DLP Lab6 Report
Generative Models

Name: Yu-Cheng Ting
ID: 0812212

1 Introduction

In this lab, the main object is to implement the Generative Adversarial Network and Diffusion Model these two generative models. For GAN, the basic idea is the adversarial training between Generator **G** and a Discriminator **D**. For Generator, it would try its best to generate the good images which can fool the Discriminator and for Discriminator, it would try its best to classify which images are real images, which are fake images. For DDPM, it has two main process. The first process is the forward process which is the diffusion process. The second process is the reverse process which is the denoising process. For forward process, it gradually adds noise to the original image **X** until it is fully a pure Gaussian noise. For reverse process, it gradually denoise the image until it is fully clear. Our task is to synthesize the geometry according to multi-label conditions. For example, if my label is "purple sphere, green cubic, grey cylinder", then I need to generate the corresponding figure.

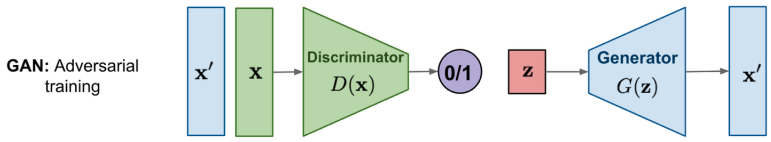


Figure 1: GAN

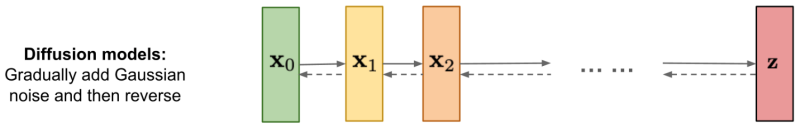
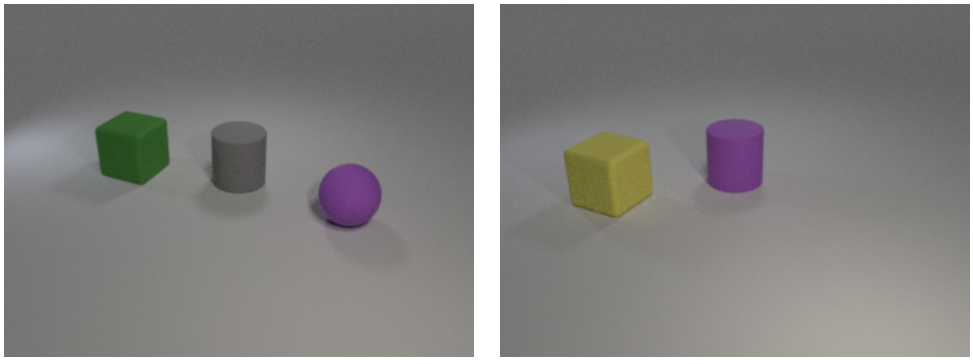


Figure 2: Diffusion Model



(a) "purple sphere, green cubic, grey cylinder" (b) "yellow cubic, purple cylinder

Figure 3: Examples

2 Implementation details

2.1 Label embedding

There are total eight different colors and 3 kinds of shapes for one object. Since our label is multi-hot label, which means we might have at most three objects for our label. So I set the label as a 24-dim vector and there are at most three 1's to represent three object. I then cocatenate with the noise z to feed it into model.

2.2 GAN

For the GAN template, I use the code from github. The following code is the Generator's and Discriminator's architecture.
For both Generator and Discriminator, they are basically combination of CNN and Linear layers. The loss function I use is the Binary-Cross-Entropy Loss. Since the discriminator only has to discriminate whether it is real of fake, so using BCE Loss is the best choice.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = opt.img_size // 4
        self.l1 = nn.Sequential(nn.Linear(opt.latent_dim + 50, 128 * self.
            ↪ init_size ** 2))
        self.lab_emb = nn.Linear(24, 50)

        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, opt.channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )
    def forward(self, z, label):
        label = self.lab_emb(label)
        out = torch.cat((z, label), dim=1)
        out = self.l1(out)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img = self.conv_blocks(out)
        return img
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.lab_emb = nn.Linear(24, 50)
        self.l1 = nn.Linear(12338, 12288)

        def discriminator_block(in_filters, out_filters, bn=True):
            block = [nn.Conv2d(in_filters, out_filters, 3, 2, 1), nn.LeakyReLU
                ↪ (0.2, inplace=True), nn.Dropout2d(0.25)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block
```

