生成模型

神经网络与深度学习课程实验报告

班级：110

姓名：耿翊中

学号：2021213382

2023 年 12 月

目录

1. 实验要求

任务：（20 分）请选择一个生成模型（例如 VAE 或 GAN），以 CIFAR-10 为训练数据，训练一个生成器并从中生成新的图像。要求：代码需有注释，最终提交代码（jupyter notebook 文件）和模型，并撰写报告，包含模型原理、实验设置、实验结果分析（例如损失图、生成图像的例子）等。

附加题 1.（10 分）在上述基础上，进一步复现条件生成网络（conditional VAE 或 GAN），并与第一题中方法比较生成图像的差异，最终提交代码、模型，并撰写报告。具体要求同上。
附加题 2.（10 分）思考并定义一个图像生成问题，通过调研目前存在的方法和策略，设计相应的解决方法，并分析方法的可行性以及创新性。（注：本题目不需要代码实现，要求有新意，并撰写报告。）

2. 模型原理

2.1 GAN：生成对抗网络

2.1.1 GAN 结构

在 GAN 中，生成器和判别器是同时训练的。生成器试图生成越来越逼真的数据，而判别器则努力变得更擅长于区分真实数据和生成的数据。通过这种对抗过程，生成器能够生成越来越逼真的数据。

生成器网络（gnet）

生成器的目的是生成新的数据实例，这些数据实例看起来与训练数据类似。

输入层:

nn.ConvTranspose2d(latent_size, 4 * n_g_feature, kernel_size=4, bias=False): 这是一个转置卷积层（有时被称为反卷积层）。它将输入从潜在空间（数据的压缩表示）接收，并开始将其上采样到数据的维度。不使用偏置项可以简化模型。

批量归一化和激活函数:

nn.BatchNorm2d 和 nn.ReLU(): 这些在每个转置卷积层之后使用。批量归一化通过归一化层输入来稳定学习过程，ReLU（修正线性单元）引入非线性，使网络能够学习复杂的模式。

更多的转置卷积层:

后续的几层通过增加特征图的大小和降低其深度来进一步处理数据。每一层都包括转置卷积、批量归一化和 ReLU 激活函数。

输出层:

nn.ConvTranspose2d(n_g_feature, n_channel, kernel_size=4, stride=2, padding=1) 和 nn.Sigmoid(): 最后的转置卷积层产生与原始数据相同的通道数。Sigmoid 激活函数用于将输出数据标准化到[0,1]的范围内，这对于处理图像数据尤其有用。

```
gnet = nn.Sequential(
    nn.ConvTranspose2d(latent_size, 4 * n_g_feature, kernel_size=4, bias=False),
    nn.BatchNorm2d(4 * n_g_feature),
    nn.ReLU(),

    nn.ConvTranspose2d(4 * n_g_feature, 2 * n_g_feature, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(2 * n_g_feature),
    nn.ReLU(),

    nn.ConvTranspose2d(2 * n_g_feature, n_g_feature, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(n_g_feature),
    nn.ReLU(),

    nn.ConvTranspose2d(n_g_feature, n_channel, kernel_size=4, stride=2, padding=1),
    nn.Sigmoid()
)
```

判别器网络（dnet）

判别器的目的是区分输入数据是来自训练集还是生成器。

输入层:

nn.Conv2d(n_channel, n_d_feature, kernel_size=4, stride=2, padding=1) 和 nn.LeakyReLU(0.2): 判别器的第一层使用卷积层来提取特征，并通过 LeakyReLU 激活函数增加网络的非线性

卷积层和激活函数:

