一、 Introduction

在本實驗中聚焦在 DDPM 模型，以及如何將人類提供的標籤融入 DDPM 中，並利用這些標籤進行去噪和模型訓練。最後以 pretrain model 的正確率作為模型效能的指標之一。

二、 Implementation details

A. Model

```python
import torch
import torch.nn as nn
from diffusers import UNet2DModel

class UNet(nn.Module):
    def __init__(self, num_classes=24):
        super().__init__()
        self.dim = 512
        self.cond_embed = nn.Linear(num_classes, self.dim)
        self.model = UNet2DModel(
            sample_size=64,
            in_channels=3 + self.dim,
            out_channels=3,
            layers_per_block=3,
            block_out_channels=[128, 128, 256, 256, 512, 512],
            down_block_types=["DownBlock2D", "DownBlock2D", "DownBlock2D", "DownBlock2D", "AttnDownBlock2D", "AttnDownBlock2D"],
            up_block_types=["AttnUpBlock2D", "AttnUpBlock2D", "UpBlock2D", "UpBlock2D", "UpBlock2D", "UpBlock2D"]
        )
    def forward(self, x, label, t):
        label = label.float()
        batch = label.shape[0]
        cond = self.cond_embed(label).view(batch, self.dim, 1, 1).expand(batch, self.dim, 64, 64)
        x = torch.cat((x, cond), dim=1)

        label = label.float()
        return self.model(x, t).sample

if __name__ == "__main__":
    model = UNet()
    print(model)
    print(model(torch.randn(1, 3, 64, 64), torch.randint(0, 1, (1, 24), dtype=torch.float)).shape, 10)
```

程式碼實現條件化 U-Net 模型。先將 label 的為度用 Linear 轉到 512，在將其作為模型的輸入放進 Unet。模型結構基於 Unet2DModel，設置樣本大小為 64x64，輸入通道為 3 加上嵌入維度，輸出通道為 3，而在 Unet 模型當中，每一

個 block 都是由 ResNet 所構建的，除此之外，整體架構上 Unet 的最底下 2 層採用的是 Attention block，讓模型在底層能夠考慮所有訊息的相互關係。Forward 的時候會先將一個 multi-hot 的 label 透過 Linear 轉到 512 為度，並且將其接在 input x 之後和 t 一起丟入模型。

B. DataSet, DataLoader

```python
class Dataset(torchData):
    def __init__(self, mode='train', folder_path='iclevr'):
        super().__init__()
        self.mode = mode
        assert mode in ['train', 'val', 'test', 'new_test'], "There is no such mode !!!"
        if mode in ['test', 'new_test']:
            with open('test.json', 'r', encoding='utf-8') as file:
                datas = json.load(file)
            with open('new_test.json', 'r', encoding='utf-8') as file:
                new_datas = json.load(file)
            with open('objects.json', 'r', encoding='utf-8') as file:
                objects = json.load(file)
            label = [[1 if obj in data else 0 for obj in objects] for data in datas]
            new_label = [[1 if obj in data else 0 for obj in objects] for data in new_datas]
            self.label = label if mode == 'test' else new_label
            return

        self.folder_path = folder_path
        with open('train.json', 'r', encoding='utf-8') as file:
            data = json.load(file)
        with open('objects.json', 'r', encoding='utf-8') as file:
            self.objects = json.load(file)
        keys = list(data.keys())
        values = list(data.values())
        total_len = len(keys)
        train_len = int(total_len/3 * 0.9)*3

        if mode == 'train':
            self.data = keys[:train_len]
            self.label = values[:train_len]
        elif mode == 'val':
            self.data = keys[train_len:]
            self.label = values[train_len:]

        self.data = [os.path.join(self.folder_path, fname) for fname in self.data]
        self.transform = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5)),
        ])

    def __len__(self):
        return len(self.label)

    def __getitem__(self, index):
        if self.mode in ['test', 'new_test']:
            return torch.tensor(self.label[index])

        data = self.transform(imgloader(self.data[index]))
        label = torch.tensor([1 if obj in self.label[index] else 0 for obj in self.objects])
        return data, label
```

DataSet 紀錄該資料集的所有圖片檔名和標籤，並將 train、validation 的資料 9:1 拆分，而在做隨機存取的時候，會先讀取照片並做隨機翻轉以提升資料多樣性，標籤的表示方式為 multi-hot，代表對於所有 24 個類別，資料有包含該類別時候該 index 裡的值為 1 反之為 0。