```
        self.model = nn.Sequential(
            *discriminator_block(opt.channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )

        ds_size = opt.img_size // 2 ** 4
        self.adv_layer = nn.Sequential(nn.Linear(128 * ds_size ** 2, 1), nn.
            ↪ Sigmoid())

    def forward(self, img, label):
        label = self.lab_emb(label)
        out = torch.cat((img.view(img.size(0), -1), label), dim=1)
        out = self.l1(out).view(img.size(0), 3, 64, 64)
        out = self.model(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)

        return validity
```

```
for epoch in range(self.epoch):
    self.G.train()
    self.D.train()
    for x_, y_ in self.data_loader:
        z_ = torch.rand((self.batch_size, self.z_dim))
        y_fill_ = y_.unsqueeze(2).unsqueeze(3).expand(self.batch_size, self.
            ↪ class_num, self.input_size, self.input_size)
        x_, z_, y_, y_fill_ = x_.cuda(), z_.cuda(), y_.cuda(), y_fill_.cuda()

        # update D network
        self.D_optimizer.zero_grad()

        D_real = self.D(x_, y_fill_)
        D_real_loss = self.BCE_loss(D_real, self.y_real_)

        G_ = self.G(z_, y_)
        D_fake = self.D(G_, y_fill_)
        D_fake_loss = self.BCE_loss(D_fake, self.y_fake_)

        D_loss = D_real_loss + D_fake_loss

        D_loss.backward()
        self.D_optimizer.step()

        # update G network
        self.G_optimizer.zero_grad()

        G_ = self.G(z_, y_)
        D_fake = self.D(G_, y_fill_)
        G_loss = self.BCE_loss(D_fake, self.y_real_)

        G_loss.backward()
        self.G_optimizer.step()
```

Above is the training code for GAN. For Discriminator, we first train the Discriminator by feeding the real datas and real labels to calculate the real data's loss. Then we sample noise from standard normal distribution to generate fake datas and labels to calculate the fake data's loss. After having these two losses, we add them and do the back propagation. For Generator, we sample noise from standard normal distribution to generate data and feed the generated datas and labels into Discriminator to calculate the loss, then do the back propagation.

2.3 DDPM

For the DDPM template, I use the code from github. For the model of DDPM, I chose U-Net as the generator. The downsampling part are made of DoubleConv blocks, SelfAttention blocks. The bottleneck part is made of DoubleConv blocks. The upsampling part is made of upsampling layers and SelfAttention blocks. I use MSE loss.

```
class UNet_conditional(nn.Module):
    def __init__(self, c_in=3, c_out=3, time_dim=256, num_classes=None, device
        ↪ ="cuda"):
        super().__init__()
        self.device = device
        self.time_dim = time_dim
        self.inc = DoubleConv(c_in, 64)
        self.down1 = Down(64, 128)
        self.sa1 = SelfAttention(128, 32)
        self.down2 = Down(128, 256)
        self.sa2 = SelfAttention(256, 16)
        self.down3 = Down(256, 256)
        self.sa3 = SelfAttention(256, 8)

        self.bot1 = DoubleConv(256, 512)
        self.bot2 = DoubleConv(512, 512)
        self.bot3 = DoubleConv(512, 256)

        self.up1 = Up(512, 128)
        self.sa4 = SelfAttention(128, 16)
        self.up2 = Up(256, 64)
        self.sa5 = SelfAttention(64, 32)
        self.up3 = Up(128, 64)
        self.sa6 = SelfAttention(64, 64)
        self.outc = nn.Conv2d(64, c_out, kernel_size=1)

        if num_classes is not None:
            self.label_emb = nn.Linear(num_classes, time_dim)

    def forward(self, x, t, y):
        t = t.unsqueeze(-1).type(torch.float)
        t = self.pos_encoding(t, self.time_dim)
        if y is not None:
            t += self.label_emb(y)

        x1 = self.inc(x)
        x2 = self.down1(x1, t)
        x2 = self.sa1(x2)
        x3 = self.down2(x2, t)
        x3 = self.sa2(x3)
        x4 = self.down3(x3, t)
        x4 = self.sa3(x4)

        x4 = self.bot1(x4)
        x4 = self.bot2(x4)
        x4 = self.bot3(x4)

        x = self.up1(x4, x3, t)
        x = self.sa4(x)
        x = self.up2(x, x2, t)
        x = self.sa5(x)
        x = self.up3(x, x1, t)
        x = self.sa6(x)
        output = self.outc(x)
        return output
```

For the noise scheduling function, I set it as linear function by using the **torch.linspace()**

```
def prepare_noise_schedule(self):
    return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)
```

For the time embedding, I use the concatenation of sine and cosine. The positional encoding helps the model to understand at which timestep it is operating during the denoising process. By providing a unique encoding for each timestep, the model can effectively learn to handle different levels of noise corresponding to different steps in the diffusion process.