后续的层通过使用更多的卷积层、批量归一化和 LeakyReLU 激活函数来进一步处理特征。

输出层:

nn.Conv2d(4 * n_d_feature, 1, kernel_size=4): 最后一层是一个卷积层，其输出大小为 1，这表示判别器的决策（数据是真实的还是假的）。

```
dnet = nn.Sequential(
    nn.Conv2d(n_channel, n_d_feature, kernel_size=4, stride=2, padding=1),
    nn.LeakyReLU(0.2),

    nn.Conv2d(n_d_feature, 2 * n_d_feature, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(2 * n_d_feature),
    nn.LeakyReLU(0.2),

    nn.Conv2d(2 * n_d_feature, 4 * n_d_feature, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(4 * n_d_feature),
    nn.LeakyReLU(0.2),

    nn.Conv2d(4 * n_d_feature, 1, kernel_size=4)
)
```

2.1.2 GAN 原理

生成对抗网络（GAN）是一种深度学习模型，它由两个互相对抗的网络组成：生成器（Generator）和判别器（Discriminator）。GAN 的核心原理是通过这两个网络的对抗过程来生成数据。

1. 生成器（Generator）

生成器的目标是创造出逼真的数据实例。它接收一个随机噪声作为输入，通过神经网络生成数据。这些数据最初可能看起来很粗糙或不真实，但随着训练的进行，生成器学习如何制造越来越逼真的数据。

2. 判别器（Discriminator）

判别器的目标是区分输入数据是真实的还是由生成器创造的。它接收真实数据或生成器产生的数据作为输入，并尝试判断这些数据是真实的还是假的。其目标是准确识别出生成器制造的数据。

3. 对抗训练

GAN 的训练过程涉及到这两个网络的对抗。生成器试图欺骗判别器，制造出越来越逼真的数据，而判别器则努力学习如何区分真假数据。这个过程可以被看作是一个博弈，其中生成器和判别器不断提升各自的能力。

生成器训练：当判别器错误地将生成器的输出判定为真实数据时，生成器的性能得到提升。

判别器训练：当判别器正确地识别出真实和假数据时，判别器的性能得到提升。

4. 平衡和收敛

理想情况下，这个训练过程会达到一种平衡状态，生成器产生的数据足够逼真，以至于判别器无法准确区分真假数据。这意味着生成器能够产生高质量的数据，而判别器对真实数据和生成数据的区分能力也达到了极限。

2.1.3 GAN 生成图像原理

1. 生成器（Generator）

功能：生成器的目标是创造出逼真的图像。它从随机噪声（一种随机生成的输入数据）开始，逐渐学习如何构建与真实数据相似的图像。

工作原理：生成器通常是一个深度神经网络，它通过逐层转换输入的随机噪声，最终生成与训练数据集中的图像类似的输出。在图像生成的上下文中，生成器可能会使用如转置卷积层（deconvolutional layers）之类的技术来增加数据的空间维度，从而生成高维图像。

2. 判别器（Discriminator）

功能：判别器的任务是区分输入的图像是真实的（即来自训练数据集）还是由生成器创造的假图像。

工作原理：判别器也是一个深度神经网络，它对输入的图像进行分析，并给出一个概率，表明这个图像是真实的概率有多大。在训练过程中，它不断地从真实图像和生成器生成的图像中学习，以提高其区分真伪的能力。

3. 对抗训练过程

动态博弈：GAN 的核心在于生成器和判别器之间的动态博弈。生成器试图欺骗判别器，创造出越来越逼真的图像，而判别器则试图变得更擅长于识别这些假图像。

训练目标：生成器的目标是最大化判别器的错误率（即让判别器无法区分真假图像），而判别器的目标是准确区分真实图像和生成器生成的图像。

4. 结果与改进

生成逼真的图像：随着对抗训练的进行，生成器逐渐学会模仿训练数据集的分布，能够生成越来越逼真的图像。

训练挑战：GAN 的训练是有挑战性的，可能涉及到模式崩溃（mode collapse）、训练不稳定等问题。研究人员不断在开发新的架构、训练方法和技术，以提高 GAN 生成图像的质量和多样性。

2.2 CGAN：条件生成对抗网络

条件生成对抗网络（Conditional Generative Adversarial Networks，简称 CGAN）是生成对抗网络（GAN）的一个变体，它通过在生成器和判别器中引入额外的条件信息来生成特定类型的输出。这种条件信息通常是标签或其他类型的数据，用于指导生成过程。CGAN 的原理可以分为以下几个方面：

