Generative Models
Lab Report # 6

By
312581020
許瀚丰

Deep Learning
Spring 2024
Date Submitted: June 13, 2024

Contents

Listings

List of Tables

1 Introduction

1.1 Problem Statement

在本次實驗中，我們要嘗試設計Generative Adversarial Network與Diffusion Model，並透過multi-label的condition來完成Image Generation。而在evaluation的部分，會透過一個pretrained在此任務上的resnet18來計算準確率，而本次實驗的目標就是期望讓此準確率越高越好。

2 Implementation Details

在本次作業中，我共嘗試了以下六種不同的Approaches，分別為DCGAN、ACGAN、SAGAN、DCGAN + evaluator discriminator、SAGAN + evaluator discriminator、DDPM。

2.1 GAN

2.1.1 DCGAN

在DCGAN的實作中，我參考了Pytorch所提供的tutorial(https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)，而為了要加上condition進去，我將condition轉換成一組one-hot vector，並透過一個簡單的linear後產生出一組embedding，並跟輸入的noise concatenate後一起送進generator產生圖片，如程式碼1所示。

```
class Generator(nn.Module):
    def __init__(self, nz, ngf, nc, n_classes):
        super(Generator, self).__init__()
        self.nz = nz
        self.class_embedding = nn.Sequential(
            nn.Linear(n_classes, nz),
            nn.LeakyReLU(0.2, inplace=True)
        )
        def dconv_bn_relu(in_dim, out_dim, kernel_size, stride, padding):
            return [
                nn.ConvTranspose2d(in_dim, out_dim, kernel_size, stride, padding, bias=False),
                nn.BatchNorm2d(out_dim),
                nn.ReLU(True)
```

```python
            ]

        self.model = nn.Sequential(
            *dconv_bn_relu(2 * nz, ngf * 8, 4, 1, 0),
            *dconv_bn_relu(ngf * 8, ngf * 4, 4, 2, 1),
            *dconv_bn_relu(ngf * 4, ngf * 2, 4, 2, 1),
            *dconv_bn_relu(ngf * 2, ngf, 4, 2, 1),
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )
    def forward(self, x, labels):
        x = x.view(-1, self.nz, 1, 1)
        class_embedding = self.class_embedding(labels).reshape(-1, self.nz, 1, 1)
        x = torch.cat((x, class_embedding), dim=1)
        return self.model(x)
```

Listing 1: DCGAN Generator

而對於discriminator，為了將condition加入，我使用了與generator加入condition類似的作法，由於discriminator原始的輸入是一張$3*64*64$的影像，我透過將condition經過一個linear產生出$64*64$大小的embedding，並將其沿著channels concatenate，因此輸入就變為$4*64*64$，如程式碼2所示。

```python
class Discriminator(nn.Module):
    def __init__(self, nc, ndf, n_classes):
        super(Discriminator, self).__init__()
        self.class_embedding = nn.Linear(n_classes, 64 * 64)
        self.model = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc + 1, 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),
        )
    def forward(self, x, labels):
        labels = labels.float()
        class_embedding = self.class_embedding(labels).view(-1, 1, 64, 64)
        x = torch.cat((x, class_embedding), dim=1)
        return self.model(x)
```

Listing 2: DCGAN Discriminator

Loss Function的部份，我是使用Hinge Loss的作法，其作法是對於discriminator來說，當送進去的是真實影像，只有$D(x) < 1$才會更新。而由Generator產生的Fake Image，也只有$D(x) > 1$才會有更新。而對於generator來說，由於其是希望discriminator給的分數越高越好，因此其目標就是去最大化D(x)，此x是由generator產生的Fake Image，由於在Pytorch中對於預設是最小化，因此就將其加上負號，如程式碼3所示。

```
def adversarial_loss_d(r_logit, f_logit):
    r_loss = torch.max(1 - r_logit, torch.zeros_like(r_logit)).mean()
    f_loss = torch.max(1 + f_logit, torch.zeros_like(f_logit)).mean()
    return r_loss, f_loss

def adversarial_loss_g(f_logit):
    f_loss = - f_logit.mean()
    return f_loss
```

