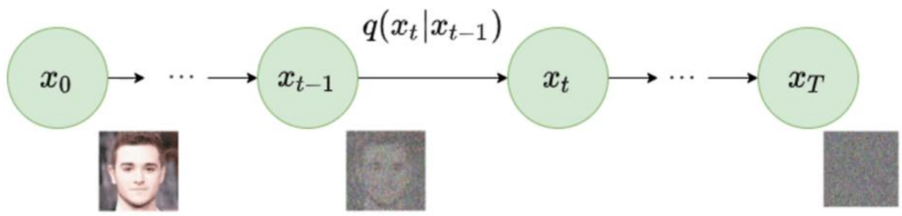
A. Introdunction

這次的 lab 主要是要實現 DDPM，根據多標籤條件，來生成符合的圖片。我們必須透過 ivlevr dataset 訓練一個 diffusion model，並且透過 test.json 以及 new_test.json 來生成指定條件的圖片。

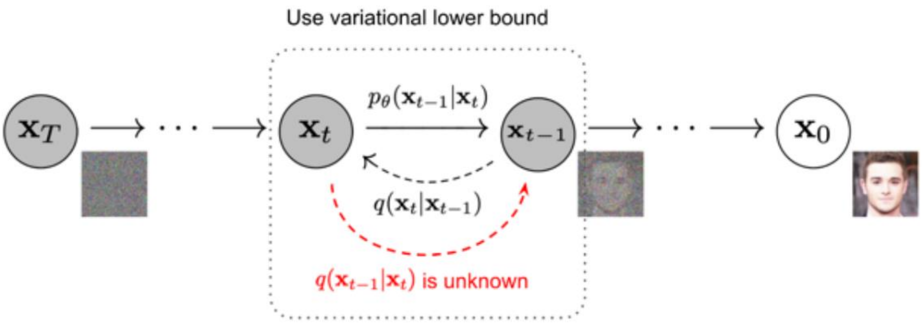


Conditional DDPM 示意圖

而 DDPM 的主要概念是，將輸入圖像經過多次的添加噪音，並且透過我們的網路，學習添加到的噪音，然後透過逐漸去噪的方式，來生成我們最後要生成的目標圖像。以下為示意圖。



DDPM 正向傳播示意圖



DDPM 逆向擴散示意圖

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar\alpha_t}\mathbf{x}_0 + \sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar\alpha_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

演算法架構

B. Implementation details

Design of UNet

```python
net = UNet2DModel(
    sample_size=64,       # the target image resolution
    in_channels=3,                 # additional input channels for class condition
    out_channels=3,
    layers_per_block=2,
    block_out_channels=(128, 128, 256, 256, 512, 512),
    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",
        "AttnUpBlock2D",        # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",            # a regular ResNet upsampling block
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
    class_embed_type = None,
)
net.class_embedding = nn.Linear(24 ,512)
```

Type of DDPM and Noise Schedule

```python
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='squaredcos_cap_v2')
```

```python
timesteps = torch.randint(0, noise_scheduler.config.num_train_timesteps, (img.shape[0],)).long().to("cuda")
noisy_x = noise_scheduler.add_noise(img, noise, timesteps)
```

Sampling

```python
def sample(net,noise_scheduler,dataloader):
    # Sampling loop
    for  img, cond in tqdm(dataloader, ncols=120):
        img = img.to("cuda")
        cond = cond.to("cuda")
        for t in(noise_scheduler.timesteps):
            # Get model pred
            with torch.no_grad():
                residual = net(img, t.to("cuda"), cond).sample
            # Update sample with step
            img = noise_scheduler.step(residual, t, img).prev_sample
    return img
```

Loss function

```python
# Our loss finction
loss_fn = nn.MSELoss()
```

```python
noise = torch.randn_like(img)

timesteps = torch.randint(0, noise_scheduler.config.num_train_timesteps, (img.shape[0],)).long().to("cuda")

#timesteps = torch.randint(0, 1000, (img.shape[0],)).long().to("cuda")
noisy_x = noise_scheduler.add_noise(img, noise, timesteps)
with accelerator.accumulate(net):
    # Get the model prediction
    pred = net(noisy_x, timesteps, cond).sample # Note that we pass in the labels y
    # Calculate the loss
    loss = loss_fn(pred, noise) # How close is the output to the noise
```

在loss function的選擇上我使用Mean Square Error，來計算預測出的噪音，以及添加到輸入的噪音的loss。

Specify the hyperparameters (learning rate, epochs, etc.)

