南　　开　　大　　学

计　算　机　学　院

深度学习及应用实验报告

实验四　生成网络 (GAN) 的实现

姓名：苏航

学号：2111039

专业：信息安全

2024 年 6 月 10 日

摘要

本实验在理解生成网络基本思想的情况下，动手实现了基本生成网络以及基于卷积的生成网络，在 FashionMnist 数据集上进行训练，比较了两种网络的生成效果。并通过调节输入生成器的随机数得出其对生成图片生成的影响，并给出了一定的解释说明。

关键字：生成神经网络;CNN; 随机数

目录

一、 实验要求 **1**

二、 实验内容 **1**

三、 实验过程 **1**

（一） 生成神经网络的基本原理 1

（二） 原始 GAN 的实现 2

（三） 原始 GAN 的训练与评估 3

（四） 卷积 GAN 的实现 6

（五） 尝试改变随机数 9

四、 总结 **15**

一、 实验要求

本次实验要求掌握生成神经网络的原理, 学会在 pytorch 框架下搭建生成神经网络来训练 FashionMNIST 数据集

.

二、 实验内容

打印原始版本 GAN 网络结构（可用 print(net) 打印，复制文字或截图皆可）、在 Fashion-MNIST 数据集上的训练 loss 曲线; 自定义一组随机数，生成 8 张图; 针对自定义的 100 个随机数，自由挑选 5 个随机数，查看调整每个随机数时，生成图像的变化（每个随机数调整 3 次，共生成 15x8 张图），总结调整每个随机数时，生成图像发生的变化解释不同随机数调整对生成结果的影响 (重点部分)

三、 实验过程

(一) 生成神经网络的基本原理

生成对抗网络是一种深度学习模型，由两个神经网络组成：生成器（Generator）和判别器（Discriminator），二者通过对抗训练的方式共同学习。其基本原理如下：

生成器（Generator）：生成器的目标是学习生成与真实数据类似的新样本。它接收随机噪声（通常是高斯分布或均匀分布中的随机向量）作为输入，并尝试将其映射到数据空间中。生成器通过不断尝试生成越来越逼真的数据样本来提高自己的性能。

判别器（Discriminator）：判别器的任务是区分生成器生成的假样本和真实数据样本。它接收来自生成器和真实数据的样本，并尝试将它们正确分类为真实的或虚假的。判别器通过对生成器生成的样本进行评估，不断提高自己的识别能力。

对抗训练过程：生成器和判别器之间进行对抗性的训练。在训练过程中，生成器努力生成越来越逼真的样本，以欺骗判别器，使其无法区分生成的假样本和真实数据。同时，判别器也在努力提高自己的识别能力，以更准确地区分真实样本和生成样本。这种对抗过程促使两个网络都不断提高自己的性能，最终达到一个动态平衡点，生成器生成的样本足够逼真，以至于判别器无法准确区分。

目标函数：GANs 的训练目标是找到一个纳什均衡点，即生成器生成的样本足够逼真，以至于判别器无法准确区分，同时生成器不断改进以产生更逼真的样本，判别器也不断改进以提高其分类准确度。这个过程可以通过最小化生成器和判别器的损失函数来实现。通常情况下，生成器的损失函数是判别器无法正确分类生成的样本的概率，而判别器的损失函数是将真实样本和生成样本正确分类的概率之和。

具体的模型结构如下所示:

