Deep Learning Lab6: Generative Models

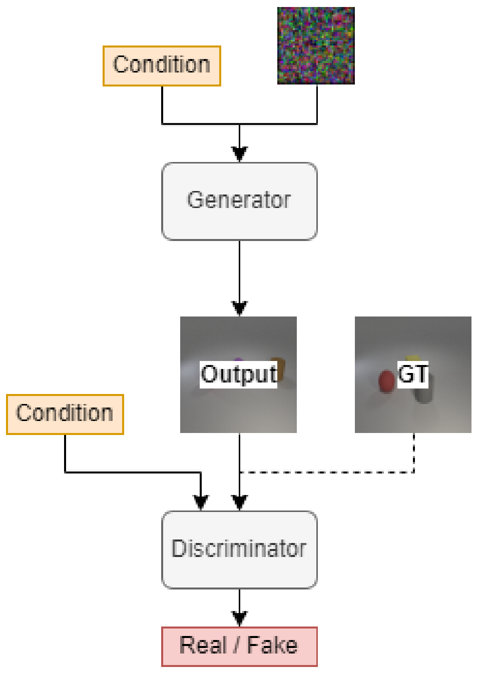
ID: 412551030   Name: 黃羿寧

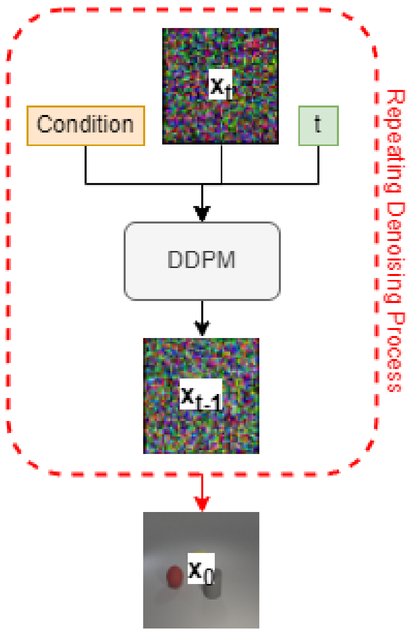
1. Introduction

In Lab6, we were requested to implement two kinds of typical generative model: Denoising Diffusion Probabilistic Models (DDPMs)[1] and Generative Adversarial Networks (GANs)[2]. Specifically, we applied two conditional generative model to generate multi-label object images as described in **test.json** and **new_test.json**. The reference images were from a modified CLEVR (Compositional Language and Elementary Visual Reasoning) dataset[3], namely ICLEVR. The ICLEVR dataset contains 18009 images, which includes one to three objects selected from the object label dictionary such as "gray cube" and "red cube". Furthermore, In the following section, I will show the details of DDPM and GAN used in this lab. Followed by the methods, I will show the classification accuracy of two models and discuss the special trick used in the training process. In this lab, conditional DDPM outperforms the conditional GAN with the classification accuracy of above 80% in two synthetic images generated by DDPM and around 50%-70% in those generated by GAN.

Dataset: CLEVR (Compositional Language and Elementary Visual Reasoning)

Object label dictionary: {"gray cube": 0, "red cube": 1, "blue cube": 2, "green cube": 3, "brown cube": 4, "purple cube": 5, "cyan cube": 6, "yellow cube": 7, "gray sphere": 8, "red sphere": 9, "blue sphere": 10, "green sphere": 11, "brown sphere": 12, "purple sphere": 13, "cyan sphere": 14, "yellow sphere": 15, "gray cylinder": 16, "red cylinder": 17, "blue cylinder": 18, "green cylinder": 19, "brown cylinder": 20, "purple cylinder": 21, "cyan cylinder": 22, "yellow cylinder": 23}