1. Learning rate : 0.001
2. Epochs : 150
3. Optimizer : AdamW
4. lr_scheduler = get_cosine_schedule_with_warmup(
    optimizer=opt,
    num_warmup_steps=500,
    num_training_steps=(len(train_loader)* n_epochs),
)
5. Timestep : 1000

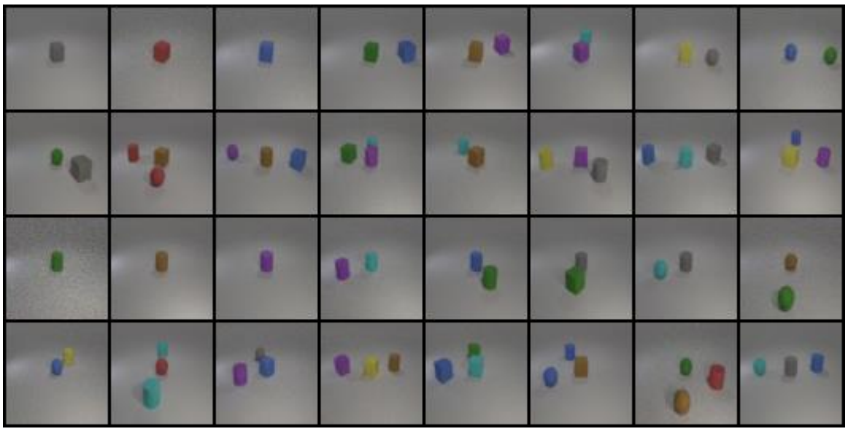
C. Result and discussion

- Show your accuracy screenshot based on the testing data

```
Test acc is 0.7222222222222222
(Epoch 92, lr:8e-05: 100%|                                    | 563/563 [03:26<00:00,  2.72it/s, loss=0.000591]
100%|                                                                  | 1/1 [00:06<00:00,  6.59s/it]
Finished epoch 92. Average of loss values: 0.00094
Test acc is 0.8611111111111112
(Epoch 93, lr:8e-05: 100%|                                    | 563/563 [03:26<00:00,  2.73it/s, loss=0.00324]
100%|                                                                  | 1/1 [00:06<00:00,  6.61s/it]
Finished epoch 93. Average of loss values: 0.00097
Test acc is 0.7638888888888888
(Epoch 94, lr:8e-05: 100%|                                    | 563/563 [03:26<00:00,  2.73it/s, loss=0.00186]
100%|                                                                  | 1/1 [00:06<00:00,  6.60s/it]
Finished epoch 94. Average of loss values: 0.00096
Test acc is 0.75
(Epoch 95, lr:8e-05: 100%|                                    | 563/563 [03:26<00:00,  2.72it/s, loss=0.00163]
100%|                                                                  | 1/1 [00:06<00:00,  6.60s/it]
Finished epoch 95. Average of loss values: 0.00095
Test acc is 0.8194444444444444
(Epoch 96, lr:8e-05: 100%|                                    | 563/563 [03:27<00:00,  2.72it/s, loss=0.000705]
100%|                                                                  | 1/1 [00:06<00:00,  6.60s/it]
Finished epoch 96. Average of loss values: 0.00100
Test acc is 0.9166666666666666
>>New best checkpoint save
(Epoch 97, lr:8e-05: 100%|                                    | 563/563 [03:27<00:00,  2.71it/s, loss=0.00071]
100%|                                                                  | 1/1 [00:06<00:00,  6.76s/it]
Finished epoch 97. Average of loss values: 0.00092
Test acc is 0.75
(Epoch 98, lr:8e-05: 100%|                                    | 563/563 [03:27<00:00,  2.72it/s, loss=0.00104]
100%|                                                                  | 1/1 [00:06<00:00,  6.61s/it]
Finished epoch 98. Average of loss values: 0.00098
Test acc is 0.8472222222222222
```