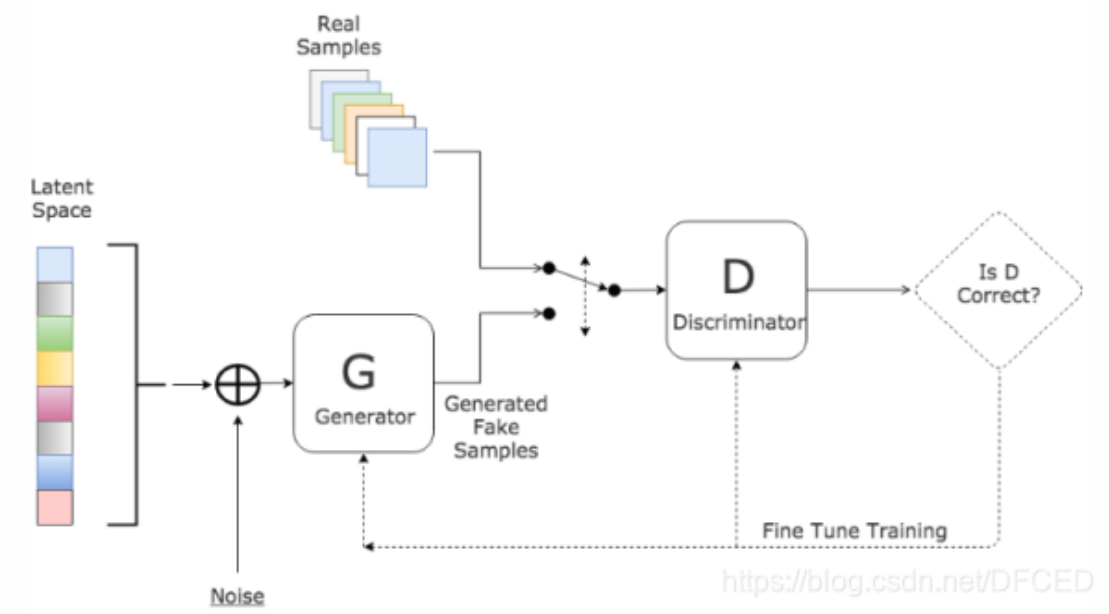

图 1: GAN 模型示意图

(二) 原始 GAN 的实现

实验要求我们复现简单的 GAN 网络结构, 如下所示。

首先是判别器：输入维度：784（28x28 的图像展平为 784 维的向量）；第一层：线性层（全连接层），输入维度为 784，输出维度为 128；第二层：LeakyReLU 激活函数，具有 0.2 的负斜率参数；第三层：线性层，输入维度为 128，输出维度为 1；输出层：经过 Sigmoid 激活函数，输出一个概率值，表示输入图像是真实图像的概率（范围为 0 到 1）

然后是生成器：输入维度：100（随机噪声向量的维度）；第一层：线性层（全连接层），输入维度为 100，输出维度为 128；第二层：LeakyReLU 激活函数，具有 0.2 的负斜率参数；第三层：线性层，输入维度为 128，输出维度为 784；输出层：经过 tanh 激活函数，将输出限制在 [-1, 1] 的范围内，然后通过 view 方法将输出转换为图像形状（batch_size x 1 x 28 x 28）

```
class Discriminator(torch.nn.Module):
    def __init__(self, inp_dim=784):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(inp_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 1)
    def forward(self, x):
        x = x.view(x.size(0), 784) # flatten (bs x 1 x 28 x 28) -> (bs x 784)
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.sigmoid(out)
        return out
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
```

```
        self.fc2 = nn.Linear(128, 784)
    def forward(self, x):
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.tanh(out) # range [-1, 1]
        # convert to image
        out = out.view(out.size(0), 1, 28, 28)
        return out
```

搭建成功模型后我们可以把它输出出来:

```
Discriminator(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (nonlin1): LeakyReLU(negative_slope=0.2)
  (fc2): Linear(in_features=128, out_features=1, bias=True)
)
Generator(
  (fc1): Linear(in_features=100, out_features=128, bias=True)
  (nonlin1): LeakyReLU(negative_slope=0.2)
  (fc2): Linear(in_features=128, out_features=784, bias=True)
)
```

图 2: 原始 GAN 网络结构图

生成器负责从随机噪声中生成假的图像样本, 而判别器则负责区分真实的图像样本和生成器生成的假样本。它们通过对抗性训练的方式相互作用, 使得生成器生成的样本越来越逼真, 同时判别器也越来越难以区分真实和假样本。

(三) 原始 GAN 的训练与评估

搭建完基本的 GAN 后我们就可以开始对其展开训练与评估, 实验要求我们采用 FashionMnist 数据集, 在进行训练之前我们需要对其进行预处理, 代码如下:

```
dataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/',
                        transform=transforms.Compose([transforms.ToTensor(),
                                                      transforms.Normalize
                                                          ((0.5,), (0.5,))]),
                        download=True)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True
    )
```