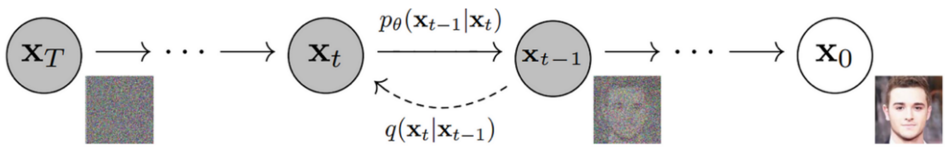
2. Implementation Details

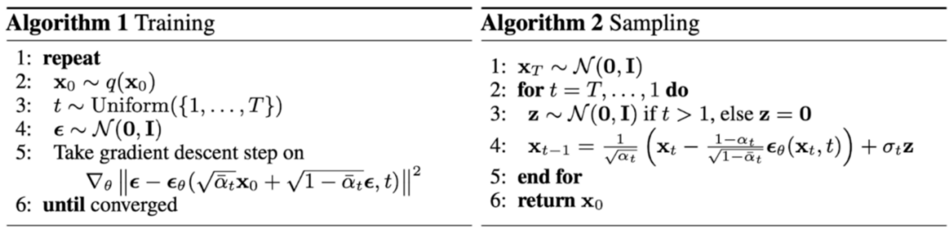
A. Data Loader

Data was loaded using following functions, including image transform and label decoding. Note that, the input to the DDPM and the GAN was slightly different. To enhance the diversity in GAN, condition with number `label_vector[label_index] = label_index+1` replaced the one-hot condition `label_vector_class[label_index] = 1` in GAN.

```
import pandas as pd
from PIL import Image
from torch.utils import data
import torchvision.transforms as transforms
import os
import json
import torch
import numpy as np

class MultiLabelDataset(data.Dataset):
    def __init__(self, root, label_mapping, json_labels):
        self.root = root
        self.label_mapping = label_mapping
        self.json_labels = json_labels
        self.img_name = list(self.json_labels.keys())
        self.transform = transforms.Compose([
                                transforms.Resize((64, 64)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5)),])
        print("> Found %d images..." % len(self.img_name))

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

    def __getitem__(self, index):
        path = os.path.join(self.root, 'train', self.img_name[index])
        img = Image.open(path).convert('RGB')

        # 多標籤處理
        img_file = self.img_name[index]
        labels = self.json_labels.get(img_file, [])

        # 將多標籤轉換為多熱編碼
        label_vector = np.zeros(len(self.label_mapping), dtype=np.float32)
        label_vector_class = np.zeros(len(self.label_mapping), dtype=np.float32)
        for label in labels:
            label_index = self.label_mapping.get(label)
            if label_index is not None:
                label_vector[label_index] = label_index+1
                label_vector_class[label_index] = 1

        img = self.transform(img)
        label_vector = torch.tensor(label_vector)
        label_vector_class = torch.tensor(label_vector_class)
        # print(label_vector)
        return img, label_vector, label_vector_class
```

B. DDPM


$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_0$



```
Algorithm 1 Training
1: repeat
2:     $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:     $t \sim \mathrm{Uniform}(\{1, \dots, T\})$
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 $t = T, \dots, 1$ 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 $\mathbf{x}_0$
```

The materials of diffusion model were adopted from the diffuser (https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers/diffusers_intro.ipynb)

Unet: UNet2DModel (provided by diffuser)

The model architecture includes 6 times of down sampling and up-sampling, aiming to handle the complicated condition. In addition to the regular down sampling and up-sampling blocks, ResNet down sampling and up-sampling block with spatial self-attention was applied to enhance feature extraction and representation. Furthermore, I additionally applied linear function to transform the condition labels into class embeddings to enhance the complexity of the condition `model.class_embedding = nn.Linear(24 ,512)`.

UNet2DModel settings

```
model = UNet2DModel(
        sample_size = 64,
        in_channels = 3,
        out_channels = 3,
        layers_per_block = 2,
        # class_embed_type = None,
        class_embed_type="linear",
        num_class_embeds=24,
        #num_class_embeds = 2325, #C (24, 3) + 1
        block_out_channels = (128, 128, 256, 256, 512, 512),
        down_block_types=(
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",
        "DownBlock2D",
        ),
        up_block_types=(
            "UpBlock2D",
            "AttnUpBlock2D",
            "UpBlock2D",
            "UpBlock2D",
            "UpBlock2D",
            "UpBlock2D",
        ),
    )
```