1. 条件信息

目的：在传统的 GAN 中，生成器和判别器仅依赖于输入数据和训练样本。而在 CGAN 中，它们还接受额外的条件信息，这些信息可以是类别标签、文本描述、另一幅图像等。

应用：例如，在生成特定类型的图像时，可以将类别标签作为条件信息输入给生成器和判别器。这样，生成器就能生成与给定标签相匹配的图像，而判别器则需要同时考虑图像的真实性和是否符合给定的条件。

2. 生成器（Generator）

调整：在 CGAN 中，生成器不仅接收随机噪声作为输入，还接收条件信息。这些信息被用来引导生成过程，以确保输出符合特定条件。

作用：这使得生成器能够根据条件生成特定类型的数据。例如，如果条件是数字标签，则生成器可以生成与该数字标签相对应的手写数字图像。

3. 判别器（Discriminator）

调整：同样，判别器也会接收相同的条件信息。它需要判断输入的数据不仅是否真实，还要判断是否符合给定的条件。

作用：这增加了判别器的难度，因为它必须同时考虑数据的真实性和条件的匹配度。

4. 对抗训练

过程：和传统的 GAN 一样，CGAN 也通过对抗训练过程来训练生成器和判别器。生成器尝试生成既逼真又符合条件的数据，而判别器则试图正确判断数据的真实性及其与条件的匹配程度。

结果：随着训练的进行，生成器变得越来越擅长于生成符合特定条件的逼真数据，而判别器则变得越来越擅长于识别这些数据。

```
class ConditionalGenerator(nn.Module):
    def __init__(self, latent_size, n_g_feature, n_channel, num_classes):
        super(ConditionalGenerator, self).__init__()
        self.label_emb = nn.Embedding(num_classes, latent_size)

        self.main = nn.Sequential(
            nn.ConvTranspose2d(latent_size * 2, 4 * n_g_feature, kernel_size=4, bias=False),
            nn.BatchNorm2d(4 * n_g_feature),
            nn.ReLU(),

            nn.ConvTranspose2d(4 * n_g_feature, 2 * n_g_feature, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(2 * n_g_feature),
            nn.ReLU(),

            nn.ConvTranspose2d(2 * n_g_feature, n_g_feature, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(n_g_feature),
            nn.ReLU(),

            nn.ConvTranspose2d(n_g_feature, n_channel, kernel_size=4, stride=2, padding=1),
            nn.Sigmoid()
        )

    def forward(self, z, labels):
        c = self.label_emb(labels)
        x = torch.cat([z, c.unsqueeze(2).unsqueeze(3)], 1)
        return self.main(x)


```

```
class ConditionalDiscriminator(nn.Module):
    def __init__(self, n_channel, n_d_feature, num_classes):
        super(ConditionalDiscriminator, self).__init__()
        self.label_emb = nn.Embedding(num_classes, n_channel * 32 * 32)

        self.main = nn.Sequential(
            nn.Conv2d(n_channel * 2, n_d_feature, kernel_size=4, stride=2, padding=1)
            ,
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_d_feature, 2 * n_d_feature, kernel_size=4, stride=2, padding=1, bias=False)
,
            nn.BatchNorm2d(2 * n_d_feature),
            nn.LeakyReLU(0.2),

            nn.Conv2d(2 * n_d_feature, 4 * n_d_feature, kernel_size=4, stride=2, padding=1, bias=F
alse),
            nn.BatchNorm2d(4 * n_d_feature),
            nn.LeakyReLU(0.2),

            nn.Conv2d(4 * n_d_feature, 1, kernel_size=4)
        )

    def forward(self, x, labels):
        c = self.label_emb(labels).view(-1, n_channel, 32, 32)
        x = torch.cat([x, c], 1)
        return self.main(x)
```