```
def pos_encoding(self, t, channels):
    inv_freq = 1.0 / (
        10000
        ** (torch.arange(0, channels, 2, device=self.device).float() /
            ↪ channels)
    )
    pos_enc_a = torch.sin(t.repeat(1, channels // 2) * inv_freq)
    pos_enc_b = torch.cos(t.repeat(1, channels // 2) * inv_freq)
    pos_enc = torch.cat([pos_enc_a, pos_enc_b], dim=-1)
    return pos_enc
```

For the sampling method also the testing code. As we all know, diffusion model generates images by gradually denoising the noise (doing the reverse process) to get the image. By iteratively generates the noise need to be removed, we substract it with the original image to get the clearly image.

```
def sample(self, model, n, labels, cfg_scale=3):
    model.eval()
    with torch.no_grad():
        x = torch.randn((n, 3, self.img_size, self.img_size)).to(self.device)
        for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
            t = (torch.ones(n) * i).long().to(self.device)
            predicted_noise = model(x, t, labels)
            if cfg_scale > 0:
                uncond_predicted_noise = model(x, t, None)
                predicted_noise = torch.lerp(uncond_predicted_noise,
                    ↪ predicted_noise, cfg_scale)
            alpha = self.alpha[t][:, None, None, None]
            alpha_hat = self.alpha_hat[t][:, None, None, None]
            beta = self.beta[t][:, None, None, None]
            if i > 1:
                noise = torch.randn_like(x)
            else:
                noise = torch.zeros_like(x)
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 -
                ↪ alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise
    model.train()
    return x
```

For the training code, we first sample a time step and for that time step, we add noise to the image to get the specific level of noisy we want. Then we predict the noise we want and calculate the MSE loss until converge.

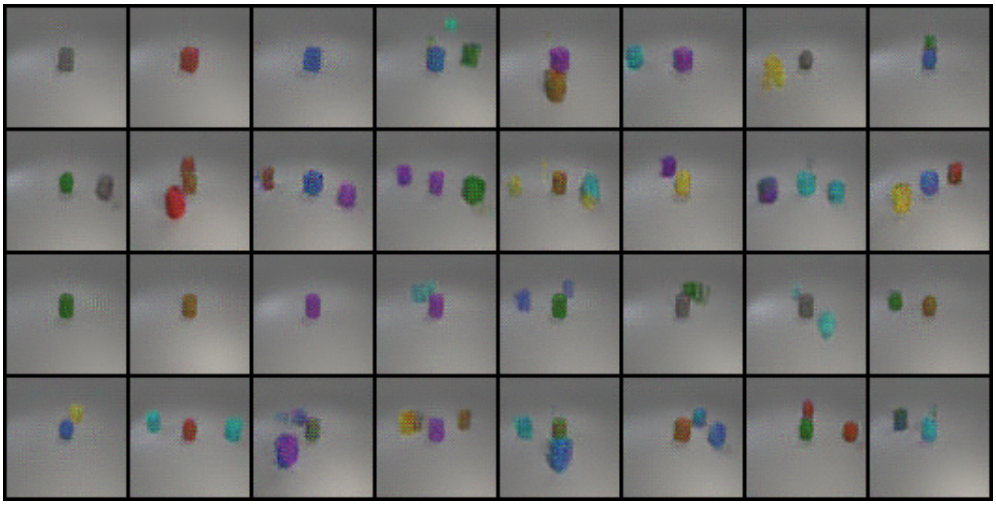
```
for epoch in range(args.epochs):
    for i, (images, labels) in enumerate(dataloader):
        t = diffusion.sample_timesteps(images.shape[0]).to(device)
        x_t, noise = diffusion.noise_images(images, t)
        if np.random.random() < 0.1:
            labels = None
        predicted_noise = model(x_t, t, labels)
        loss = mse(noise, predicted_noise)