Training Procedure

Following DDPM training process listed above, I first prepare random t, data and Gaussian noise, and by using the pre-defined $\sqrt{\bar{\alpha}_t}$ and $\sqrt{1-\bar{\alpha}_t}$, it will be also to compute noisy img(xt) in step t. Then the label, random timestep, and the noisy img(xt) will be applied to the diffuser model. Finally, the loss can be MSE computed between the predicted noise (ϵ_θ) and the ground truth noise (ϵ) and be backpropagated to the model.

```
5:  Take gradient descent step on
        $\nabla_\theta \left\| \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t) \right\|^2$
```

```python
def train(args, model, device, train_loader, optimizer, epoch, lr_scheduler,
accelerator):
    model.train()
    for batch_idx, (data, cond) in enumerate(train_loader):
        #print(data.shape[0])
        data, cond = data.to(device,dtype=torch.float32), cond.to(device)
        cond = cond.squeeze()
        optimizer.zero_grad()
        # select t
        rand_t = torch.tensor([random.randint(1, timestep) for i in
range(data.shape[0])])
        # select noise
        noise = torch.randn(data.shape[0], 3, 64, 64)
        xt = compute_xt(args, data, rand_t, noise)
        output = model(sample = xt.to(args.device), timestep =
rand_t.to(args.device), class_labels  = cond.to(torch.float32).to(args.device))
        loss = nn.MSELoss()(output.sample.to(args.device), noise.to(args.device))
        accelerator.backward(loss)
        lr_scheduler.step()
        optimizer.step()
```

Sampling Procedure

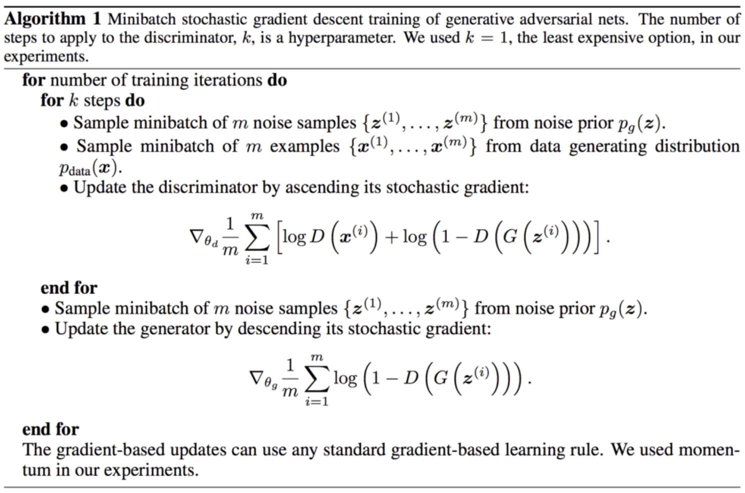
To convert conditional embeddings to embeddings, we start from the maximum timestep and initialize with Gaussian noise. We then iteratively input x_t, the timestep, and class embeddings to denoise the image and compute x_{t-1}. Finally, to check the quality of the diffusion model, the provided evaluator is used to assess the performance.

```python
def sample(model, device, test_loader, args, filename):
    # denormalize
    transform=transforms.Compose([
            transforms.Normalize((0, 0, 0), (1/0.5, 1/0.5, 1/0.5)),
            transforms.Normalize((-0.5, -0.5, -0.5), (1, 1, 1)),
        ])
    model.eval()
    xt = torch.randn(args.test_batch, 3, 64, 64)
    with torch.no_grad():
        for batch_idx, (img, cond) in enumerate(test_loader):
            cond = cond.to(device)
            # transform one-hot to embed's input
            # cond = transform_code(cond_onehot)
            cond = cond.squeeze()
            # print(cond)
            # print(cond, cond.shape)
            # print(cond_onehot, cond_onehot.shape)
            for t in range(timestep, 0, -1):
                # pred noise
                output = model(sample = xt.to(args.device), timestep = t,
class_labels = cond.to(torch.float32).to(args.device))
                # compute xt-1
                xt = compute_prev_x(xt.to(args.device), t,
output.sample.to(args.device), args)

            # evaluate
            from evaluator import evaluation_model
            evaluate = evaluation_model()
            acc = evaluate.eval(xt.to(args.device), cond.to(args.device))
            torch.save(xt, f = filename+".pt")
            print("Test Result:", acc)
```

Other settings

Total time step was selected at 1200, and the noise schedule was linear schedule. The batch size was 20, and the total epoch was set at 100 with the learning rate of 0.00005 using AdamW optimizer.