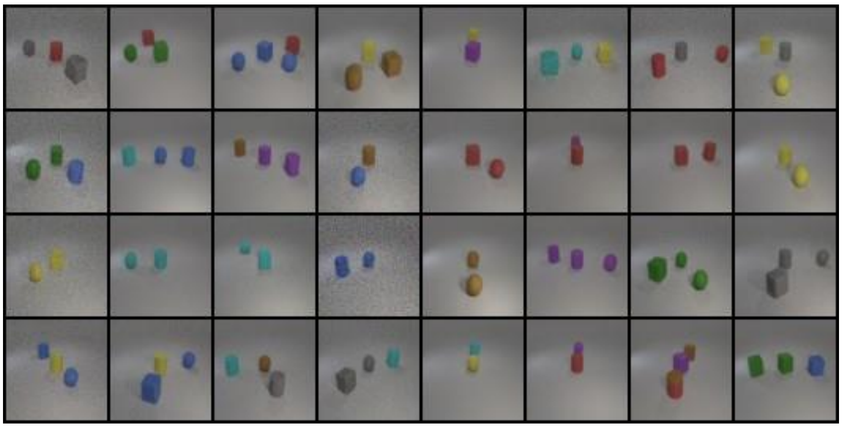
Testing Results

- Show your synthetic image grids and a progressive generation image.



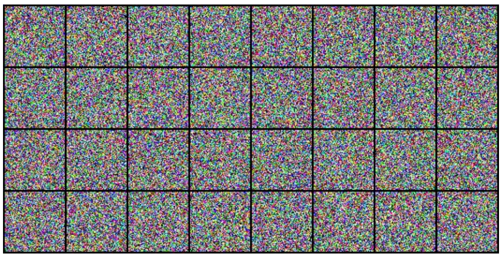
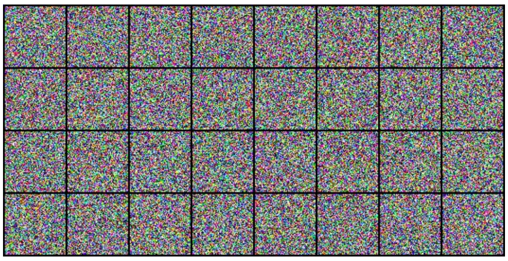
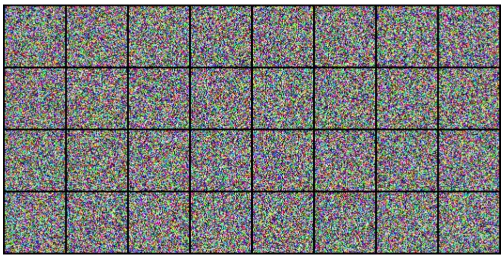
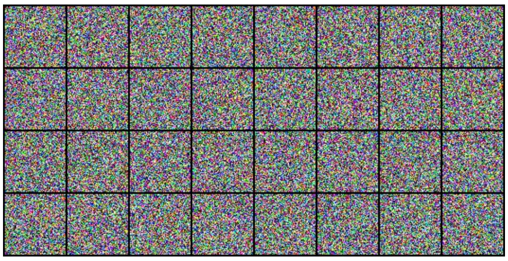
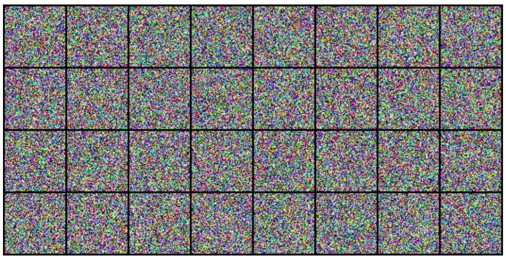
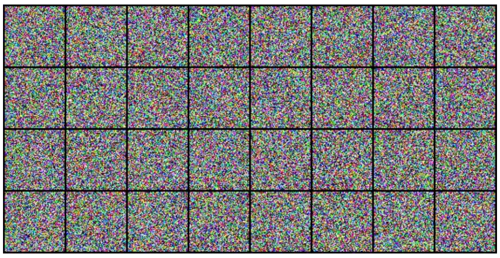
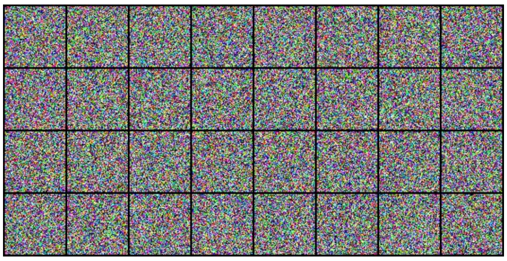
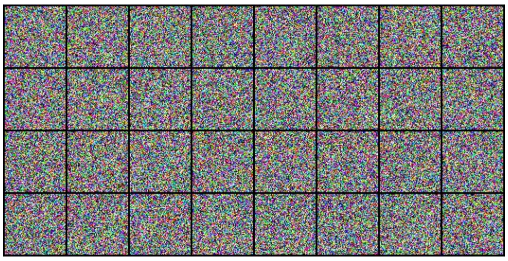
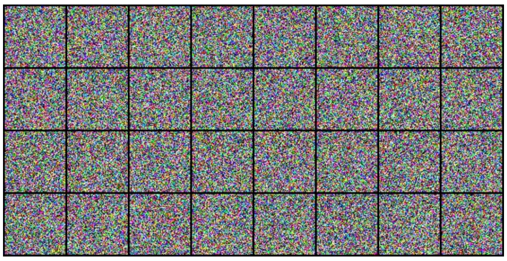
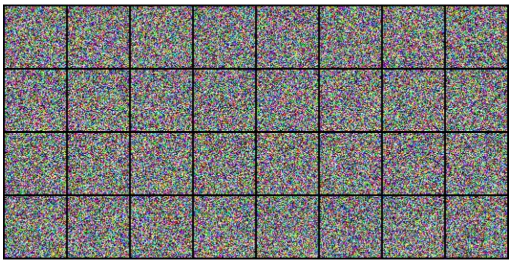
```
Test acc is 0.9166666666666666
```