3. 实验设置

3. 1 数据准备

加载 CIFAR-10 数据集并应用变换。

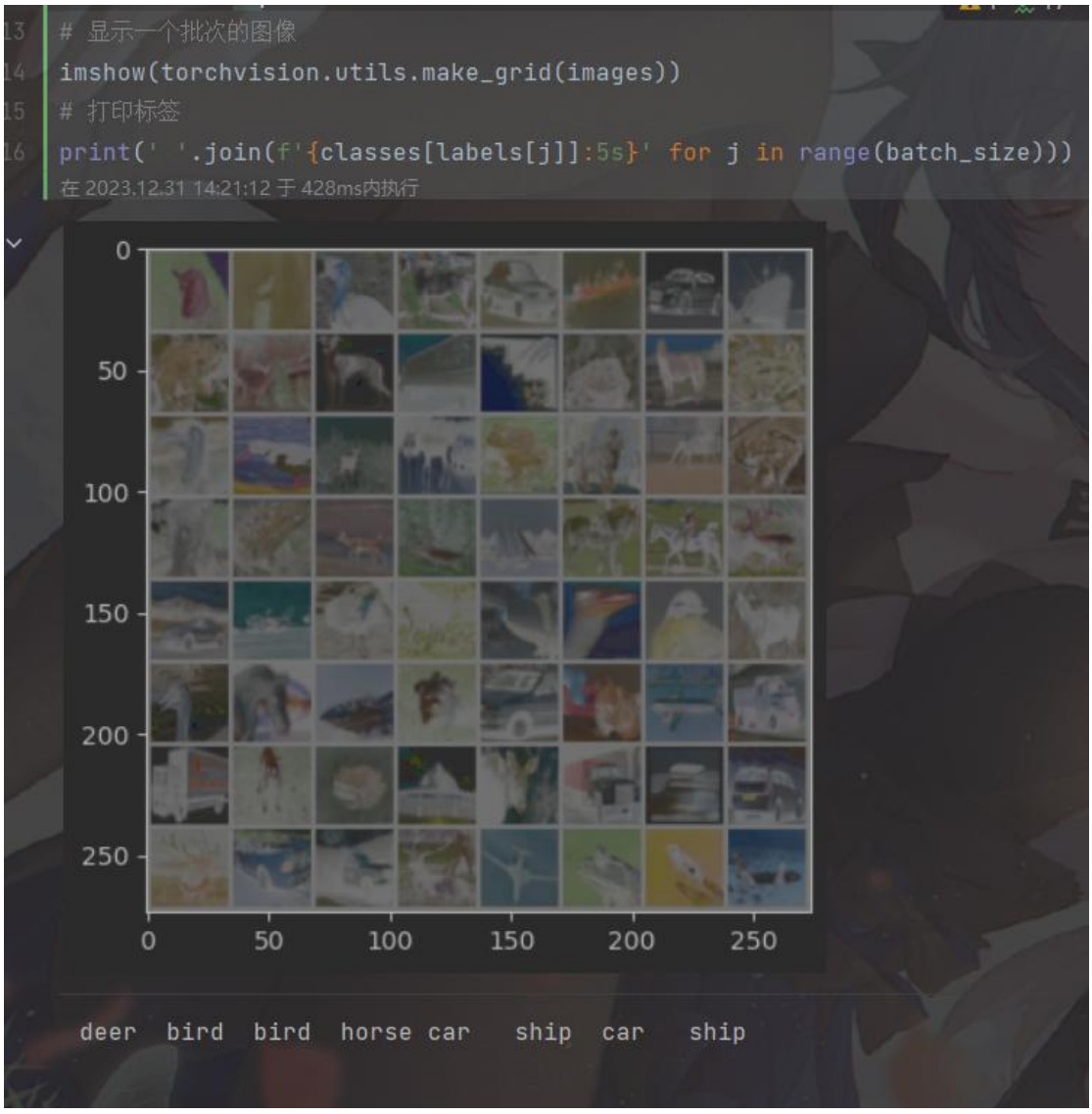
使用 DataLoader 进行批处理和洗牌。

```
dataset = CIFAR10(root='./CIFARdata', download=True, transform=transforms.ToTensor())
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

由于 DIFAR-10 数据集本身比较模糊，所以展示一部分，以跟结果进行对照