损失函数上我们选用交叉熵损失函数, 这个损失函数通常用于二分类问题, 在 GAN 中, BCELoss 通常用来衡量生成图像与真实图像之间的差异。

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device
    ('cpu')
D = Discriminator().to(device)
G = Generator().to(device)
optimizerD = torch.optim.Adam(D.parameters(), lr=0.0002)
```

```
optimizerG = torch.optim.Adam(G.parameters(), lr=0.0002)
criterion = nn.BCELoss()
```

然后我们就可以开始训练了，注意 GAN 模型的训练方式不同于其他模型的训练思路，因为它有两个独立的互不干涉的模型，在具体训练时需要分别优化这两个模型

```
    def train_and_generate_images(G, D, num_rounds=200):
    # 标签定义
    lab_real = torch.ones(64, 1, device=device)
    lab_fake = torch.zeros(64, 1, device=device)
    # 储存损失值
    loss_D_list = []
    loss_G_list = []
    # 将数据加载器转换为迭代器
    dataloader_iter = iter(dataloader)
    model_count = 1  # 模型保存次数
    for epoch in range(num_rounds):
        for i, data in enumerate(dataloader, 0):
            # STEP 1: 判别器优化步骤
            try:
                x_real, _ = next(dataloader_iter)
            except StopIteration:
                dataloader_iter = iter(dataloader)
                x_real, _ = next(dataloader_iter)

            x_real = x_real.to(device)
            batch_size = x_real.size(0)  # 动态获取批量大小
            lab_real = torch.ones(batch_size, 1, device=device)
            lab_fake = torch.zeros(batch_size, 1, device=device)
            optimizerD.zero_grad()
            D_x = D(x_real)
            lossD_real = criterion(D_x, lab_real)
            z = torch.randn(batch_size, 100, device=device)  # 随机噪声, 64个
                样本, z_dim=100
            x_gen = G(z).detach()
            D_G_z = D(x_gen)
            lossD_fake = criterion(D_G_z, lab_fake)
            lossD = lossD_real + lossD_fake
            lossD.backward()
            optimizerD.step()
            # STEP 2: 生成器优化步骤
            optimizerG.zero_grad()
            z = torch.randn(batch_size, 100, device=device)  # 随机噪声, 64个
                样本, z_dim=100
            x_gen = G(z)
            D_G_z = D(x_gen)
            lossG = criterion(D_G_z, lab_real)  # -log D(G(z))
            lossG.backward()
            optimizerG.step()
```