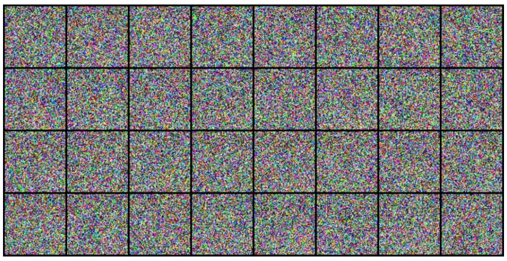
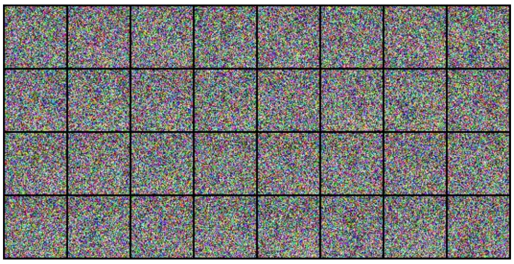
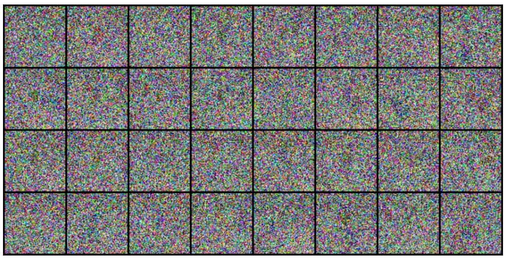
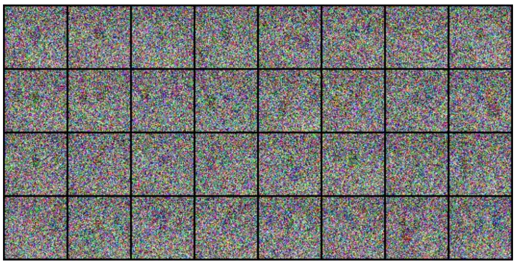
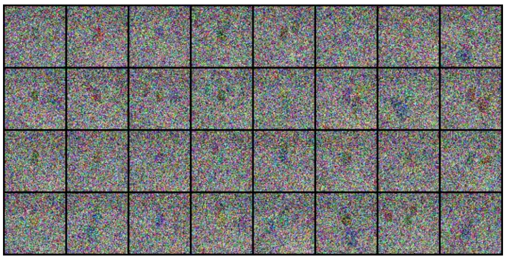
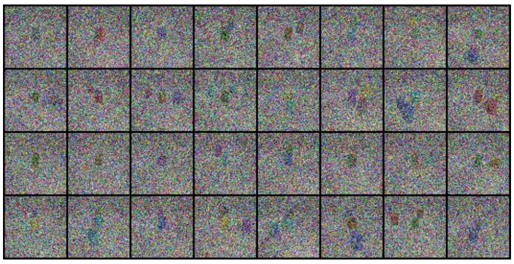
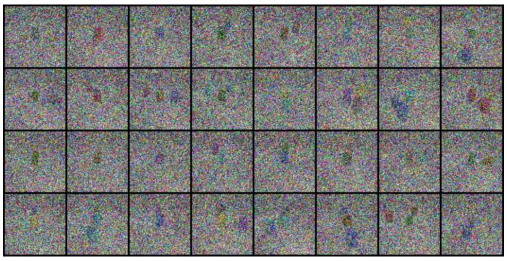
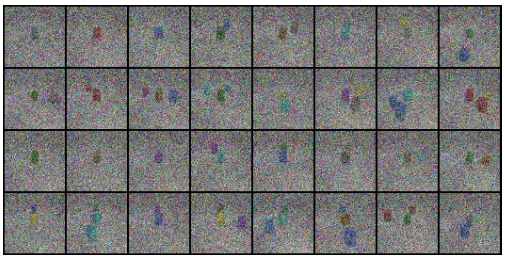
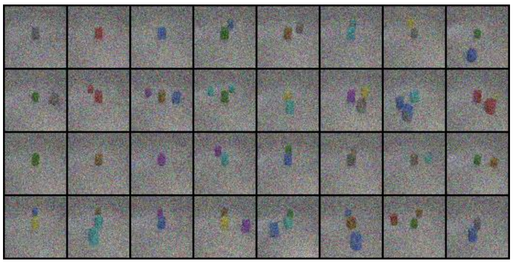
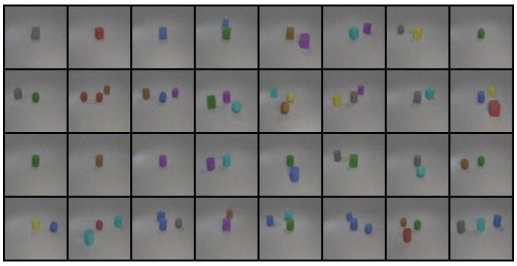
Test.json



```
New Test acc is 0.8928571428571429
```