```
import torchvision
batch_size = 8
# 获取一个批次的图像和标签
images, labels = next(iter(dataloader))
# 定义 CIFAR-10 类别
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
# 显示图像的函数
def imshow(img):
```

```
        img = img / 2 + 0.5  # 反规范化
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()
# 显示一个批次的图像
imshow(torchvision.utils.make_grid(images))
# 打印标签
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```



3. 2 模型设置和训练

3. 2. 1 模型设置

使用二元交叉熵损失（BCEWithLogitsLoss）。
为生成器和判别器分别设置 Adam 优化器。

```
criterion = nn.BCEWithLogitsLoss()
goptimizer = torch.optim.Adam(gnet.parameters(), lr=0.0002, betas=(0.5, 0.999))
doptimizer = torch.optim.Adam(dnet.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

3.2.2 训练过程

初始化

1. Batch Size：这指定了每次训练迭代中使用的数据样本的数量，这里设置为 64。
2. Fixed Noises：固定的随机噪声向量，用于生成固定点的图像，以观察生成器随训练进度的变化。
3. 损失列表：用于记录判别器（d_losses）和生成器（g_losses）的损失。

训练循环

训练包括多个周期（epochs），每个周期遍历整个数据集。

对于每个批次：

1. 训练判别器：

使用真实图像训练判别器。计算判别器对真实图像的损失（dloss_real）。

使用生成器生成的假图像训练判别器。计算判别器对假图像的损失（dloss_fake）。

更新判别器的损失为这两部分的和，并进行反向传播和优化步骤。

2. 训练生成器：

生成器试图欺骗判别器，使其将假图像判定为真实图像。

计算生成器的损失（gloss），并进行反向传播和优化步骤。

3. 损失记录：记录这两个网络的损失，用于后续分析和可视化。
4. 保存和打印信息：

每 100 个批次，使用固定的噪声生成图像并保存，以观察生成器的进展。

打印训练的相关信息，包括周期索引、批次索引、损失值和判别器的表现。

损失曲线绘制

最后，使用 matplotlib 绘制生成器和判别器的损失曲线，以可视化训练过程中的损失变化。

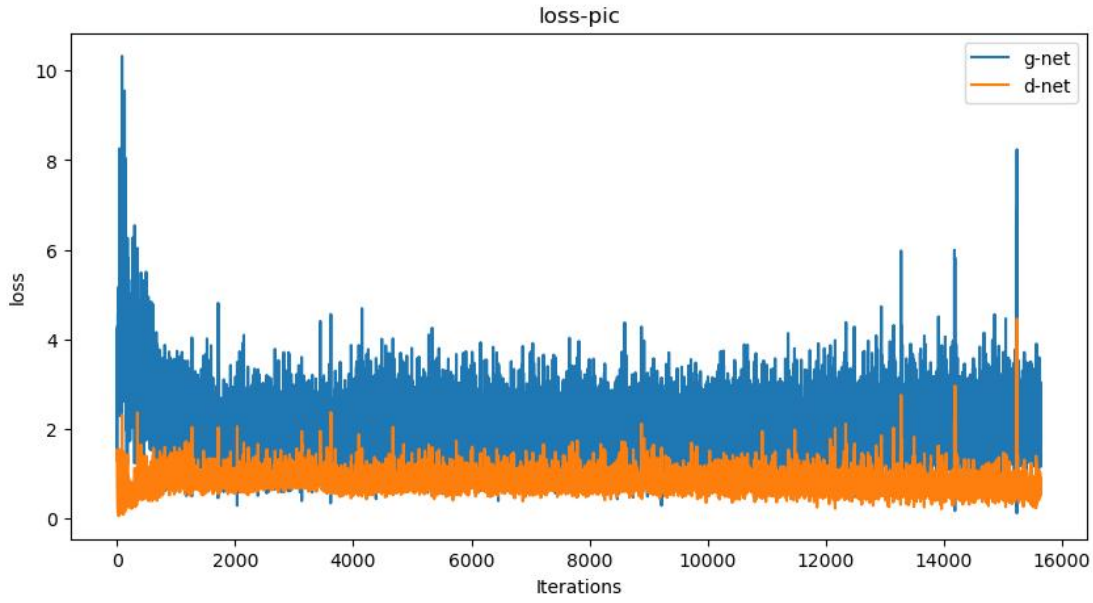
```
batch_size = 64
fixed_noises = torch.randn(batch_size, latent_size, 1, 1)