```
        loss_D_list.append(lossD.item())
        loss_G_list.append(lossG.item())
        print(epoch, lossD.item(), lossG.item())
        # 保存模型
        if epoch % 10 == 0:
            D_path = os.path.join(save_dir, f'discriminator_{model_count}.pth')
            G_path = os.path.join(save_dir, f'generator_{model_count}.pth')
            torch.save(D.state_dict(), D_path)
            torch.save(G.state_dict(), G_path)
        model_count += 1
    return loss_D_list, loss_G_list
```

实现该网络的训练和评估后我们得到下面的结果:

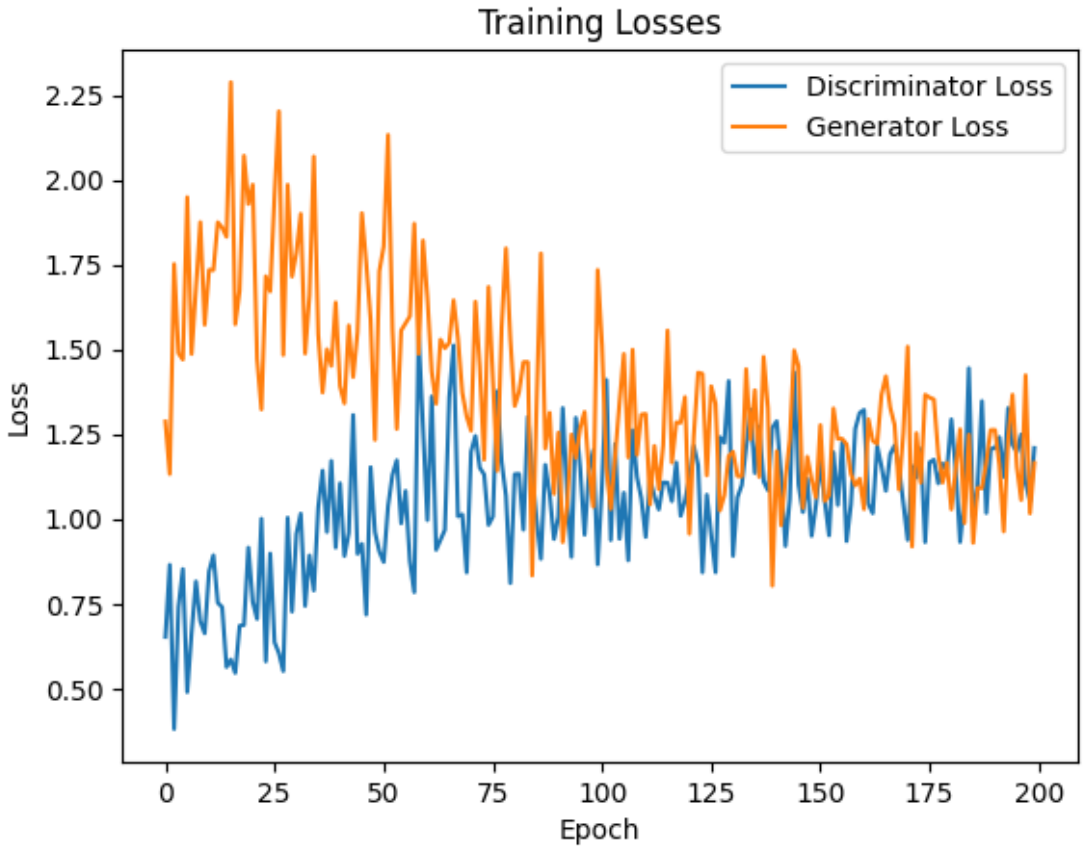

图 3: 原始 GAN 训练损失曲线

从这个训练得到的 Loss 曲线上来看，判别器的 Loss 曲线近似收敛，生成器的 Loss 曲线却有发散的趋势。这也是原始 GAN 模型存在的最大问题：生成器难以收敛。而且随着轮数的增多，肯能会出现生成器和判别器的质量都没有变好，甚至一起退化的情况。接下来我们将分析生成器生成的图片。

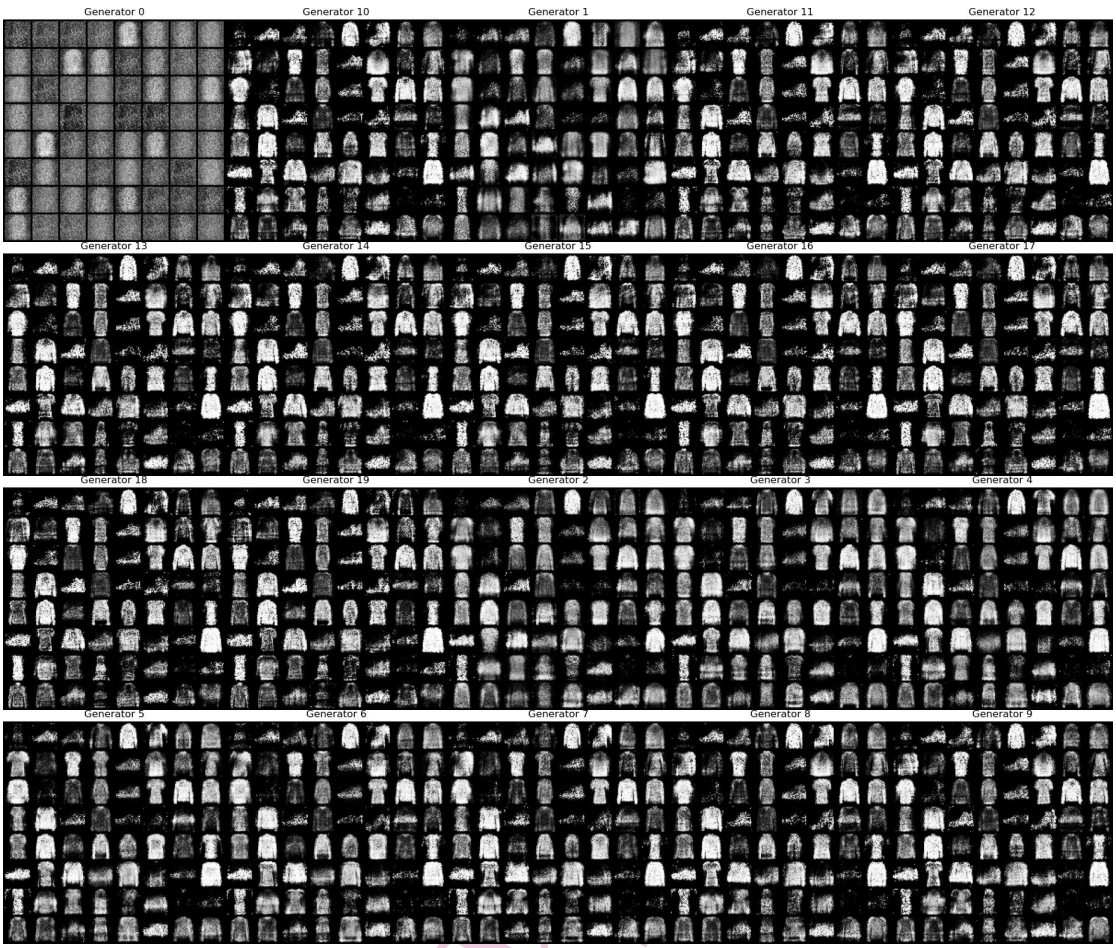