C. BetaSchedular

```python
class BetaScheduler:
    def __init__(self, num_diffusion_timesteps=2000, beta_start=1e-4, beta_end=0.02, device='cuda'):
        self.device = device
        self.num_diffusion_timesteps = num_diffusion_timesteps
        self.beta_schedular = DDPMScheduler(num_train_timesteps=self.num_diffusion_timesteps, beta_start=beta_start, beta_end=beta_end, beta_schedule='squaredcos_cap_v2')


    def make_noise(self, x_start, t):
        x_start = x_start.to(self.device)
        noise = torch.randn_like(x_start, device=self.device)
        t = torch.tensor([t], device=self.device) if isinstance(t, int) else t
        return self.beta_schedular.add_noise(x_start, noise, t), noise

    def reverse(self, x_t, t, noise):
        # all value in t is same
        assert all(t == t[0])
        return self.beta_schedular.step(noise, t[0].cpu(), x_t).prev_sample
```

BetaScheduler 類別用於管理 diffusion 模型中的雜訊參數，負責產生雜訊並去噪。採用 DDPMScheduler 框架。make_noise 吃一個資料 x_start(x_0)和時間 t，產生與隨機雜訊 noise，然後透過 DDPMScheduler 內建函式 add_noise 將雜訊加入 x_start，傳回新增雜訊後的資料和原始雜訊，其 add_noise 遵循著以下公式產生雜訊圖片，reverse 也同樣採用其內建函式實作。

$$q(\boldsymbol{x_t} \mid \boldsymbol{x_0}) = \mathcal{N}\left(x_t; \sqrt{\overline{\alpha_t}}\boldsymbol{x_0}, (1-\overline{\alpha_t})\boldsymbol{I}\right)$$

D. Training Strategy

```python
def main(args):
    train_dataset = Dataset(mode='train', folder_path='iclevr')
    eval_dataset = Dataset(mode='val', folder_path='iclevr')
    train_loader = DataLoader(train_dataset,
                        batch_size=args.batch_size,
                        num_workers=args.num_workers,
                        shuffle=True)
    val_loader = DataLoader(eval_dataset,
                        batch_size=args.batch_size,
                        num_workers=args.num_workers,
                        shuffle=False)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    beta_scheduler = BetaScheduler(num_diffusion_timesteps=args.max_time_step, beta_start=1e-4, beta_end=0.02, device=device)
    # model = UNet(args.max_time_step, in_channels=3, out_channels=3).to(device) # 24 channels for 24 classes
    model = UNet()
    if args.load_path != '':
        print(f"Loading model from {args.load_path}")
        model.load_state_dict(torch.load(args.load_path))

    if torch.cuda.device_count() > 1:
        print(f"Using {torch.cuda.device_count()} GPUs for DataParallel.")
        model = nn.DataParallel(model)
    model = model.to(device)

    optimizer = optim.AdamW(model.parameters(), lr=args.lr)
    scheduler =  get_cosine_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps= len(train_loader) * 500,
    )
    criterion = nn.MSELoss()
    smart_save = SMARTSave(postfix = args.postfix)
```

本次實驗在正式 training 之前，會先將資料平均分布在所有的 GPU 上面以加快訓練速度，其他細項如 loss 採用 MSE，schedular 採用 diffusers 的內建 cos loss。

```python
for epoch in range(args.num_epoch):
    model.train()
    tq = tqdm(train_loader, ncols=args.ncols)
    total_loss = 0
    for i, (x, label) in enumerate(tq):
        x = x.to(device)
        label = label.to(device)
        t = torch.randint(0, args.max_time_step, (x.size(0),), device=device).long()
        x_t, noise = beta_scheduler.make_noise(x, t)
        pred_noise = model(x_t, label, t)
        loss = criterion(pred_noise, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()
        total_loss += loss.item()
        tq.set_description(f"Epoch [{epoch+1}/{args.num_epoch}], loss: {total_loss / (i + 1):.6f}")

    if (epoch + 1) % 2 == 0:
        model.eval()
        total_loss = 0
        with torch.no_grad():
            tq = tqdm(val_loader, ncols=args.ncols)
            for i, (x, label) in enumerate(tq):
                x = x.to(device)
                label = label.to(device)
                t = torch.randint(0, args.max_time_step, (x.size(0),), device=device).long()
                x_t, noise = beta_scheduler.make_noise(x, t)
                pred_noise = model(x_t, label, t)
                loss = criterion(pred_noise, noise)
                total_loss += loss.item()
                tq.set_description(f"[Eval] loss: {total_loss / (i + 1)}")
        avg_loss = total_loss / len(val_loader)
        smart_save(model.module if isinstance(model, nn.DataParallel) else model, avg_loss)
    if (epoch + 1) % 5 == 0:
        gt, label = eval_dataset[random.randint(0, len(eval_dataset)-1)]
        label = label.unsqueeze(0).to(device)
        img = inference(model, beta_scheduler, label, image_size=(3, 64, 64), device=device)
        save_image(img, os.path.join(f'inference{("_" if args.postfix != "" else "") + args.postfix}.png'), normalize=True)
        save_image(gt, os.path.join(f'ground_truth{("_" if args.postfix != "" else "") + args.postfix}.png'), normalize=True)
```