        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()
```

3 Results and discussion

3.1 Show your synthetic image grids

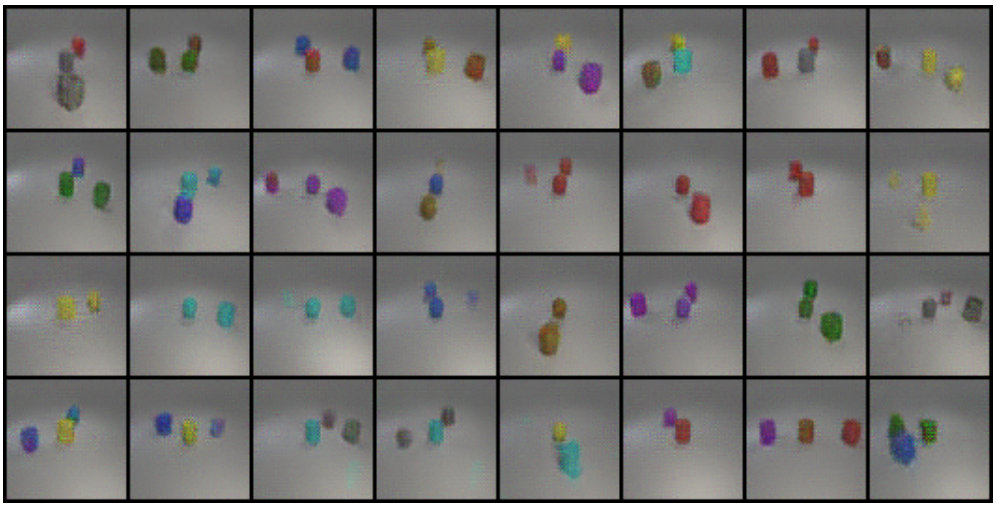


(a) Generated Sample

```
PS C:\Users\AlfredTing\Desktop\DLP\lab6\file\luo> python evaluator.py
Evaluating result:  0.75
```

(b) Score: 75%

Figure 4: GAN Testing Data

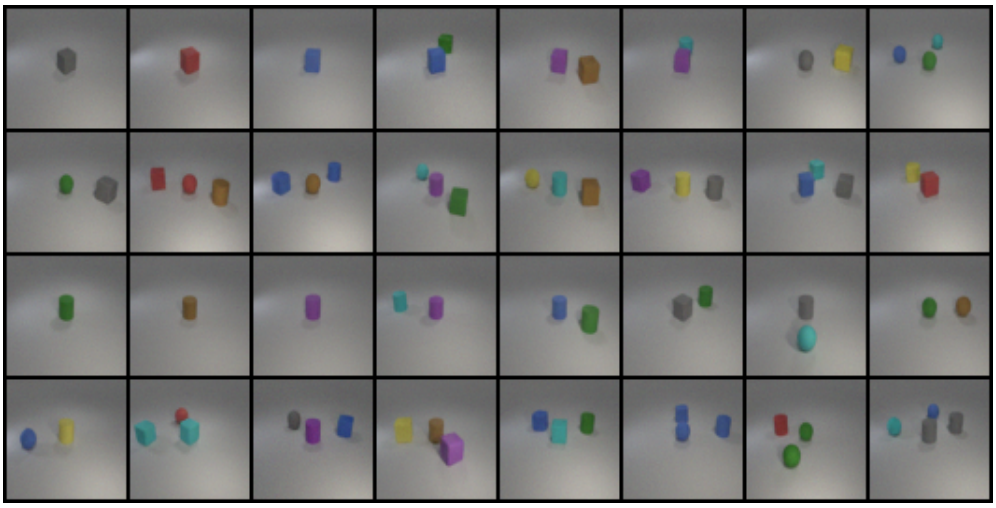


(a) Generated Sample

```
PS C:\Users\AlfredTing\Desktop\DLP\lab6\file\luo> python evaluator.py
Evaluating result:  0.7619047619047619
```

(b) Score: 76.2%

Figure 5: GAN New Testing Data



(a) Generated Sample

```python
# Testing json
x1 = torch.tensor([]).cuda()
for i in range(4):
    img1 = diffusion.sample(model, 8, y1[i*8:i*8+8], cfg_scale=0)
    x1 = torch.cat((x1, img1), dim=0)
torchvision.utils.save_image(x1, 'ddpm_test.png', nrow=8)
temp = x1
temp = (temp - 0.5) / 0.5

e = evaluation_model()
print(f'Evaluating result: {e.eval(temp, y1)}')
```

```
999it [00:27, 36.53it/s]
999it [00:27, 36.68it/s]
999it [00:27, 36.73it/s]
999it [00:27, 36.58it/s]
torch.Size([32, 24])
Evaluating result: 0.8472222222222222
```

(b) Score: 84.7%

Figure 6: DDPM Testing Data

(a) Generated Sample