图 4: 原始 GAN 生成的 20 组图像

上图是从初始开始每隔 10 轮调用生成器生成的图像，我们可以看到初始时是一个随机的噪声向量，后来随着训练的进行越来越逼真，但后来的生成图片的质量并没有显著上升，还有一些毛刺，这证明生成器的质量还不太好。只是增加训练轮数不能提升生成图片的质量我们就要考虑使用其他生成器了。

(四) 卷积 GAN 的实现

生成器接受一个形状为 (batch_size, z_dim) 的随机噪声向量 x；然后将输入向量 x 变形为 (batch_size, z_dim, 1, 1) 的 4D 张量，以便适应反卷积操作。

反卷积层包含三层:

第一层:nn.ConvTranspose2d(z_dim, 128, kernel_size=7, stride=1, padding=0, bias=False) 将输入噪声向量变换为 (batch_size, 128, 7, 7) 的特征图。

第二层: nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False) 将特征图上采样为 (batch_size, 64, 14, 14)。

第三层: nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1, bias=False) 将特征图上采样为最终的图像大小 (batch_size, 1, 28, 28)。

然后在每个反卷积层后面加上 nn.BatchNorm2d 层来稳定训练过程，这个层就是批归一化层；使用 nn.ReLU(True) 作为中间层的激活函数，使用 nn.Tanh() 作为输出层的激活函数，将输出范围映射到 [-1, 1]。

下面我们根据 pytorch 提供的框架构建生成器:

```
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(z_dim, 128, kernel_size=7, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        x = x.view(x.size(0), 100, 1, 1)  # (batch_size, z_dim, 1, 1)
        return self.main(x)
```

判别器的结构和生成器不同,具体来说它有两个卷积层,中间层有两个激活函数 nn.LeakyReLU(0.2, inplace=True) 这是用来防止神经元坏死的。在第二个卷积层之后还有一个批量归一化层，之后就是展平，全连接层以及激活函数，具体代码如下:

```
    class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.main(x)
```

接下来我们观察其评估结果:

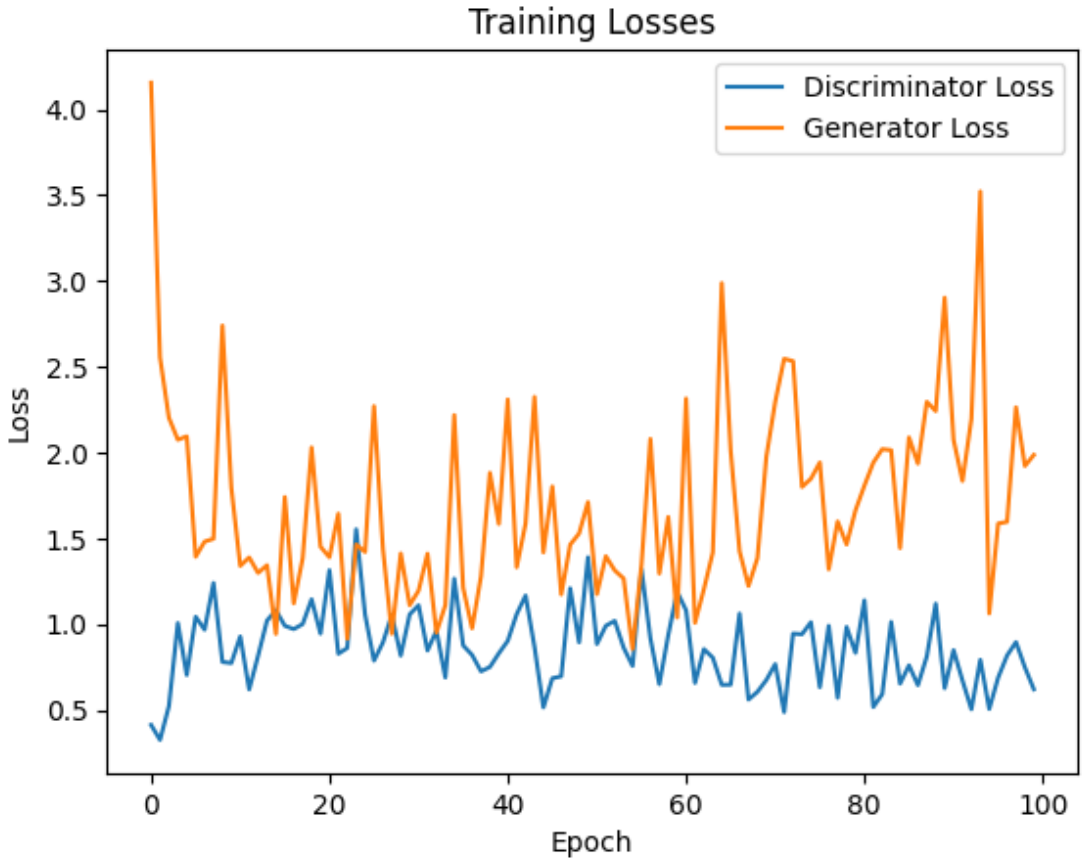

图 5: 卷积 GAN 的损失曲线

我们观察到在卷积 GAN 中生成器的 Loss 曲线在后续发散了，这显然不是我们希望的。这也是 GAN 模型最大的弊病，所幸我们在每一轮训练中都保存了模型，下面我们将显示这些模型展示的图片。后续变换随机数的操作也是在这个卷积 GAN 的基础上进行的。