而在 training 的過程中使用 tqdm，首先會先隨機產生時間 t，並且透過 BetaSchedular 中的 make_noise 得到加上雜訊的圖片以及其雜訊。接著將雜訊圖片、標籤、時間 t 丟進模型當中預測雜訊 noise，最後透過 MSE 計算 loss。而每 2 個 epoch 則做一次 validation，並且每次儲存 validation loss 最小的那個模型。每 5 個 epoch 則做一次 inference 可視化，將 validation data 的資料隨機抽取一個做 inference 得出照片並記錄起來。

E. Inference

```
def inference(model, beta_scheduler, label, image_size=(3, 64, 64), device='cuda'):
    model.eval()
    with torch.no_grad():
        batch_size = label.shape[0]
        x_t = torch.randn(batch_size, *image_size, device=device)
        for t in range(beta_scheduler.num_diffusion_timesteps - 1, -1, -1):
            t_tensor = torch.full((batch_size,), t, device=device, dtype=torch.long)
            pred_noise = model(x_t, label, t_tensor)
            x_t = beta_scheduler.reverse(x_t, t_tensor, pred_noise)
        x_t = x_t.clamp(-1, 1)
        return x_t
```

Inference 函數作為生成圖片的函式，其假設圖片加上許多次的雜訊後，其樣態接近常態分佈，則一開始設 x_t 為一個常態分佈的圖像，透過模型預測的雜訊丟給 BetaSchedular 做 reverse，

隨著 t_tensor 的值越來越小，x_t 會越來越接近 label 標籤所表示的圖片，而在最終將圖片數值固定在-1~1 之間並回傳生成後的圖片。

F. Test

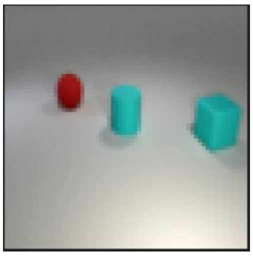
```python
def main(args):
    device = torch.device('cuda:' + args.gpu if torch.cuda.is_available() else "cpu")
    # model = UNet(args.max_time_step, in_channels=3, out_channels=3).to(device)
    model = UNet().to(device)
    model.load_state_dict(torch.load(args.model_path, map_location=device))
    beta_scheduler = BetaScheduler(num_diffusion_timesteps=args.max_time_step, beta_start=1e-4, beta_end=0.02, device=device)

    label, new_label = get_test_dataset(device=device)

    image = inference(model, beta_scheduler, label, image_size=(3, 64, 64), device=device)
    os.makedirs(os.path.join('images', 'test'), exist_ok=True)
    for i in range(image.shape[0]):
        save_image(image[i], os.path.join('images', 'test', f'{i}.png'), normalize=True)
    print("Saved images to images/test")

    new_image = inference(model, beta_scheduler, new_label, image_size=(3, 64, 64), device=device)
    os.makedirs(os.path.join('images', 'new_test'), exist_ok=True)
    for i in range(new_image.shape[0]):
        save_image(new_image[i], os.path.join('images', 'new_test', f'{i}.png'), normalize=True)
    print("Saved new images to images/new_test")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--model_path', '-p', type=str, default='result/best_model.pt', help='Path to the trained model')
    parser.add_argument('--max_time_step', '-st', type=int, default=2000, help='Number of diffusion steps')
    parser.add_argument('--gpu', '-g', type=str, default='0', help='GPU index to use')
    args = parser.parse_args()

    main(args)
```