Listing 3: Hinge Loss

在training時，首先會先sample一組noise，並將其與condition一起輸入generator產生Fake Image。接著會先訓練discriminator，希望能夠判斷影像是真實的還是由generator產生的。最後再去訓練generator，希望能夠讓discriminator預測的分數越高越好。如程式碼4所示。

```
for i, (imgs, labels) in trange:
    batch_size = imgs.shape[0]
    imgs = imgs.to(device)
    labels = labels.to(device)

    # generate fake images
    z = torch.randn(batch_size, opt.latent_dim).to(device)
```

```
    gen_imgs = generator(z, labels)

    # ---------------------
    #  Train Discriminator
    # ---------------------

    real_pred = discriminator(imgs, labels)
    fake_pred = discriminator(gen_imgs.detach(), labels)
    d_real_loss, d_fake_loss = adversarial_loss_d(real_pred, fake_pred)
    d_loss = d_real_loss + d_fake_loss

    optimizer_D.zero_grad()
    d_loss.backward()
    optimizer_D.step()


    # -----------------
    #  Train Generator
    # -----------------

    validity = discriminator(gen_imgs, labels)
    g_loss = adversarial_loss_g(validity)

    optimizer_G.zero_grad()
    g_loss.backward()
    optimizer_G.step()

    gloss += (g_loss.item())
    dloss += d_real_loss.item() + d_fake_loss.item()
    acc += eval_model.eval(gen_imgs.detach(), labels.detach())

    trange.set_postfix({"epoch":"{}".format(epoch),"g_loss":"{0:.5f}".format(gloss / (
    i + 1)), "d_loss":"{0:.5f}".format(dloss / (i + 1)), "acc":"{0:.5f}".format(acc / (
    i + 1))})
```

Listing 4: DCGAN Training

而在testing時，只需要保留generator，而實際要做的是就是將noise與condition一起輸入generator並產生影像即可。如程式碼5所示。

```
def test(generator, epoch, eval_model):
    generator.eval()
    test_dataset = iclevrDataset(opt.data_dir, "test")
    new_test_dataset = iclevrDataset(opt.data_dir, "new_test")
```

```python
    test_dataloader = DataLoader(test_dataset, batch_size=32, num_workers=opt.n_cpu)
    new_test_dataloader = DataLoader(new_test_dataset, batch_size=32, num_workers=opt.n_cpu)
    test_acc, new_test_acc = 0, 0
    with torch.no_grad():
        labels = next(iter(test_dataloader))
        z = torch.randn(32, opt.latent_dim).to(device)
        labels = labels.to(device)
        gen_imgs = generator(z, labels)
        acc = eval_model.eval(gen_imgs, labels)
        test_acc = acc
        path = os.path.join(opt.test_dir, '{}_test_{:.4f}.png'.format(epoch, acc))
        gen_imgs = (gen_imgs+1)/2
        save_image(gen_imgs, path, nrow=8)

        labels = next(iter(new_test_dataloader))
        z = torch.randn(32, opt.latent_dim).to(device)
        labels = labels.to(device)
        gen_imgs = generator(z, labels)
        acc = eval_model.eval(gen_imgs, labels)
        new_test_acc = acc
        path = os.path.join(opt.test_dir, '{}_new_test_{:.4f}.png'.format(epoch, acc))
        gen_imgs = (gen_imgs+1)/2
        save_image(gen_imgs, path, nrow=8)
    return test_acc, new_test_acc
```

Listing 5: DCGAN Testing

2.1.2 ACGAN

對於ACGAN的部分，此方法與DCGAN唯一的差別是在discriminator的部分，不需要將generator產生影像與label結合送入discriminator判斷好壞。而是透過在discriminator上增加一個classifier，嘗試透過同時判斷輸入結果的好壞與預測輸入的condition為何來去引導generator的產生圖片，如程式碼6所示，此discriminator是由SAGAN的discriminator修改而成，其中最後的adv_layer是為了判斷產生結果的好壞，而aux_layer則是前面所提的classifier。