C. GAN



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].$$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
        $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

Conditional GAN model was trained by applying a loss of modified auxiliary classifier GANs[4, 5], which is $\min_G \mathrm{JS}(P_X \| Q_X) + \lambda(\mathrm{KL}(Q_{X,Y} \| P_{X,Y}))$, the classifier loss timing lamda of 0.2 is added to the training of the generator. The overall code was modified from DCGAN code provided in https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Generator

The generator consists of linear projection to convert the label to embeddings. And the random noise of size (batch, 512) and the condition embedding (batch, 512) will be concatenated, followed by 5 ConvTranspose and a tanh function to generate fake image.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        nz = 512+512 # latent vector + noise 100 dimension
        ngf = 64
        nc = 3 # number of channels
        self.linear = nn.Sequential(nn.Linear(24, 512), nn.LeakyReLU())
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )
    def forward(self, noise, labels):
        labels = self.linear(labels).view(labels.size(0), 512, 1, 1)
        x = torch.cat([noise, labels], dim=1)
        return self.main(x)
```

Discriminator

The discriminator also consists of linear projection to convert the label to embeddings. However, due to the requirement of the concatenation, the condition embedding now has a size of (batch, 4096) And after label and image were concatenated, it will go through 5 Conv2d and a sigmoid function to judge the realness of the image

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        ndf  = 64
        nc = 4 # number of channels
        # self.ngpu = ngpu
        self.linear = nn.Sequential(nn.Linear(24, 64*64), nn.LeakyReLU())
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input, label):
        label = self.linear(label).view(label.size(0), 1, 64, 64)
        output = torch.concat([input, label], dim=1)
        return self.main(output)
```

Training Process

The training process starts with the discriminating the real image, followed by generating the fake image, discriminating and classifying the fake image. And finally updates the generator and discriminator at the end of each batch.

```
for a, (real_images, class_labels, class_labels_class) in
enumerate(tqdm(train_dataloader, desc=f'Epoch {epoch}/{num_epochs}')):
            real_images = real_images.to(device)
            class_labels = class_labels.to(device)
            class_labels_class = class_labels_class.to(device)

            # Update Discriminator network: maximize log(D(x)) + log(1 - D(G(z)))
            # Discriminate real image
            b_size = real_images.size(0)
            true_label_update = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
            output = Dis(real_images, class_labels).view(-1)
            Dis_real_loss = criterion(output, true_label_update)
            D_x = output.mean().item()
            # Discriminate fake images
            ## 1. Generate fake images
            noise_condition = 512
            noise = torch.randn(b_size, noise_condition, 1, 1, device=device)-1
            fake_images = Gen(noise, class_labels)
            ## 2. discriminate fake images
            fake_label_update = torch.full((b_size,), fake_label, dtype=torch.float,
device=device)
            output = Dis(fake_images.detach(), class_labels).view(-1)
            Dis_fake_loss = criterion(output, fake_label_update)
            D_G_z1 = output.mean().item()
            errD = Dis_real_loss + Dis_fake_loss
            DISCRIMINATOR_LOSS += errD.item()

            # # Update Generator network: maximize log(D(G(z))) + loss (classfier)
            # ## Update Classifier on fake images
            with torch.no_grad():
                class_out_fake = Classif(fake_images.detach())
                class_loss_fake = criterion_class(class_out_fake, class_labels_class)
                out, onehot_labels = class_out_fake.cpu(), class_labels_class.cpu()
                acc = 0
                total = 0
                for i in range(b_size):
                    k = int(onehot_labels[i].sum().item())
                    total += k
                    _, outi = out[i].topk(k)
                    _, li = onehot_labels[i].topk(k)
                    for j in outi:
                        if j in li:
                            acc += 1
                CLASSIFY_ACC += acc / total
            # Update Discriminator
            if update: #
                Dis.train()
                optimizerDis.zero_grad()
                errD.backward(retain_graph=True)
                optimizerDis.step()

            ## Update Classifier on fake images
            Gen.train()
            optimizerGen.zero_grad()
            output = Dis(fake_images, class_labels).view(-1)
            Generator_loss = 0.8 * criterion(output, true_label_update) + lamda *
class_loss_fake
            GENERATOR_LOSS += Generator_loss.item()
            CLASSIFY_LOSS += class_loss_fake.item()
            Generator_loss.backward()
            optimizerGen.step()
```

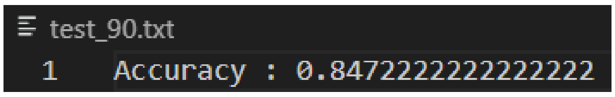
Other settings

The batch size was 512, and the total epoch was set at 3000 with the learning rate of 0.0002 using Adam optimizer.