Test 將 test.json 和 new_test.json 中的標籤透過 inference 生成圖片並儲存在 images 資料夾中。

三、 Results and discussion

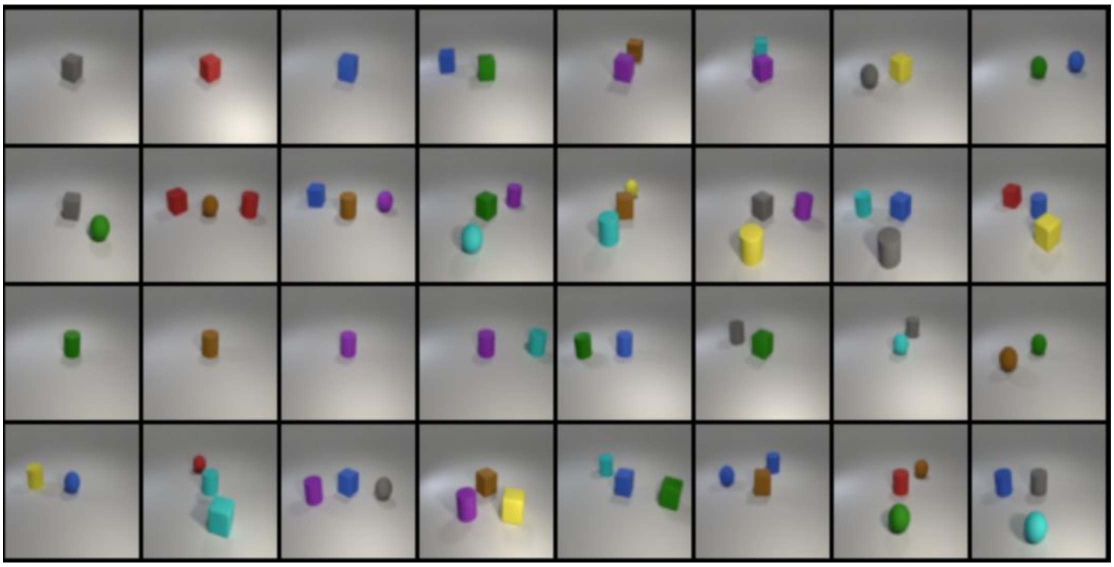
```
ray@NV11 ~/DLP/lab6 main !4 ?4
> python score.py
/mnt/nfs/work/ray/DLP/lab6/env/
precated since 0.13 and may be
  warnings.warn(
/mnt/nfs/work/ray/DLP/lab6/env/
um or `None` for 'weights' are
ts=None`.
  warnings.warn(msg)
Test set accuracy: 0.70833333333
New test set accuracy: 0.7261904
```