New_test.json

	
Timestep = 0	Timestep = 50
	
Timestep = 150	Timestep = 200
	
Timestep = 250	Timestep = 300
	
Timestep = 350	Timestep = 400
	
Timestep = 450	Timestep = 500

	
Timestep = 550	Timestep = 600
	
Timestep = 650	Timestep = 700
	
Timestep = 750	Timestep = 800
	
Timestep = 850	Timestep = 900
	
Timestep = 950	Timestep = 999
Progressive generation image	

- Discuss the results of different model architectures or methods.

介紹:

在這次的實作中，我嘗試使用 DDIM 來取代 DDPM，對於 DDPM 來說，它需要較長的擴散步數，才能得到好的結果，假設我的擴散步數為 1000 步的話，那我再去噪的時候，就必須推理 1000 次，才能夠得到原圖，這樣是非常耗時的，因此透過 DDIM 的方式，它主要的目的是，透過更小的採樣步數，來加速我們的生成過程，它跟 DDPM 的訓練目的是一樣的，只是它不再限制擴散過程必須是一個馬爾科夫鏈。

DDIM 實現

```
noise_scheduler = DDIMScheduler.from_pretrained("google/ddpm-cifar10-32")
noise_scheduler.set_timesteps(num_inference_steps=50)
noise_scheduler.config.clip_sample = False
```

實驗結果比較

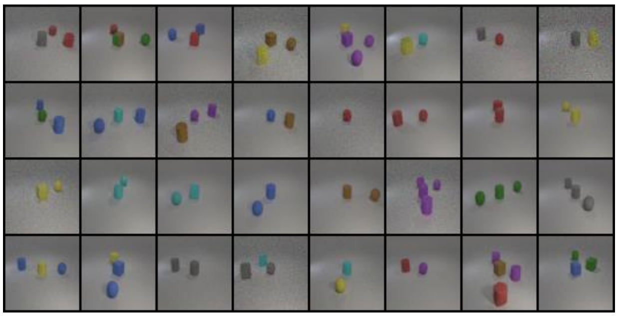
```
100%|██████████████████████████████████████| 1/1 [00:06<00:00,  6.21s/it]
```

