## 生成对抗网络实验报告

姓名：杨鑫 学号：2011028 专业：信息安全

一、实验要求

* 掌握GAN原理
* 学会使用PyTorch搭建GAN网络来训练FashionMNIST数据集

二、报告内容

* 老师提供的原始版本GAN网络结构（也可以自由调整网络）在FashionMNIST上的训练loss曲线，生成器和判别器的模型结构（print(G)、print(D)）
* 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）
* 格式不限

三、实验内容

**Fashion MNIST数据集**

|  |
| --- |
| FashionMNIST 是一个替代 MNIST 手写数字集的图像数据集。 它是由 Zalando（一家德国的时尚科技公司）旗下的研究部门提供。其涵盖了来自 10 种类别的共 7 万个不同商品的正面图片。  FashionMNIST 的大小、格式和训练集/测试集划分与原始的 MNIST 完全一致。60000/10000 的训练测试数据划分，28x28 的灰度图片。你可以直接用它来测试你的机器学习和深度学习算法性能，且不需要改动任何的代码。 |

**GAN生成对抗网络**

GAN（Generative Adversarial Networks）生成对抗网络是一种深度学习模型，由生成器和判别器两个部分组成。它的目的是生成与真实数据相似的新数据。生成器模型将随机噪声作为输入，并生成类似于真实数据的新数据。判别器模型接收真实数据和生成器生成的数据，并尝试区分哪些是真实的数据，哪些是生成的数据。两个模型相互竞争，通过反复迭代，生成器模型逐渐学会生成更接近真实数据的新数据，而判别器模型逐渐学会更准确地区分真实数据和生成的数据。

GAN模型的目标函数如下：



在这里，训练网络D使得最大概率地分对训练样本的标签（最大化log D(x)和），训练网络G最小化log(1 – D(G(z)))，即最大化D的损失。而训练过程中固定一方，更新另一个网络的参数，交替迭代，使得对方的错误最大化，最终，G 能估测出样本数据的分布，也就是生成的样本更加的真实。

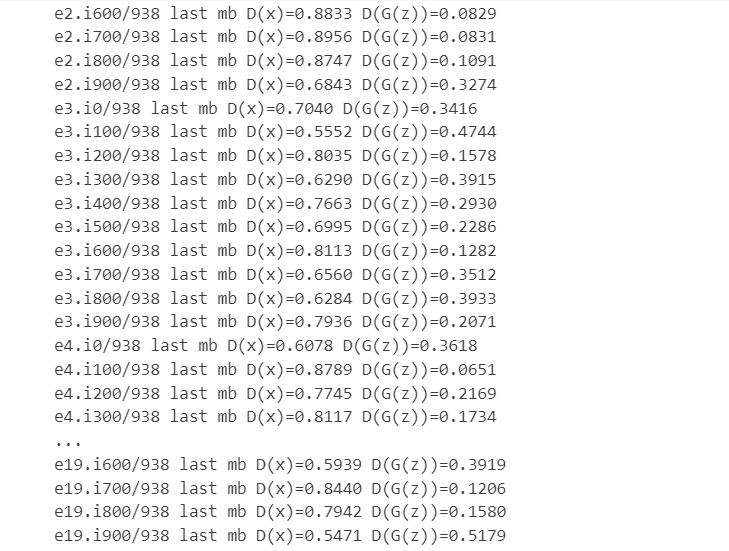
然后从式子中解释对抗，我们知道G网络的训练是希望趋近于1，也就是正类，这样G的loss就会最小。而D网络的训练就是一个2分类，目标是分清楚真实数据和生成数据，也就是希望真实数据的D输出趋近于1，而生成数据的输出即趋近于0，或是负类。这里就是体现了对抗的思想。

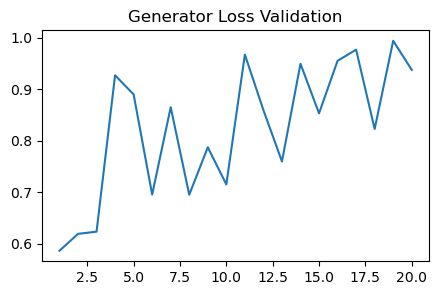
**网络结构：**

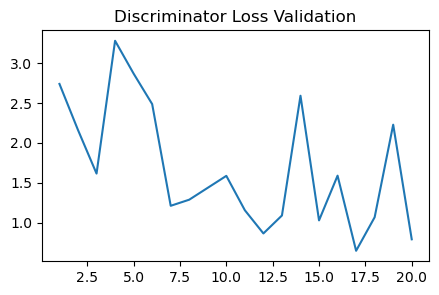
|  |
| --- |
| Discriminator(  (fc1): Linear(in\_features=784, out\_features=128, bias=True)  (nonlin1): LeakyReLU(negative\_slope=0.2)  (fc2): Linear(in\_features=128, out\_features=1, bias=True)  )  Generator(  (fc1): Linear(in\_features=100, out\_features=128, bias=True)  (nonlin1): LeakyReLU(negative\_slope=0.2)  (fc2): Linear(in\_features=128, out\_features=784, bias=True)  ) |