# 初始化用于跟踪损失的列表
d_losses = []
g_losses = []

epoch_num = 20
for epoch in range(epoch_num):
    for batch_idx, data in enumerate(dataloader):
        real_images, _ = data
        batch_size = real_images.size(0)

        # 首先，用真实图像训练判别器
        labels = torch.ones(batch_size)
        preds = dnet(real_images)
        outputs = preds.reshape(-1)
```

```
        dloss_real = criterion(outputs, labels)
        dmean_real = outputs.sigmoid().mean()

        # 接着，用假图像训练判别器
        noises = torch.randn(batch_size, latent_size, 1, 1)
        fake_images = gnet(noises)
        labels = torch.zeros(batch_size)
        fake = fake_images.detach()

        preds = dnet(fake)
        outputs = preds.view(-1)
        dloss_fake = criterion(outputs, labels)
        dmean_fake = outputs.sigmoid().mean()

        # 更新判别器
        dloss = dloss_real + dloss_fake
        dnet.zero_grad()
        dloss.backward()
        doptimizer.step()

        # 训练生成器
        labels = torch.ones(batch_size)
        preds = dnet(fake_images)
        outputs = preds.view(-1)
        gloss = criterion(outputs, labels)
        gmean_fake = outputs.sigmoid().mean()
        gnet.zero_grad()
        gloss.backward()
        goptimizer.step()

        # 记录损失
        d_losses.append(dloss.item())
        g_losses.append(gloss.item())

        if batch_idx % 100 == 0:
            fake = gnet(fixed_noises)
            save_image(fake, f'./GAN_saved/images_epoch{epoch:02d}_batch{batch_idx:03d}.png')

            print(f'Epoch index: {epoch}, {epoch_num} epoches in total.')
            print(f'Batch index: {batch_idx}, the batch size is {batch_size}.')
            print(f'Discriminator loss is: {dloss}, generator loss is: {gloss}', '\n',
                  f'Discriminator tells real images real ability: {dmean_real}', '\n',
                  f'Discriminator tells fake images real ability: {dmean_fake:g}/{gmean_fake:g}')
```