accuracy **0.715**



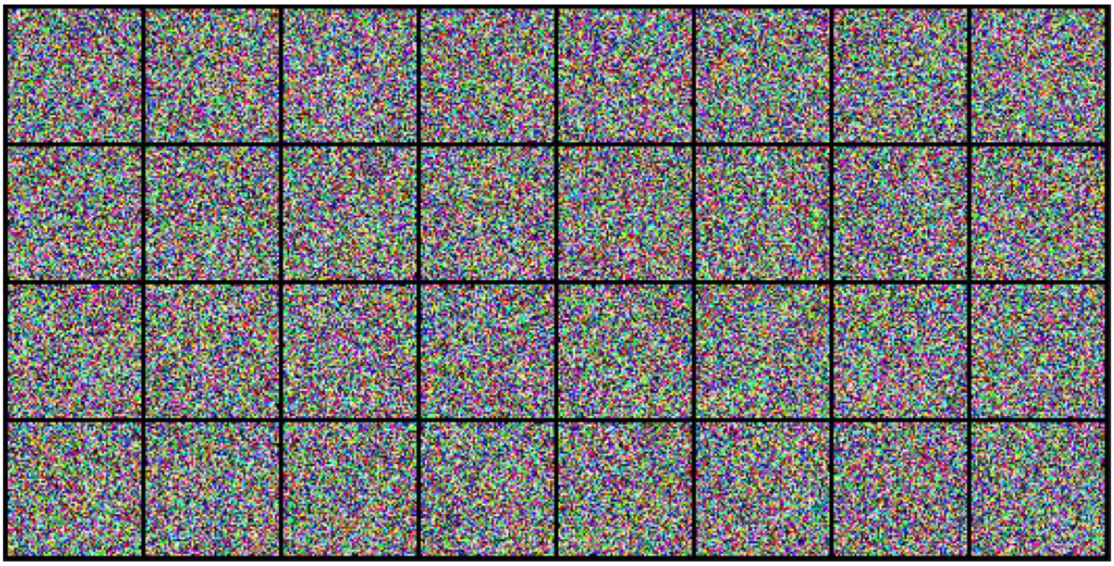
cyan cube, red sphere, cyan cylinder

A. Test Result

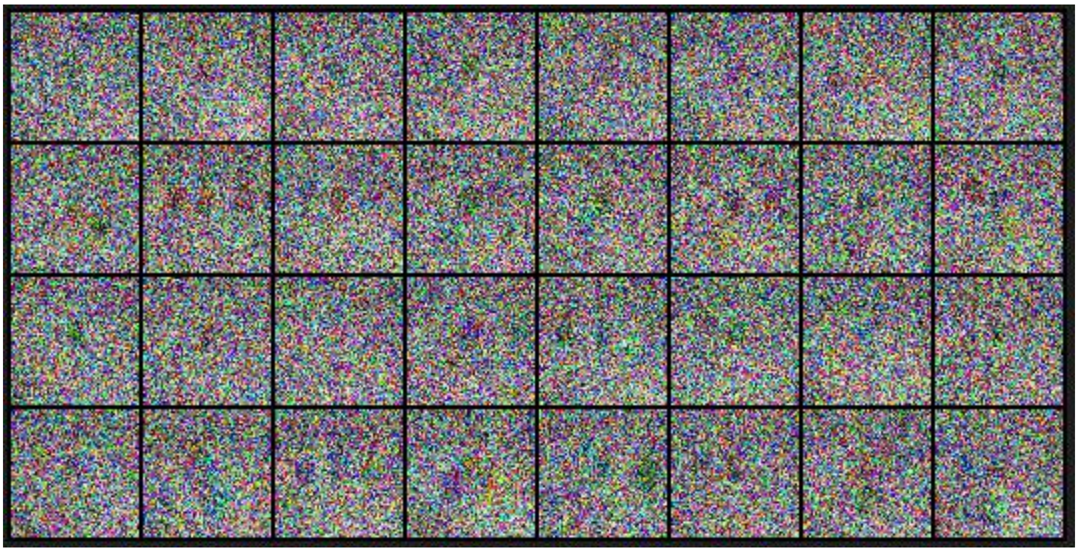


a. denoising process

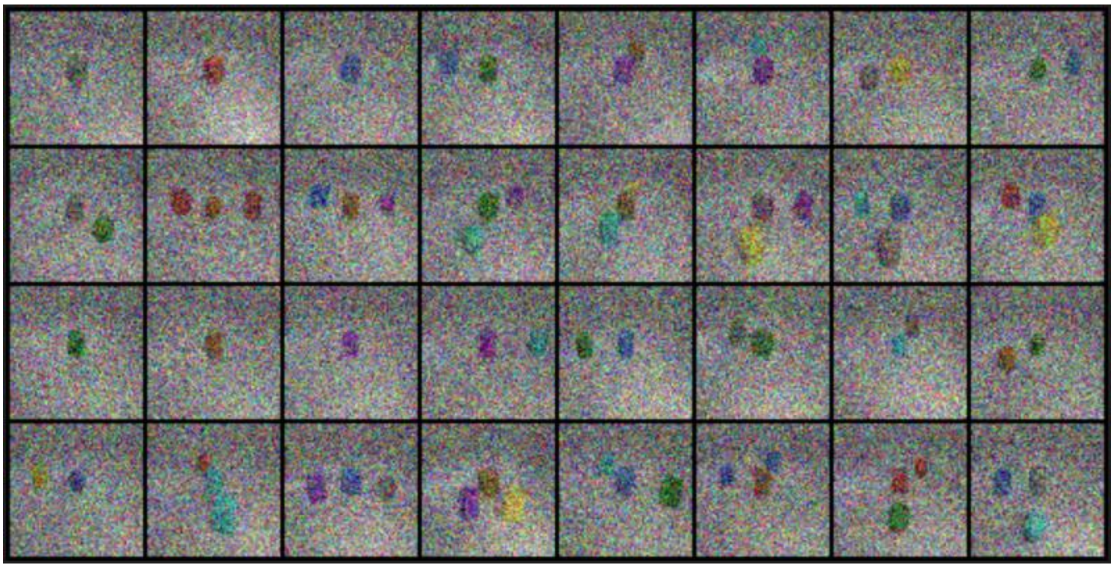
i. t = 900