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.ndf = 16
```

```python
        self.ncf = 100
        def discriminator_block(in_dim, out_dim, kernel_size, stride, padding, bn=True
    ):
            block = []
            block.append(spectral_norm(nn.Conv2d(in_dim, out_dim, kernel_size, stride,
     padding, bias=False)))
            if bn: block.append(nn.BatchNorm2d(out_dim))
            block.append(nn.LeakyReLU(0.2, inplace=True))
            block.append(nn.Dropout2d(0.25))
            return block

        self.conv_blocks = nn.Sequential(
            *discriminator_block(3, self.ndf, 3, 2, 1, bn=False),
            *discriminator_block(self.ndf, self.ndf * 2, 3, 1, 0),
            *discriminator_block(self.ndf * 2, self.ndf * 4, 3, 2, 1),
            *discriminator_block(self.ndf * 4, self.ndf * 8, 3, 1, 0),
            *discriminator_block(self.ndf * 8, self.ndf * 16, 3, 2, 1),
            *discriminator_block(self.ndf * 16, self.ndf * 32, 3, 1, 0),
        )

        self.adv_layer = nn.Sequential(
            nn.Linear(5 * 5 * self.ndf * 32, 1),
        )
        self.aux_layer = nn.Sequential(
            nn.Linear(5 * 5 * self.ndf * 32, 24),
            nn.Sigmoid()
        )

    def forward(self, x):
        out = self.conv_blocks(x)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)
        label = self.aux_layer(out)

        return validity, label
```

Listing 6: ACGAN discriminator

對於training的部份，由於其多了需要預測生成的影像是何種condition，因此需要多一個auxi_loss來計算(此Loss是使用BCE Loss計算而來)，而此Loss是可以透過調整其權重來(在此以100為例)來增加condition對於訓練的影響，如程式碼7所示。

```python
for i, (imgs, labels) in trange:
        batch_size = imgs.shape[0]
```

```
        imgs = imgs.to(device)
        labels = labels.to(device)

        # ---------------------
        #  Train Discriminator
        # ---------------------


        z = torch.randn(batch_size, opt.latent_dim).to(device)
        gen_imgs = generator(z, labels)

        real_pred, real_label = discriminator(imgs)
        real_auxi_loss = auxiliary_loss(real_label, labels)

        fake_pred, fake_label = discriminator(gen_imgs.detach())
        fake_auxi_loss = auxiliary_loss(fake_label, labels)

        d_fake_loss, d_real_loss = adversarial_loss_d(real_pred, fake_pred)

        d_loss = d_real_loss + d_fake_loss + 100 * (real_auxi_loss + fake_auxi_loss)

        optimizer_D.zero_grad()
        d_loss.backward()
        optimizer_D.step()

        # -----------------
        #  Train Generator
        # -----------------

        validity, pred_label = discriminator(gen_imgs)
        g_loss = adversarial_loss_g(validity) + 100 * auxiliary_loss(pred_label,
    labels)

        optimizer_G.zero_grad()
        g_loss.backward()
        optimizer_G.step()

        gloss += (g_loss.item())
        dloss += d_real_loss.item() + d_fake_loss.item()
        acc += eval_model.eval(gen_imgs.detach(), labels.detach())

        trange.set_postfix({"epoch":"{}".format(epoch),"g_loss":"{0:.5f}".format(gloss
     / (i + 1)), "d_loss":"{0:.5f}".format(dloss / (i + 1)), "acc":"{0:.5f}".format(acc
```

```
    / (i + 1))})
```

Listing 7: ACGAN Training

2.1.3 SAGAN

在本實驗中，我也嘗試使用Self-Attention GAN(SAGAN)的方式來完成，由於我參考的SAGAN程式碼(https://github.com/heykeetae/Self-Attention-GAN)並沒有提供conditional Image Generation，因此我嘗試類似於DCGAN的作法，在generator與discriminator中透過將condition經過一些Linear後接在輸入的部分，如程式碼8所示。