DDIM

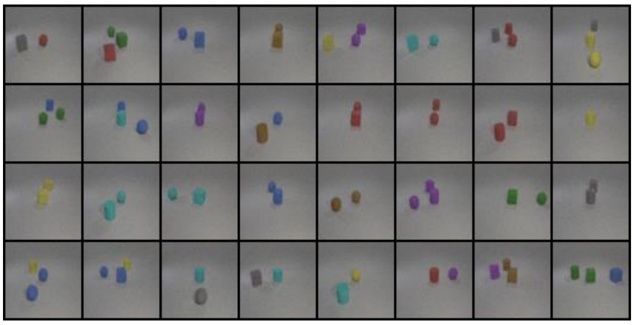
```
100%|██████████████████████████████████████| 1/1 [02:00<00:00, 120.02s/it]
```

DDPM

在這裡我們可以看到，兩者在推理的速度上的差異，假設我的 DDIM 的 num_inference_step = 50，其推理速度相較於 DDPM 是快 20 倍的。



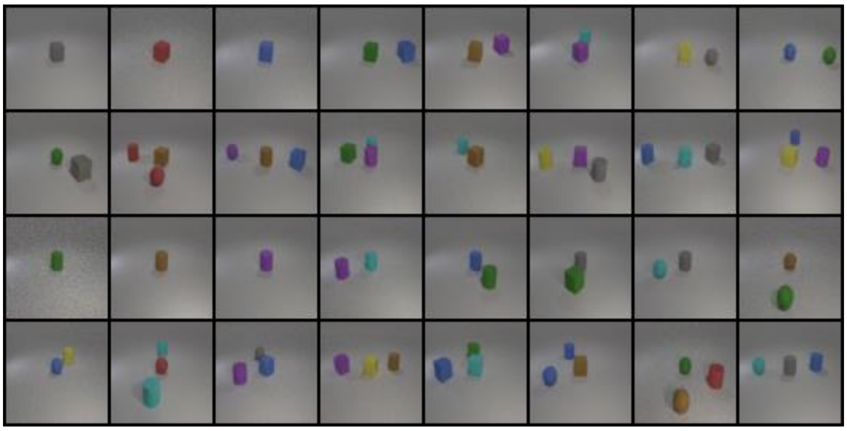
DDIM 推理結果



DDPM 推理結果

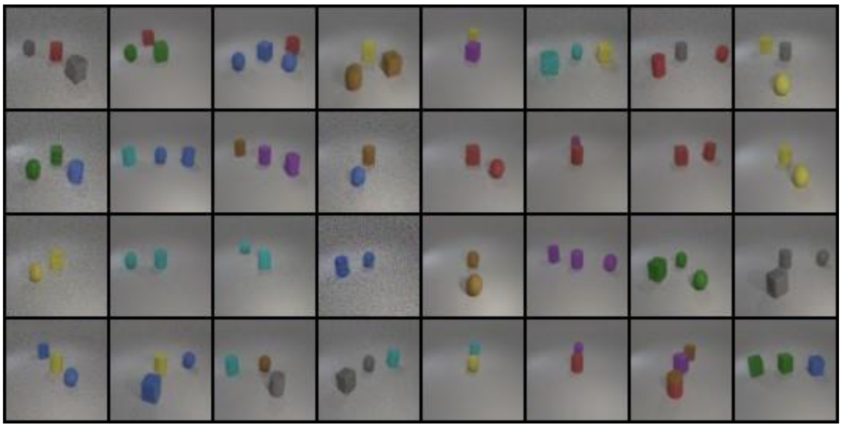
這裡我們也可以看到，即使 DDIM 相較於 DDPM，其採樣步數較小，也沒有因為這樣而失去圖片生成的品質，反而還優於 DDPM。

2. Experimental results



```
Test acc is 0.9166666666666666
```

test.json



```
New Test acc is 0.8928571428571429
```

new_test.json

程式碼執行注意事項:

1. 檔案目錄結構必須如下