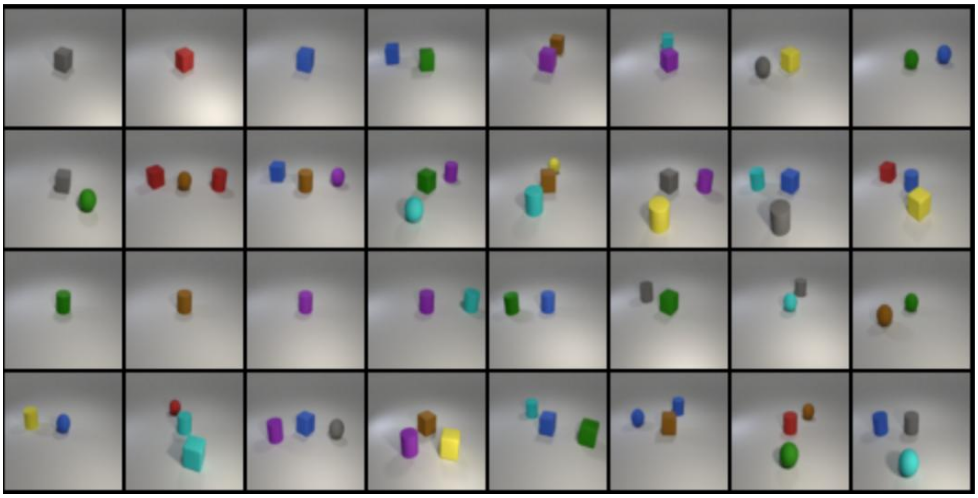
ii. t = 500



iii. t = 200

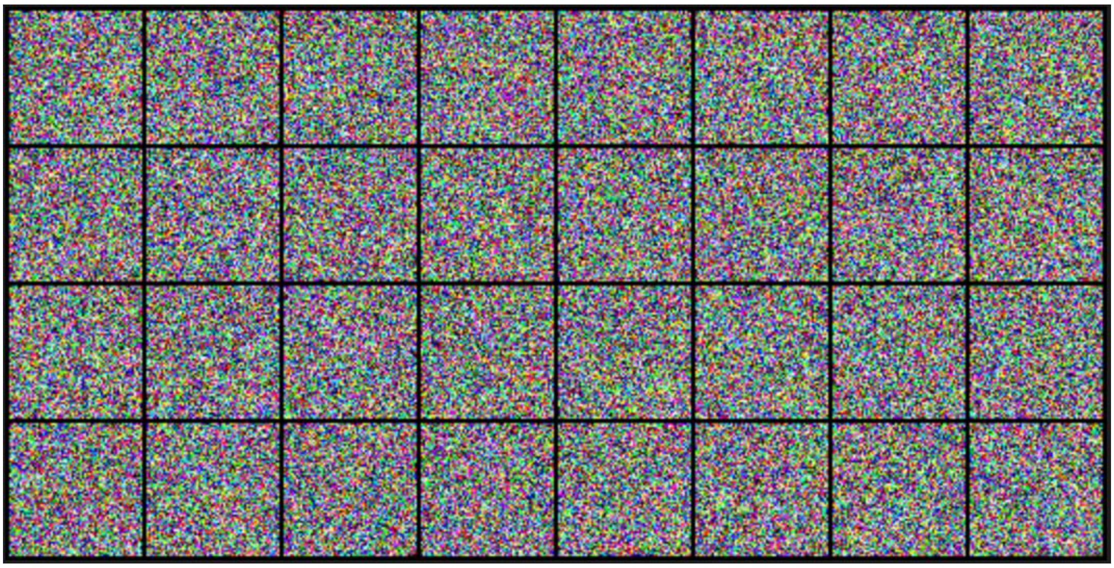


iv. t = 0

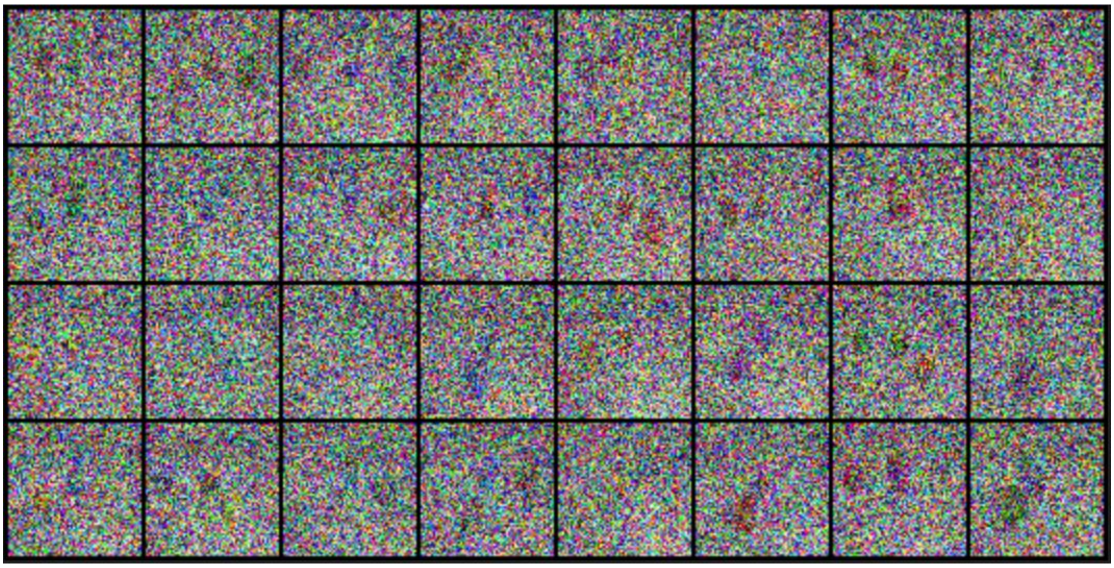


B. New Test Result

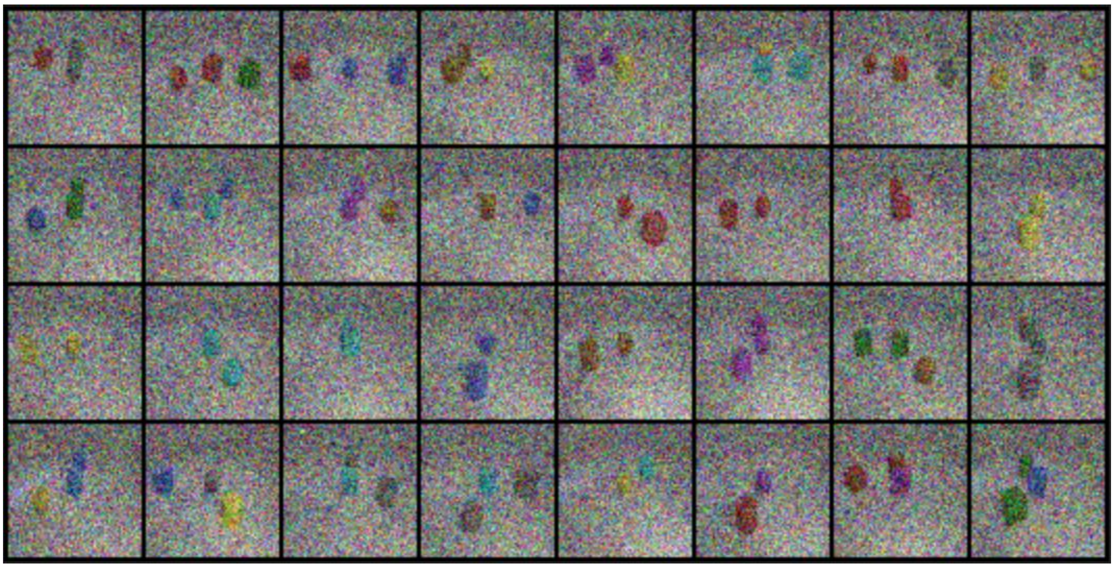
a. t = 900



b. t = 500



c. t = 200



d. t = 0

四、 extra experiments

在使用 diffusers 之前有實作過一個沒有 Attention block 的 UNet 版本，其使用之前 Lab 的程式碼實作，加上了 sin cos 去做 time embed，以及 Linear 去做 label embed，並且將這些值 broadcast 到 UNet 中的每一層。然而實驗