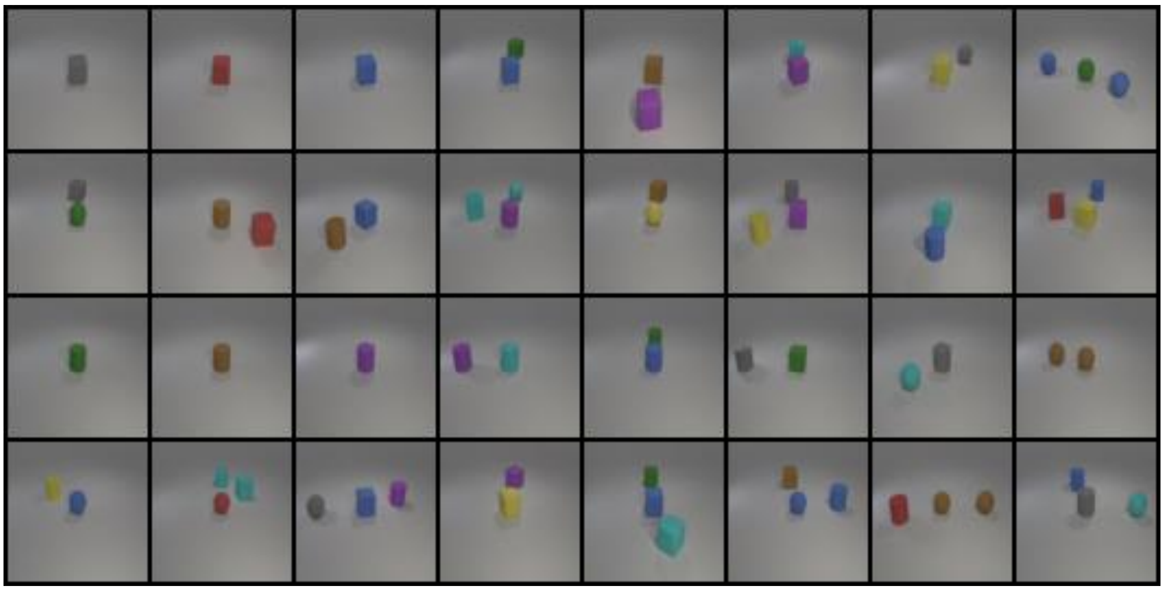
3. Results and discussion

A. Show your synthetic image grids (total 12%: 3% * 2 models * 2 testing data) and a denoising process image (3%)

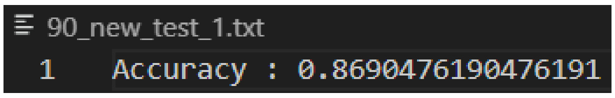
(1) **DDPM (epoch 90)** Test Image, the accuracy of object evaluation is 0.847



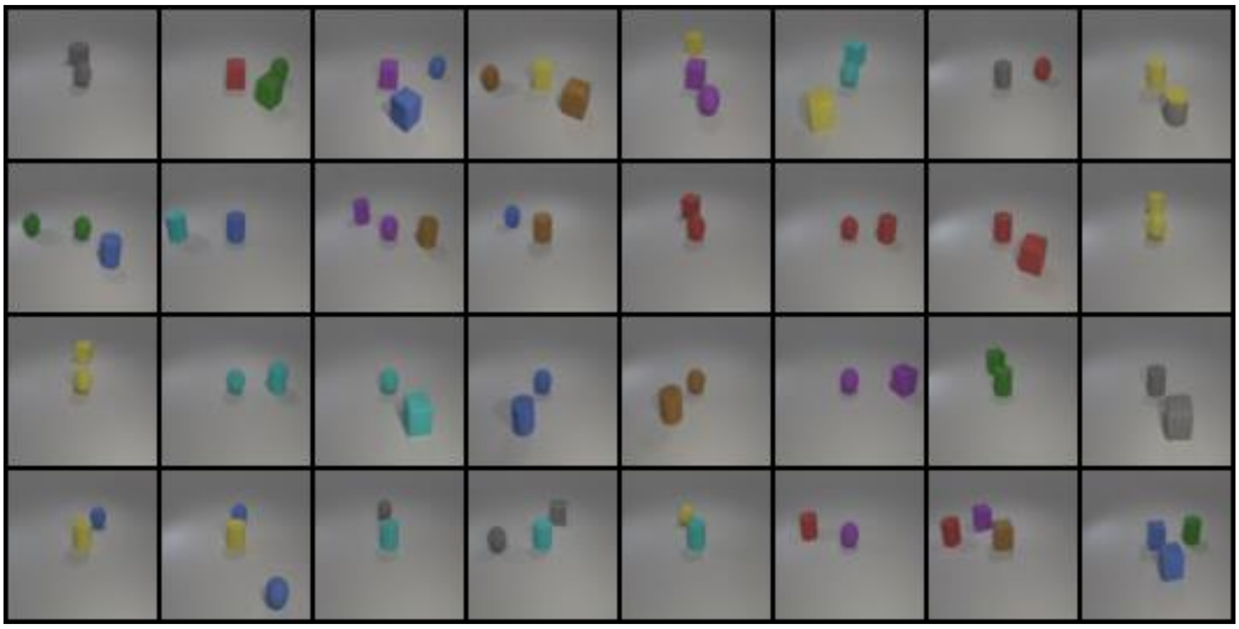
```
test_90.txt
1    Accuracy : 0.8472222222222222
```