```
# New testing json
x2 = torch.tensor([]).cuda()
for i in range(4):
    img2 = diffusion.sample(model, 8, y2[i*8:i*8+8], cfg_scale=0)
    x2 = torch.cat((x2, img2), dim=0)
torchvision.utils.save_image(x2, 'ddpm_new_test.png', nrow=8)
temp = x2
temp = (temp - 0.5) / 0.5

e = evaluation_model()
print(f'Evaluating result: {e.eval(temp, y2)}')
```

```
999it [00:27, 36.79it/s]
999it [00:27, 36.80it/s]
999it [00:27, 36.82it/s]
999it [00:27, 36.87it/s]
torch.Size([32, 24])
Evaluating result: 0.8690476190476191
```

(b) Score: 86.9%

Figure 7: DDPM New Testing Data

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

GAN used to be the SOTA model for image generation until the DDPM appears. For GAN, its biggest advantage is the speed of inference time. It can directly generates the image from a Gaussian noise. While behind the sweet, the mode collapse problems and the training unstability makes it hard to train, this is the disadvantages. For DDPM, its advantages are high quality image and stable training compared to GAN. However, the inference time for DDPM is very long, this is one of its biggest problems. It also requires high cost computational resources. Conclusion: GANs are preferred for applications requiring fast generation and high-quality images, despite their challenging training process. They are well-suited for tasks where direct sampling is crucial and quick image generation is necessary. DDPMs are favored for stable training and generating diverse, high-quality images, albeit at a higher computational cost. They are beneficial for applications where the training stability and flexibility in the generation process are critical.

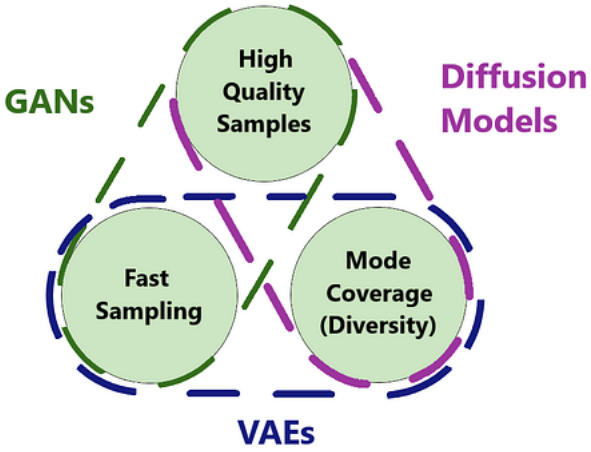


Figure 8: GAN and Diffusion Comparison

3.3 Discussion of your extra implementations or experiments

Since the training of DDPM is so stable, I have no trouble doing that. For GAN, there are so many questions. At first, I only apply the simplest GAN architecture, which is **Linear layer** to generate and discriminate. However, the outcome is so bad. Then I switch to DCGAN, and this becomes my final version. Since the generation starts from sampling a Gaussian noise $\mathbf{z}$ and then decodes to the task we want. So I think the dimension of the $\mathbf{z}$ will be related to the quality or anything to the GAN. I tested for dimension 50, 100, 150 of different z's and the best one is 50. I think maybe the task is not that complicated and we do not need so many dimensions to represent the latent features. The more dimension, the more complicate the task will be.

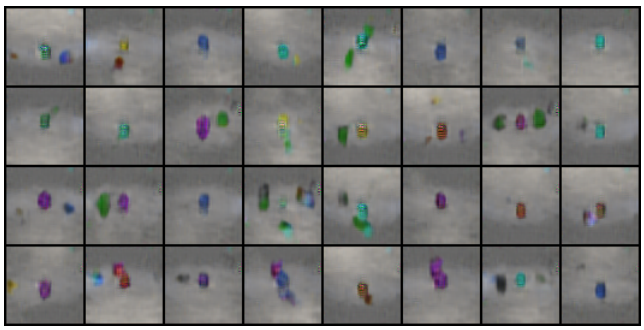


Figure 9: Bad Dimension Example