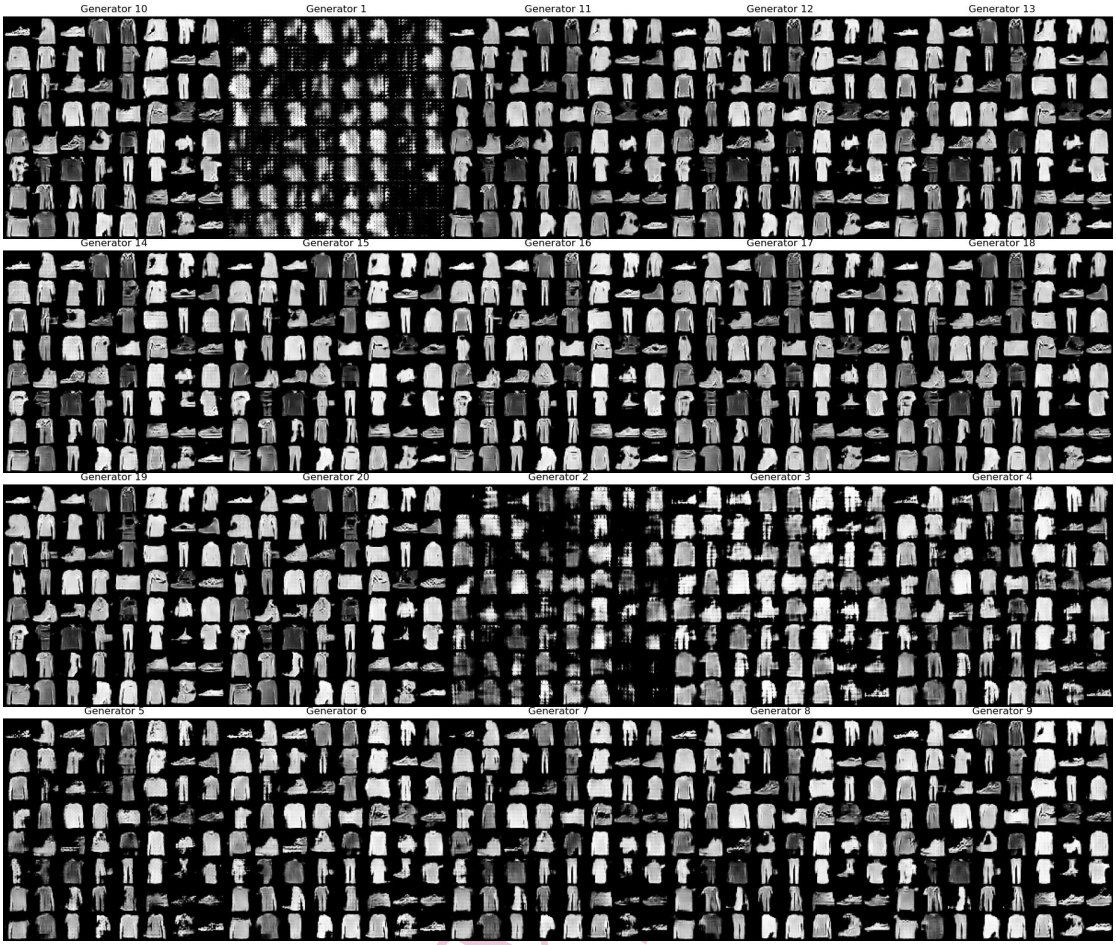

图 6: 卷积 GAN 前二十轮生成的图像

我们发现改进后的卷积 GAN 网络能更快的提升生成器的质量，仅仅 20 轮训练得到的图像质量已经比原始 GAN 的质量要高!

（五） 尝试改变随机数

老师要求我们尝试改变随机数查看对生成器结果的影响，这里我们挑选了在卷积 GAN 中表现最好的 8 种生成器，如图所示:

图 7: 表现最好的 8 种卷积生成器原始图像

我们将随机噪声增加到以前的 2 倍，观察到如下结果

图 8: 随机噪声变为原来 2 倍图像

我们观察到其亮度明显比以前更亮但是特征信息相比以前更少了, 现在我们把随机噪声减少为以前的 0.5 倍, 观察到如下结果

图 9: 随机噪声变为原来 0.5 倍图像

然后我们把随机噪声调整成以前的 0.5 倍最为直观的感受是亮度变暗了，所拥有的噪点也更多。当我们进一步考虑调低噪声比例时我们发现生成的图片质量迅速变低。总体来说：调小随机

噪声的比例，可能会使最后生成图片的亮度变低同时也会使得生成图像的噪点增多，调大随机数的比例会使图片亮度，但是图片的细节也会丢失更多主要是缺块的增大。