```
class Generator(nn.Module):
    """Generator."""

    def __init__(self, batch_size, image_size=64, z_dim=100, conv_dim=64, class_num
    =24):
        super(Generator, self).__init__()
        self.imsize = image_size
        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        repeat_num = int(np.log2(self.imsize)) - 3
        mult = 2 ** repeat_num # 8
        layer1.append(SpectralNorm(nn.ConvTranspose2d(z_dim + z_dim, conv_dim * mult,
    4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult

        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4,
    2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)

        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4,
    2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
```

```
        layer3.append(nn.ReLU())

        if self.imsize == 64:
            layer4 = []
            curr_dim = int(curr_dim / 2)
            layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
            layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
            layer4.append(nn.ReLU())
            self.l4 = nn.Sequential(*layer4)
            curr_dim = int(curr_dim / 2)

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn( 128, 'relu')
        self.attn2 = Self_Attn( 64,  'relu')

        self.class_embed = nn.Sequential(
            nn.Linear(class_num, z_dim),
            nn.ReLU(True),
        )

    def forward(self, z, label):
        z = z.view(z.size(0), z.size(1), 1, 1)
        label = self.class_embed(label).view(z.size(0), z.size(1), 1, 1)
        z = torch.cat([z, label], 1)
        out=self.l1(z)
        out=self.l2(out)
        out=self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out=self.last(out)

        return out, p1, p2


class Discriminator(nn.Module):
```

```
    """Discriminator, Auxiliary Classifier."""

    def __init__(self, batch_size=64, image_size=64, conv_dim=64, class_num=24):
        super(Discriminator, self).__init__()
        self.imsize = image_size
        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        layer1.append(SpectralNorm(nn.Conv2d(3 + 1, conv_dim, 4, 2, 1)))
        layer1.append(nn.LeakyReLU(0.1))

        curr_dim = conv_dim

        layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer2.append(nn.LeakyReLU(0.1))
        curr_dim = curr_dim * 2

        layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer3.append(nn.LeakyReLU(0.1))
        curr_dim = curr_dim * 2

        if self.imsize == 64:
            layer4 = []
            layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
            layer4.append(nn.LeakyReLU(0.1))
            self.l4 = nn.Sequential(*layer4)
            curr_dim = curr_dim*2
        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.Conv2d(curr_dim, 1, 4))
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(256, 'relu')
        self.attn2 = Self_Attn(512, 'relu')
        self.label_embed = nn.Linear(class_num, 64 * 64)

    def forward(self, x, label):
        label = self.label_embed(label).view(x.size(0), 1, x.size(2), x.size(3))
        x = torch.cat([x, label], 1)
```

```python
        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out1=self.last(out)
        return out1.squeeze(), p1, p2
```

Listing 8: SAGAN

2.1.4 Pretrained evaluator Discriminator

由於助教有提供一個用於判斷accuracy的模型，因此我也嘗試直接在DCGAN與SAGAN中嘗試加上Pretrained evaluator作為另一個discriminator，用於幫助模型使其產生的結果能夠有更高的accuracy。而在實作上(以DCGAN為例，SAGAN也是用相同方法)，與ACGAN的作法有些類似，只是將判斷影像的好壞與用於預測label的部分分別使用不同的discriminator來產生，evaluator discriminator如程式碼9所示。

```python
class aux_discriminator(nn.Module, evaluation_model):
    """Discriminator containing the auxiliary classifier."""
    def __init__(self):
        nn.Module.__init__(self)
        evaluation_model.__init__(self)
        self.resnet18.requires_grad_(False)

    def forward(self, x):
        out = self.resnet18(x)
        return out
```

Listing 9: Evaluator Discriminator

而在training時，由於此discriminator不需要訓練，因此我直接將其用於訓練generator的部分，計算產生出的影像gen_imgs，將其經過evaluator discriminator，並透過BCE Loss來計算結果(aux_loss)，並將此loss用於generator的訓練，如程式碼10所示。