```
# 绘制损失曲线
plt.figure(figsize=(10,5))
plt.title("loss-pic")
plt.plot(g_losses, label="g-net")
plt.plot(d_losses, label="d-net")
plt.xlabel("Iterations")
plt.ylabel("loss")
plt.legend()
plt.show()
```

4. 实验结果分析

4.1 损失曲线



初始阶段：

图中显示，训练初期，生成器损失（蓝色线）相对较高，这意味着生成器生成的图像质量不高，判别器能较容易地区分真实图像和生成图像。

判别器损失（橙色线）开始时下降较快，这表明判别器在学习如何准确区分真假图像方面取得了快速进展。

训练过程：

随着训练的进行，生成器损失逐渐下降，并开始稳定。这表明生成器在生成逼真图像的能力上有所提升。

判别器损失在训练中期和后期波动较小，维持在一个较低的水平，这表示判别器已经较好地学会了区分真实图像和生成图像。

波动：

两个网络的损失都显示出一定的波动性，这是 GAN 训练中常见的现象，可能是由于每个批次数据的不同或者学习率设置引起的。

损失稳定性：

对于生成器来说，损失的波动在训练后期有所增加，这可能是因为生成器在尝试生成更多样化的图像，而判别器在不断适应这些新样本。
判别器损失相对稳定，这可能表明它能够较为一致地对真实和生成的图像进行准确判断。
收敛性：

如果生成器和判别器的损失都达到稳定且较低的状态，可能表明网络正在收敛。但在这幅图中，我们看不到明显的收敛迹象，特别是生成器的损失仍然有较大的波动。
模式崩溃：

在某些情况下，GAN 训练可能会遇到模式崩溃（mode collapse）的问题，此时生成器开始生成非常相似的输出，以欺骗判别器。尽管损失曲线没有直接显示这种情况，但如果生成器损失的波动过于剧烈，可能是一个警示信号。

4. 2 训练过程中与测试生成图片

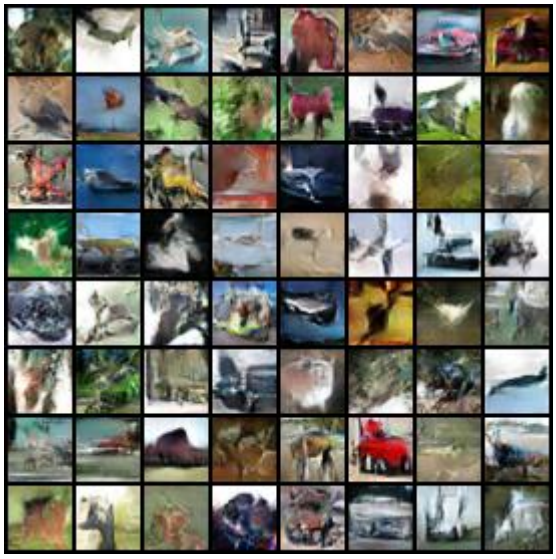
训练过程中生成的图片：

1epoch,200batch：

5epoch,200batch:

15epoch,200batch:

20epoch,700batch:



使用保存的模型，生成图片：

```
gnet_save_path = 'gnet.pt'
torch.save(gnet, gnet_save_path)

dnet_save_path = 'dnet.pt'
torch.save(dnet, dnet_save_path)

gnet = torch.load(gnet_save_path)
gnet.eval()

dnet = torch.load(dnet_save_path)
dnet.eval()

for i in range(100):
```

```
        noises = torch.randn(batch_size, latent_size, 1, 1, device=device)
        fake_images = gnet(noises).to(device)
        save_image(fake_images, f'./GAN_Generated_Images/{i}.png')
```

可以明显看出，随着迭代次数增加，生成图片越来越真实

5. 附加题：CGAN 的实现

1. 条件信息：

CGAN 模型中的生成器和判别器都接收额外的条件信息。在这段代码中，条件信息是通过嵌入层（nn.Embedding）处理的类别标签，这些嵌入后的标签与噪声向量或图像数据相结合，用于影响生成的图像内容或判别器的决策。

```
# 生成器中的条件信息嵌入
self.label_emb = nn.Embedding(num_classes, latent_size)

# 判别器中的条件信息嵌入
self.label_emb = nn.Embedding(num_classes, n_channel * 32 * 32)
```

2. 网络结构的改变：

在生成器（ConditionalGenerator）中，条件标签首先通过一个嵌入层转换为与潜在噪声相同大小的张量，然后与潜在噪声在通道维度上连接。这样可以保证生成的图像与条件标签相对应。

在判别器（ConditionalDiscriminator）中，条件标签通过嵌入层转换为具有与输入图像相同空间维度的张量，然后与图像数据在通道维度上连接。这允许判别器在判断图像真假的同时考虑条件信息。

```
# 生成器中将条件信息和随机噪声合并
c = self.label_emb(labels)
x = torch.cat([z, c.unsqueeze(2).unsqueeze(3)], 1)

# 判别器中将条件信息和图像数据合并
c = self.label_emb(labels).view(-1, n_channel, 32, 32)
x = torch.cat([x, c], 1)
```

3. 损失函数的计算：

损失函数计算过程中考虑了条件信息，使得生成器不仅要生成逼真的图像，还要确保图像与给定的条件相匹配。判别器在评估图像时，也要判断图像是否符合其条件标签。