發現若沒有 Attention block，模型容易產生 label 是紅色方塊卻產生出完美的綠色圓形，推測為少了 Attention block 使得 label 的資料無法讓模型被考慮到，讓模型更偏向於把圖片生成好。最終使用 evaluator.py 只有 0.323 的準確率。

```python
class UNet(nn.Module):
    def __init__(self, time_step, in_channels = 3, out_channels = 3, num_classes = 24):
        super(UNet, self).__init__()
        self.down1 = DOWN(in_channels, 64)
        self.down2 = DOWN(64 * 2, 128)
        self.down3 = DOWN(128 * 2, 256)
        self.down4 = DOWN(256 * 2, 512)
        self.bottleneck = CC(512 * 2, 512)
        self.up4 = UP(512 * 2, 256, skip_channels=512)
        self.up3 = UP(256 * 2, 128, skip_channels=256)
        self.up2 = UP(128 * 2, 64, skip_channels=128)
        self.up1 = UP(64 * 2, out_channels, skip_channels=64)
        self.final_conv = nn.Conv2d(out_channels * 2, out_channels, kernel_size=1)

        self.dim = 128
        self.cond_embed = nn.Sequential(
            nn.Embedding(num_classes, self.dim),
            nn.Linear(self.dim, self.dim),
            nn.ReLU(),
            nn.Linear(self.dim, self.dim)
        )
        self.time_embed = nn.Parameter(self.sinusoidal_embedding(time_step, self.dim), requires_grad=False)
        self.boardcast_down1 = nn.Linear(self.dim * 2, 64)
        self.boardcast_down2 = nn.Linear(self.dim * 2, 128)
        self.boardcast_down3 = nn.Linear(self.dim * 2, 256)
        self.boardcast_down4 = nn.Linear(self.dim * 2, 512)
        self.boardcast_bottleneck = nn.Linear(self.dim * 2, 512)
        self.boardcast_up4 = nn.Linear(self.dim * 2, 256)
        self.boardcast_up3 = nn.Linear(self.dim * 2, 128)
        self.boardcast_up2 = nn.Linear(self.dim * 2, 64)
        self.boardcast_up1 = nn.Linear(self.dim * 2, out_channels)
    def sinusoidal_embedding(self, timesteps, dim):
        half_dim = dim // 2
        emb = torch.log(torch.tensor(10000.0)) / (half_dim - 1)
        emb = torch.exp(torch.arange(half_dim, dtype=torch.float) * -emb)
        emb = torch.arange(timesteps, dtype=torch.float).unsqueeze(1) * emb.unsqueeze(0)
        emb = torch.cat([torch.sin(emb), torch.cos(emb)], dim=1)
        if dim % 2 == 1:
            emb = torch.pad(emb, (0, 1, 0, 0))
        return emb
```

```python
def forward(self, x, cond, t):
    cond_embed = torch.matmul(cond.float(), self.cond_embed[0].weight)
    cond_embed = self.cond_embed[1:](cond_embed)
    t_embed = self.time_embed[t]

    embed = torch.cat([cond_embed, t_embed], dim=1)

    x1, o1 = self.down1(x)
    embed1 = self.boardcast_down1(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x1.size(2), x1.size(3))
    x1 = torch.cat([x1, embed1], dim=1)

    x2, o2 = self.down2(x1)
    embed2 = self.boardcast_down2(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x2.size(2), x2.size(3))
    x2 = torch.cat([x2, embed2], dim=1)

    x3, o3 = self.down3(x2)
    embed3 = self.boardcast_down3(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x3.size(2), x3.size(3))
    x3 = torch.cat([x3, embed3], dim=1)

    x4, o4 = self.down4(x3)
    embed4 = self.boardcast_down4(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x4.size(2), x4.size(3))
    x4 = torch.cat([x4, embed4], dim=1)

    x5 = self.bottleneck(x4)
    embed5 = self.boardcast_bottleneck(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x5.size(2), x5.size(3))
    x5 = torch.cat([x5, embed5], dim=1)

    x6 = self.up4(x5, o4)
    embed6 = self.boardcast_up4(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x6.size(2), x6.size(3))
    x6 = torch.cat([x6, embed6], dim=1)

    x7 = self.up3(x6, o3)
    embed7 = self.boardcast_up3(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x7.size(2), x7.size(3))
    x7 = torch.cat([x7, embed7], dim=1)

    x8 = self.up2(x7, o2)
    embed8 = self.boardcast_up2(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x8.size(2), x8.size(3))
    x8 = torch.cat([x8, embed8], dim=1)

    x9 = self.up1(x8, o1)
    embed9 = self.boardcast_up1(embed).view(embed.size(0), -1, 1, 1).expand(-1, -1, x9.size(2), x9.size(3))
    x9 = torch.cat([x9, embed9], dim=1)

    return self.final_conv(x9)
```

五、 Execute

執行 score.py 即可

要訓練模型則先執行 train.py 後執行 test.py 再執行 score.py