```python
for i, (imgs, labels) in trange:

    batch_size = imgs.shape[0]
```

```
    imgs = imgs.to(device)
    labels = labels.to(device)

    z = torch.randn(batch_size, opt.latent_dim).to(device)
    gen_imgs = generator(z, labels)

    # ---------------------
    #  Train Discriminator
    # ---------------------

    real_pred = discriminator(imgs, labels)
    fake_pred = discriminator(gen_imgs.detach(), labels)
    d_real_loss, d_fake_loss = adversarial_loss_d(real_pred, fake_pred)
    d_loss = d_real_loss + d_fake_loss

    optimizer_D.zero_grad()
    d_loss.backward()
    optimizer_D.step()


    # -----------------
    #  Train Generator
    # -----------------

    validity = discriminator(gen_imgs, labels)
    aux_loss = nn.BCELoss()(a_discriminator(gen_imgs), labels)
    g_loss = adversarial_loss_g(validity) +  aux_loss

    optimizer_G.zero_grad()
    g_loss.backward()
    optimizer_G.step()

    gloss += (g_loss.item())
    dloss += d_real_loss.item() + d_fake_loss.item()
    acc += eval_model.eval(gen_imgs.detach(), labels.detach())

    trange.set_postfix({"epoch":"{}".format(epoch),"g_loss":"{0:.5f}".format(gloss / (
    i + 1)), "d_loss":"{0:.5f}".format(dloss / (i + 1)), "acc":"{0:.5f}".format(acc / (
    i + 1))})
```

Listing 10: DCGAN with Evaluator Discriminator

2.2 Diffusion Model

在DDPM的部分，我是使用huggingface所提供的diffusers來實作，此工具能夠快速的建立DDPM的model與denoising所使用的scheduler等。

2.2.1 Model Architecture

在DDPM模型架構的部分，我使用UNet2DModel來建立，而為了將condition加入訓練中，我嘗試使用了一個linear來將condition轉為一個embedding，並在forward時將condition的資訊加入DDPM的訓練中。且由於在Diffusers的實作中，是將class_embedding的shape設定為Unet第一個Block的out_dim的四倍，因此我在實作時，Unet的Downsample的dim是從class_embedding的四分之一開始往上加。而在forward的部分，就是同時將輸入的noise、timesteps、condition一起送進模型中，如程式碼11所示。

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

class conditionalDDPM(nn.Module):
    def __init__(self, num_classes=24, dim=512):
        super().__init__()
        channel = dim // 4
        self.ddpm = UNet2DModel(
            sample_size = 64,
            in_channels = 3,
            out_channels = 3,
            layers_per_block = 2,
            block_out_channels = [channel, channel, channel*2, channel*2, channel*4,
    channel*4],
            down_block_types=["DownBlock2D", "DownBlock2D", "DownBlock2D", "
    DownBlock2D", "AttnDownBlock2D", "DownBlock2D"],
            up_block_types=["UpBlock2D", "AttnUpBlock2D", "UpBlock2D", "UpBlock2D", "
    UpBlock2D", "UpBlock2D"],
            class_embed_type="identity",
        )
        self.class_embedding = nn.Linear(num_classes, dim)

    def forward(self, x, t, label):
        class_embed = self.class_embedding(label)
```

```python
        return self.ddpm(x, t, class_embed).sample
```