(2) **DDPM (epoch 90)** New Test Image, the accuracy of object evaluation is 0.869

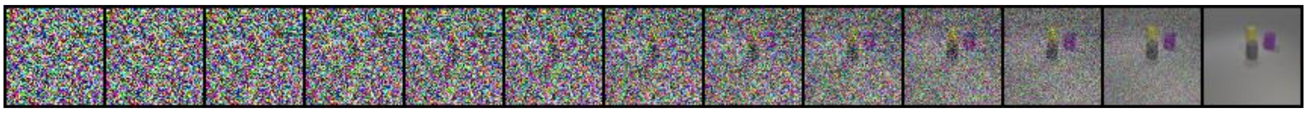


```
90_new_test_1.txt
1    Accuracy : 0.8690476190476191
```

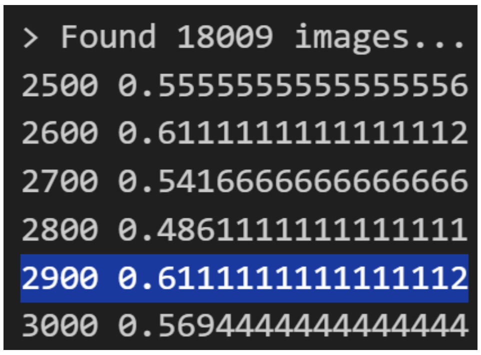


(3) **The denoising process of the DDPM**

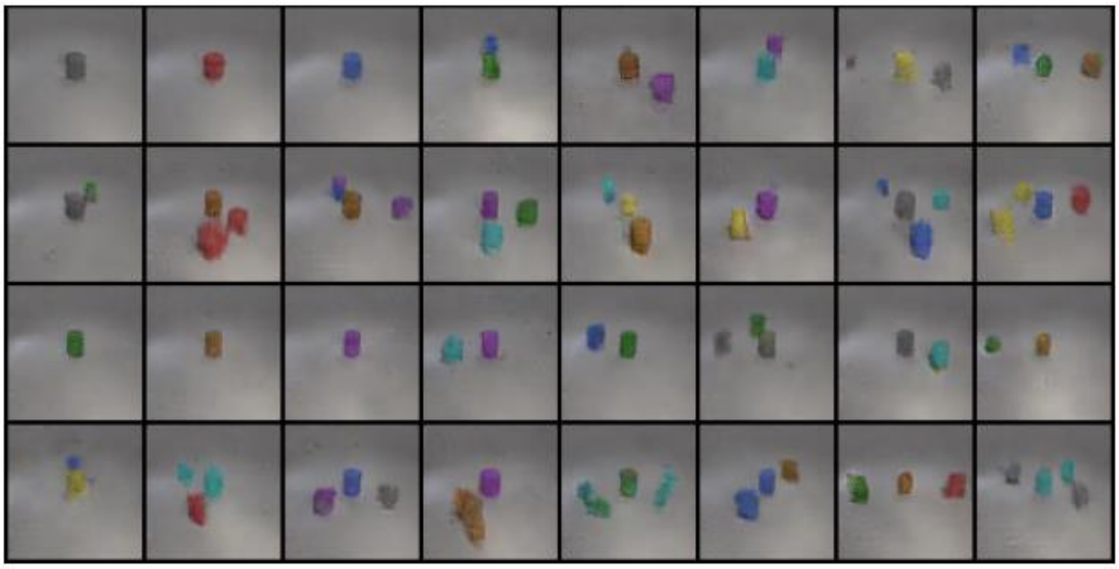
The total timestep is 1200, and the figure shows in step of 100. The image is generated through denoising process.