```
# 计算真实图像的损失时考虑条件标签
outputs = netD(real_images, labels).view(-1, 1)
d_loss_real = criterion(outputs, real_labels)

# 生成假图像并计算损失时考虑条件标签
fake_images = netG(noise, labels)
```

```
    outputs = netD(fake_images.detach(), labels).view(-1, 1)
    d_loss_fake = criterion(outputs, fake_labels)
```

4. 训练过程：
训练过程中，每次生成假图像或进行真实图像的真假判断时，都要输入相应的条件标签。这是 CGAN 和普通 GAN 的主要区别。在普通 GAN 中，生成器只接收随机噪声，而判别器只接收图像。

```
# 训练判别器时使用条件标签
real_labels = torch.ones(current_batch_size, 1).to(device)
labels = torch.randint(0, num_classes, (current_batch_size,), device=device)
outputs = netD(real_images, labels).view(-1, 1)

# 训练生成器时使用条件标签
noise = torch.randn(current_batch_size, latent_size, 1, 1, device=device)
labels = torch.randint(0, num_classes, (current_batch_size,), device=device)
fake_images = netG(noise, labels)
```

5. 生成图像的多样性和特定性：
通过在训练过程中引入条件，CGAN 能够生成特定类别的图像。例如，如果条件是“猫”，生成器将生成猫的图像，而判别器将学习如何识别猫的图像是否真实。这与普通 GAN 不同，后者生成的图像通常更加多样化，但缺乏特定的指导条件。

```
# 生成特定类别的图像
with torch.no_grad():
    fake = netG(fixed_noises.to(device), fixed_labels.to(device)).detach().cpu()
```

6. 附加题：思考并定义一个图像生成问题，通过调研目前存在的方法和策略，设计相应的解决方法，并分析方法的可行性以及创新性。

图像生成问题定义：
问题：设计一个系统来生成逼真的街道风景图像，这些图像能够根据不同的天气条件（如晴天、雨天、雪天）和日间或夜间的时间设置进行调整。这种系统可以用于视频游戏环境的设计、虚拟现实、以及驾驶模拟器中环境的快速渲染。

现有方法和策略：
使用条件生成对抗网络（CGAN）根据不同的天气和时间条件生成图像。
利用循环神经网络（RNN）或变分自编码器（VAE）来捕捉时序信息，适用于动态场景的生成。
引入风格迁移技术，将某种天气或时间的风格应用到基本街道场景上。

解决方法设计：
多条件生成对抗网络（Multi-Conditional GAN, MCGAN）：设计一个 GAN 网络，它接受多个条件标签，包括天气和时间。生成器能够根据这些标签生成对应条件的街道风景图像。

生成器：接受随机噪声、天气标签、时间标签作为输入，输出相应条件的街道图像。
判别器：判断输入的街道图像是否逼真，并且检查生成图像是否与所给条件匹配。

渐进式训练策略：为了提高图像质量和生成速度，采用渐进式训练方法。从生成低分辨率的图像开始，逐步增加网络的深度和复杂性，最终生成高分辨率的图像。

风格编码器：引入一个风格编码器，它从现有的街道风景图像中学习不同天气和时间的风格特征。这些风格特征随后可以被用来指导生成器生成具有特定风格的图像。

可行性分析：
使用 CGAN 作为基础是可行的，因为 CGAN 已经被证明可以生成符合特定条件的图像。
渐进式训练策略可以帮助模型首先学习大致的结构和组成，然后逐步增加细节，这有助于提高生成图像的质量并加快训练速度。
风格编码器可以从现有数据中学习风格，这样的技术已经在风格迁移和艺术作品生成中得到应用。
创新性分析：
将 MCGAN 应用到街道风景生成是一个创新的应用领域，特别是在交互式应用程序中，如视频游戏和模拟器。
渐进式训练策略虽然不是新的，但将其与多条件 GAN 相结合以生成高质量的街道场景是一个新颖的方法。
风格编码器的引入为控制生成图像的风格提供了更大的灵活性和准确性。