Listing 11: DDPM Unet

2.2.2 Training

在DDPM的training中，我所使用的Noise schedule是使用squaredcos_cap_v2，而total timestep是1000。在training loop中，在剛開始會先sample一組與要產生的影像大小的noise與隨機的timesteps，接著就將原始的影像加上noise(noise會依照timesteps的值去調整其大小)。而模型實際要做的是，就是去預測此noise為何，因此loss function的部分就是透過MSE去最小化預測的noise與實際的noise的差距，如程式碼12所示。

```python
def get_random_timesteps(batch_size, total_timesteps, device):
    return torch.randint(0, total_timesteps, (batch_size,)).long().to(device)

def train_one_epoch(epoch, model, optimizer, train_loader, loss_function, noise_scheduler, total_timesteps, device):
    model.train()
    train_loss = []
    progress_bar = tqdm(train_loader, desc=f'Epoch: {epoch}', leave=True)
    for i, (x, label) in enumerate(progress_bar):
        batch_size = x.shape[0]
        x, label = x.to(device), label.to(device)
        noise = torch.randn_like(x)

        timesteps = get_random_timesteps(batch_size, total_timesteps, device)
        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)
        output = model(noisy_x, timesteps, label)

        loss = loss_function(output, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss.append(loss.item())
        progress_bar.set_postfix({'Loss': np.mean(train_loss)})

    return np.mean(train_loss)
```

Listing 12: DDPM Training

2.2.3 Testing

在testing的部分，與training有些類似，首先要先sample一組與要生成的影像大小相同的noise，接著就一步一步(根據timesteps)的去預測根據目前的x與timesteps預測noise為何，再透過noise scheduler的step將影像逐漸去除掉noise。跟Training不同的是，testing是一個step一個step慢慢的去denoise，而Training則是用sample timesteps的方式去預測noise，因此其在testing時產生影像會慢上不少，如程式碼13。

```python
def inference(dataloader, noise_scheduler, timesteps, model, eval_model, save_prefix='test'):
    all_results = []
    acc = []
    progress_bar = tqdm(dataloader)
    for idx, y in enumerate(progress_bar):
        y = y.to(device)
        x = torch.randn(1, 3, 64, 64).to(device)
        denoising_result = []
        for i, t in enumerate(noise_scheduler.timesteps):
            with torch.no_grad():
                residual = model(x, t, y)

            x = noise_scheduler.step(residual, t, x).prev_sample
            if i % (timesteps // 10) == 0:
                denoising_result.append(x.squeeze(0))

        acc.append(eval_model.eval(x, y))
        progress_bar.set_postfix_str(f'image: {idx}, accuracy: {acc[-1]:.4f}')

        denoising_result.append(x.squeeze(0))
        denoising_result = torch.stack(denoising_result)
        row_image = make_grid((denoising_result + 1) / 2, nrow=denoising_result.shape[0], pad_value=0)
        save_image(row_image, f'result/{save_prefix}_{idx}.png')

        all_results.append(x.squeeze(0))
    all_results = torch.stack(all_results)
    all_results = make_grid(all_results, nrow=8)
    save_image((all_results + 1) / 2, f'result/{save_prefix}_result.png')
    return acc
```

Listing 13: DDPM Testing

3 Results and discussion

3.1 Show your synthetic image grids and a denoising process image

3.1.1 DCGAN

DCGAN的實驗結果如1所示，其在test與new_test的準確率分別為**0.7361**與**0.7381**。

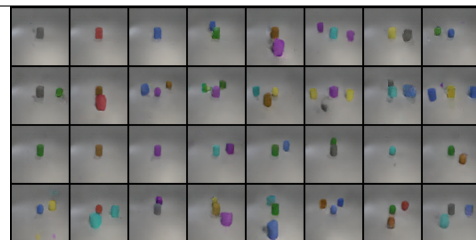
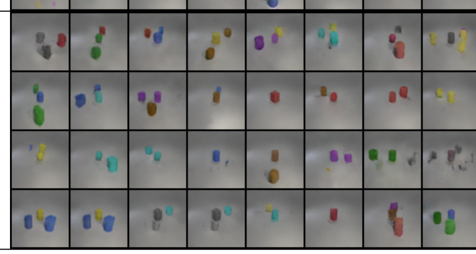
test	
new_test	