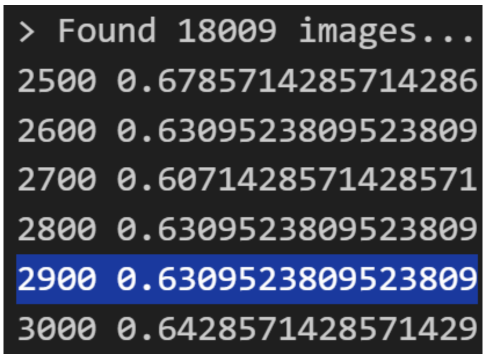
(4) **GAN Test Image (epoch = 2900)**, the accuracy of object evaluation is 0.61



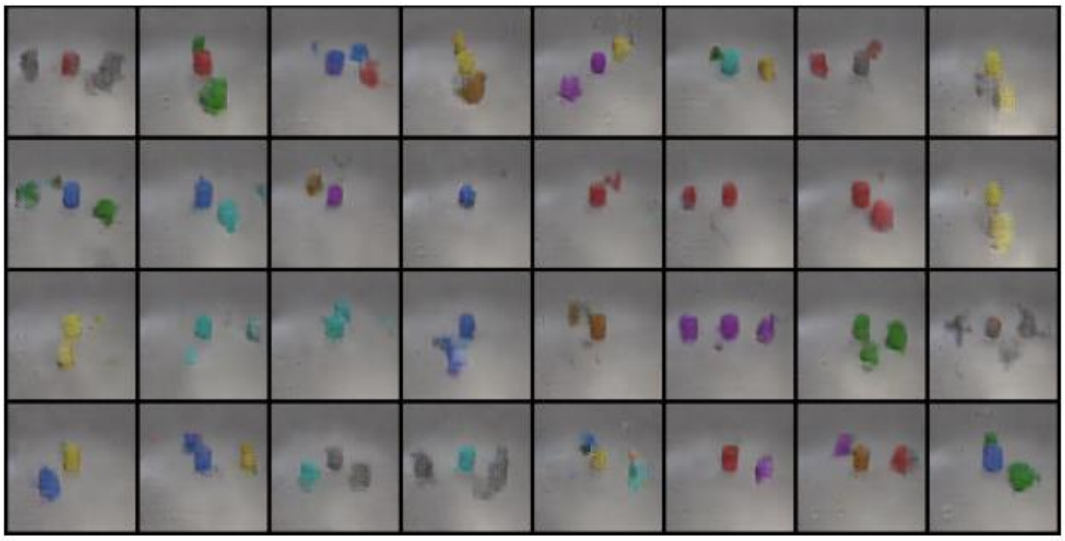
```
> Found 18009 images...
2500 0.5555555555555556
2600 0.6111111111111112
2700 0.5416666666666666
2800 0.4861111111111111
2900 0.6111111111111112
3000 0.5694444444444444
```



(5) **GAN New Test Image (epoch = 2900)**, the accuracy of object evaluation is 0.631



```
> Found 18009 images...
2500 0.6785714285714286
2600 0.6309523809523809
2700 0.6071428571428571
2800 0.6309523809523809
2900 0.6309523809523809
3000 0.6428571428571429
```



B. Compare the advantages and disadvantages of the GAN and DDPM models (10%)

Generative Adversarial Networks (GANs) and Denoising Diffusion Probabilistic Models (DDPMs) are two kinds of generative model, and both of them were widely used and modified these days. Traditionally, GANs are known for their high-quality outputs and efficiency in generating samples but suffer from training instability and sensitivity to hyperparameters. On the other hand, DDPMs offer stable training and better mode coverage but are slower in both training and generation, and involve more complex theoretical frameworks. In my practice, although DDPM costs more time, conditional DDPM outperforms the conditional GAN even without classifier guidance in the sampling stage. Furthermore, one important thing in GAN training is that, the training process was extremely unstable as shown in figure below. Although I've tried several times (at least 5 times) to modify the hyperparameter and re-trained the model, the loss of generator still kept increasing. It often leads to generate a total fake figure with a small overfitting to the classifier. Therefore, it's challenging to train GAN for multi-label object image.