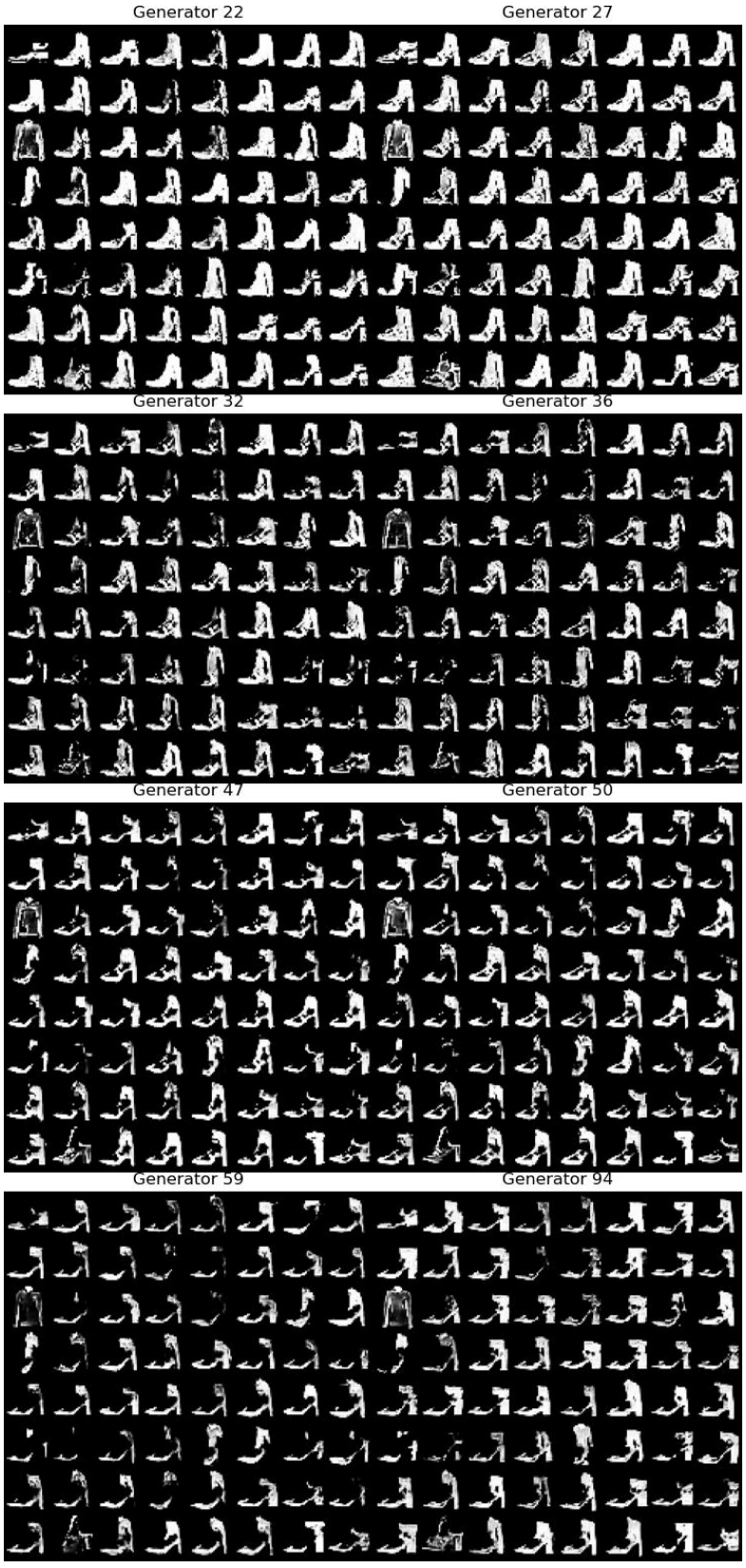

图 10: 随机噪声平移-3 图像

我们观察到当我们的随机噪声向一个方向偏移的比较大时，就会朝着某种固定的风格迁移。如上图所示：绝大部分图像都向高跟鞋方向偏移，如果是 +3 的话就会朝着 T 恤衫的方向偏移

图 11: 随机噪声加高斯噪声图像

加上高斯噪声后我们观察到部分图像出现了风格迁移，但是整体的风格并没有发生太大的变化。

四、 总结

本次实验成功实现生成对抗神经网络模型，并成功实现了卷积 GAN。通过这次实验我们对生成神经网络的结构有了更加深刻的理解。改变输入随机数是改变生成图片风格的重要方式，其中缩小噪声比例会导致生成图片的亮度变暗，产生更多黑噪点；放大噪声比例会使图片变亮，产生更多白噪点（缺失部分）。选择合适的噪声比例可能会让我们得到更好质量的图像。对随机噪声朝着某些方向进行偏移，也会使得生成的图片风格发生迁移。（具体如何偏移要看训练集的样本分布）。对随机噪声加高斯噪声可能会导致部分图片发生风格迁移。

总体而言：在高斯分布中噪声向量的标准差越大，生成样本的数量就会越多。极端情况下，均匀分布只会生成一种样本；生成器的网络结构对生成样本的数量也有一定影响。增加生成器的层数或神经元数量可能会增加生成样本的数量。通过增加噪声向量的维度，我们可以增加生成样本的变化空间，从而获得更多样的样本。