Table 1: DCGAN testing result

3.1.2 ACGAN

ACGAN實驗結果如2所示，其在test與new_test的準確率分別為**0.7639**與**0.8095**。

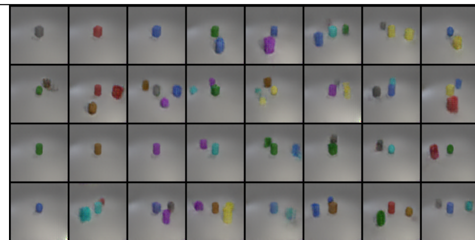
test	
new_test	

Table 2: ACGAN testing result

3.1.3 SAGAN

SAGAN實驗結果如3所示，其在test與new_test的準確率分別為**0.7083**與**0.8095**。

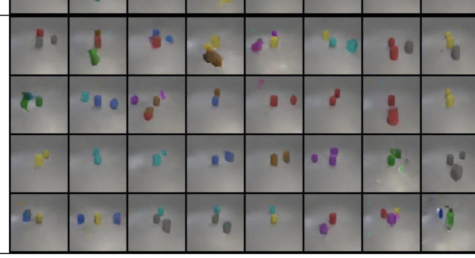
test	
new_test	

Table 3: SAGAN testing result

3.1.4 DCGAN with Evaluator Discriminator

DCGAN with Evaluator Discriminator實驗結果如4所示，其在test與new_test的準確率分別為**0.8889**與**0.9048**。

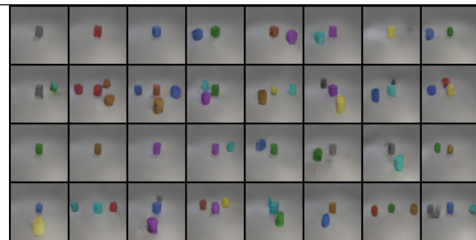
test	
new_test	

Table 4: DCGAN with Evaluator Discriminator testing result

3.1.5 SAGAN with Evaluator Discriminator

DCGAN with Evaluator Discriminator實驗結果如5所示，其在test與new_test的準確率分別為**0.9306**與**0.9167**。

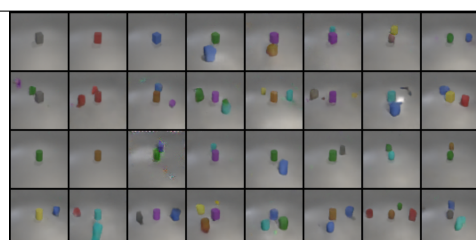
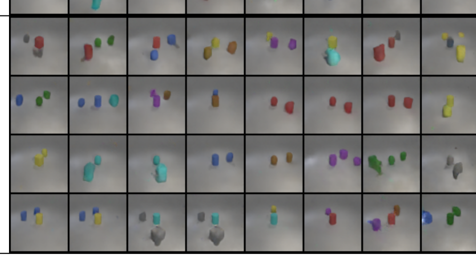
test	
new_test	

Table 5: SAGAN with Evaluator Discriminator testing result

3.1.6 DDPM

DDPM實驗結果如6所示，其在test與new_test的準確率分別為**0.9583**與**0.9375**。

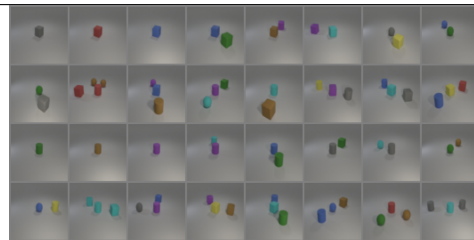
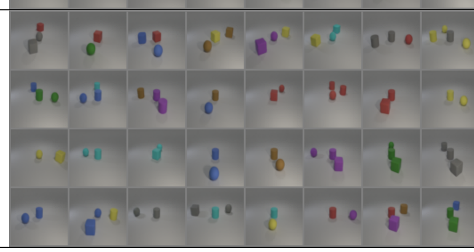
test	
new_test	