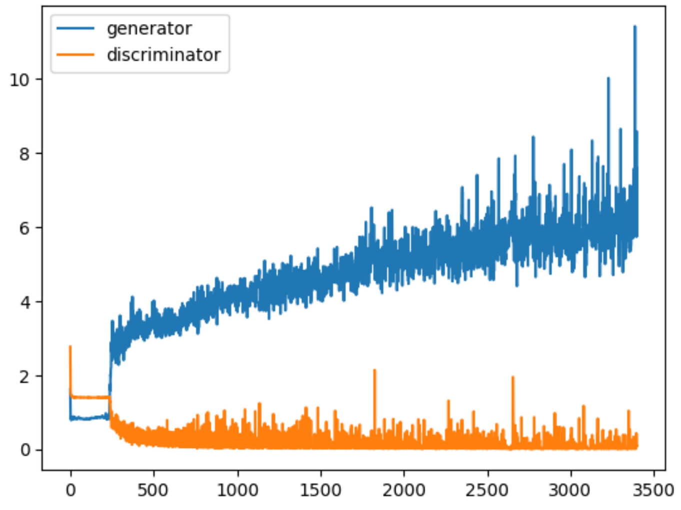


Fig. Loss of the generator and the discriminator in GAN

C. Discussion of your extra implementations or experiments (10%)

Experiment on class embeddings

In this lab, I also test different complexity (amount of dimension) of the class embedding in the training of conditional GAN. As the result shown below, we can see if the model architecture remains the same except for the dimension amount of the class embedding, the classification result shows slightly different in the timing of increasing, and the ceiling performance of the model.

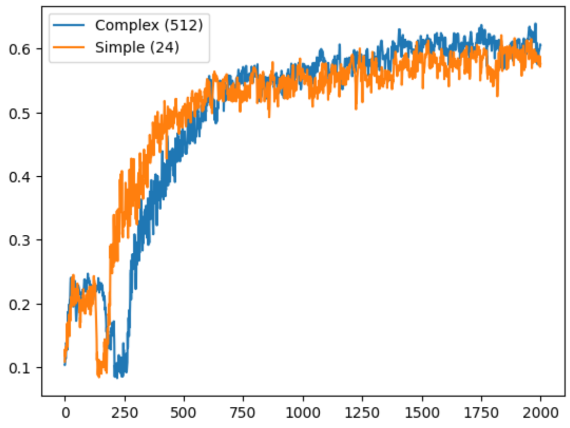
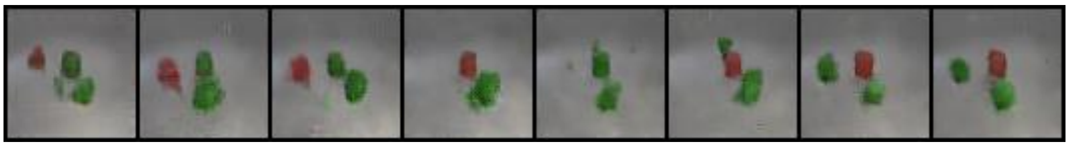


Fig. Classification Accuracy of Fake Figures during Training Epochs

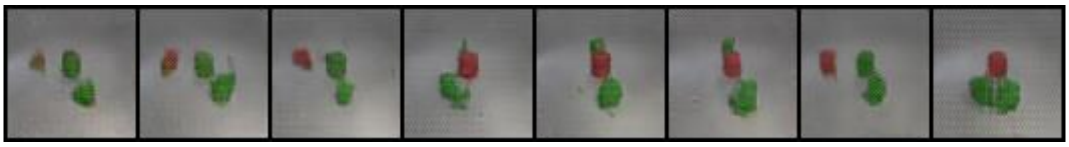
GAN Training with Fixed Lamda or Adaptive Lamda of auxiliary classifier loss

In the GAN model training, I anticipated the Lamda of the auxiliary classifier loss influencing the generated image a lot. Hence, here I showed the differences in generated images by using different approaches. Although the epoch 2700 in the adaptive lamda situation show better visualization, there seems to have small differences. Hence, if the lamda value should have a schedule controlling it, we still need more experiment to find out.

Lamda increase from 0.2 to 0.5 (epoch 1900-2700)



Lamda fixed at 0.2 (epoch 1900-2700)



4. References

1 J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems,* vol. 33, pp. 6840-6851, 2020.

2 I. Goodfellow *et al.*, "Generative adversarial networks," *Communications of the ACM,* vol. 63, no. 11, pp. 139-144, 2020.

3 J. Johnson, B. Hariharan, L. Van Der Maaten, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick, "Clevr: A diagnostic dataset for compositional language and elementary visual reasoning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2901-2910.

4 A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," in *International conference on machine learning*, 2017: PMLR, pp. 2642-2651.

5 L. Hou, Q. Cao, H. Shen, S. Pan, X. Li, and X. Cheng, "Conditional gans with auxiliary discriminative classifier," in *International Conference on Machine Learning*, 2022: PMLR, pp. 8888-8902.