Table 6: DDPM testing result

DDPM的denosing process如表7所示，上方為此產生影像所給的condition。

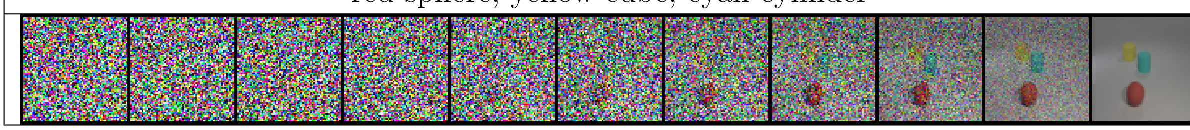
red sphere, yellow cube, cyan cylinder


Table 7: DDPM denoising process

3.1.7 Comparison Result

最終每個方法在test與new test產生的結果，經由evaluator預測的準確率如表8所示。從結果中我們可以觀察到，在GAN的方法中，SAGAN加上evaluator的方法是最好的，test與new test都可以達到0.91以上的結果。而使用DDPM進行訓練產生的結果則更加卓越，兩者皆可以達到0.93以上。

Approach	test accuracy	new test accuracy
DCGAN	0.7361	0.7381
ACGAN	0.7639	0.8095
SAGAN	0.7083	0.8095
DCGAN with evaluator	0.8889	0.9048
SAGAN with evaluator	0.9306	0.9167
DDPM	0.9583	0.9375

Table 8: Result Comparison

3.2 Compare the advantages and disadvantages of the GAN and DDPM models

從這次實驗我發現，雖然GAN與Diffusion的目標都是希望利用condition來生成對應的影像，然而兩者在實作與產生的結果上仍有一些明顯的優缺點。

對於GAN而言，其在訓練上相較於DDPM難訓練許多，不僅有許多超參數需要設定，在訓練時也不太穩定，有時也會遇到train到一半產生的影像就壞掉，並且後續也不會改善。且GAN也較為容易產生mode collapse，同樣的condition生成的影像也都較為類似，缺乏Diversity。然而在inference時，GAN只需要經過Generator就能直接產生，所需花費的時間與DDPM相比少了非常多。

而對於DDPM來說，使用DDPM在訓練上相較於GAN在訓練上更為穩定一些，且生成的影像多樣性較高。但如上面所提的，DDPM在生成影像需要一步一步的慢慢denoising，因此生成影像的時間成本非常高。

3.3 Discussion of your extra implementations or experiments

3.3.1 ACGAN hyperparameter tuning

在ACGAN中，由於其Discriminator需要同時判斷生成的好壞與生成影像預測的condition，因此調整兩者的比例也非常影響產生的結果，因此在本次實驗中，我嘗試去調整生成影像預測的condition的權重，來讓產生的影像能夠有更好的accuracy。實驗結果如表9，從結果中我們可以觀察到將weight調高，生成的accuracy確實會有所提升。另外，我也嘗試將weight設成1000，但其直接導致完全生成不出正常的影像，因此未來或許能夠使用類似scheduling的方式，讓其在初期時專注於產生品質好的影像，後其在逐漸將condition的weight逐步調高，或許就能讓結果更好。

Approach	weight	test accuracy	new test accuracy
ACGAN	100	0.6944	0.7738
ACGAN	200	0.7639	0.7381
ACGAN	300	0.7639	0.8095

Table 9: ACGAN Comparision

3.3.2 DCGAN with eval discriminator hyperparameter tuning

在DCGAN with eval Discriminator中，與ACGAN類似，仍需要去調整eval discriminator的權重。但由於在原始的DCGAN的Discriminator就已經有將condition的資訊加入，因此在訓練時eval discriminator的權重不需要設得太大，且此discriminator是不需要訓練的，因此只需要在訓練Generator時加入即可，實驗結果如表10所示，從實驗結果中可以發現，加上eval discriminator可以讓預測的accuracy大幅上升。

Approach	weight	test accuracy	new test accuracy
DCGAN with eval discriminator	1	0.8889	0.9048
DCGAN with eval discriminator	0.5	0.8611	0.8571
DCGAN with eval discriminator	0.25	0.8056	0.8095
DCGAN	0	0.7361	0.7381

Table 10: DCGAN with eval discriminator Comparision