**训练损失函数曲线：**

当迭代次数为20的时候，使用本机的cpu进行训练：

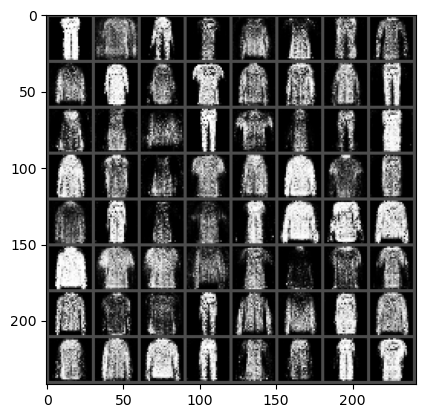






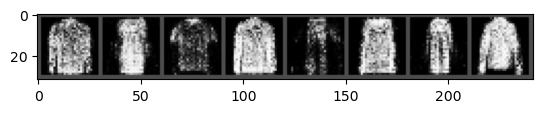
GAN的训练过程是一个非常复杂的最小化最大化博弈，生成器和判别器一直在进行对抗，因此损失函数非常杂乱，可能会出现不稳定的情况，导致GAN很难训练。此外，由于生成器和判别器的训练是相互竞争的，因此损失函数的变化可能会非常快，这也增加了GAN训练的挑战性。通常需要通过其他方法来评估GAN的训练效果，例如通过可视化生成的图像、计算生成图像与真实图像之间的差异、计算判别器的准确率等。这些方法可以提供更直观和准确的指标，帮助我们评估GAN的训练效果。

可以查看迭代二十次后生成器生成的图像，如下图，可以发现相比于刚开始具有很好的图像生成效果。



**自定义一组随机数，生成8张图**

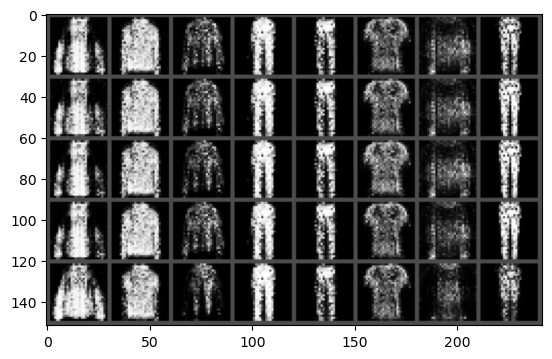
|  |
| --- |
| #自定义一组随机数，生成八张图  fixed\_noise = torch.randn(8, 100, device=device)  x\_gen = G(fixed\_noise)  show\_imgs(x\_gen, new\_fig=False) |



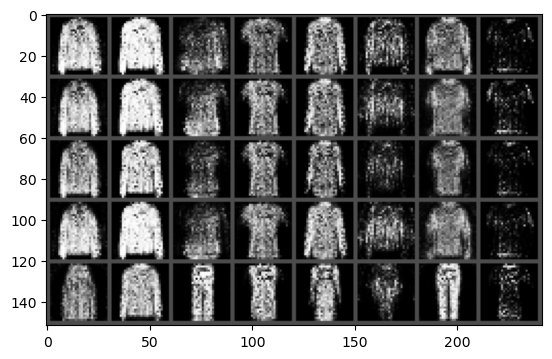
**针对自定义的 100 个随机数，自由挑选 5 个随机数，查看调整每个随机数时，生成图像的变化（每个随机数调整 3 次，共生成 15x8 张图），总结调整每个随机数时，生成图像发生的变化**

这里为了方便画图，进行了如下的操作：首先利用fixed\_noise = torch.randn(8, 100, device=device) 生成了八个100维的随机数作为生成图像所要使用的随机数。然后利用fixed\_noise = fixed\_noise.repeat(5,1) 将这八个随机数复制为5份作为我们要更改的五个位置。每次实验更改的随机数位置和数值都相同。这样，每次实验可以画出一张8x5的图，其中每行代表更改不同位置的随机数。每张图的第一行代表的是更改第4个位置的随机数，第二行是代表更改第25个位置的随机数，第三行代表的是更改第52个位置的随机数，第四行代表第78个位置的随机数，最后一行代表的是更改第93个位置的随机数(这里的位置的选择考虑随机性和充分性)。

|  |
| --- |
| fixed\_noise = torch.randn(8, 100, device=device)  fixed\_noise = fixed\_noise.repeat(5,1)  rand\_num = 0.5  for i in range(0, 8):      fixed\_noise[i][4] = rand\_num  for i in range(8, 16):      fixed\_noise[i][25] = rand\_num  for i in range(16, 24):      fixed\_noise[i][52] = rand\_num  for i in range(24, 32):      fixed\_noise[i][78] = rand\_num  for i in range(32, 40):      fixed\_noise[i][93] = rand\_num  x\_gen = G(fixed\_noise)  show\_imgs(x\_gen, new\_fig=False) |



1将随机数指定为0.5



2将随机数指定为3



3将随机数指定为10

从结果我们观察可以看到，当随机数设定较小的时候(0.3)，在不同位置改变这样小的一个随机数，效果是不大的。但是当随机数设置的较大的时候(10)，如果我们在不同位置改变一个这样的随机数，得到的效果是比较明显的，从最后一张图可以看出，相对于位置78变为10，位置93的改变直接让衣服变成了裤子。当我们在观察相同位置的随机数改变的时候，可以发现过小的随机数(0.3)或者过大的随机数(10)都会导致生成的图像的亮度较低，甚至有些图片接近于全黑，而当我们使用一个适中随机数的时候，我们的模型生成的效果相对来说较为明亮，且生成的图像效果较好。同时对于不同的随机数可以看到不同的随机数对于某一类的生成效果都不同，比如较小的随机数的时候对于右边裤子的生成效果好对于左边衣服的生成效果很差，然而对于随机数3来说对于左边衣服生成的效果比较好而对于右边裤子生成效果很差。

**用卷积实现生成器和判别器**

**DCGAN（Deep Convolutional Generative Adversarial Network）**是一种基于生成对抗网络（GAN）的图像生成模型。它的思路是通过训练一个生成器网络和一个判别器网络来实现逼真图像的生成。

DCGAN的核心思想是将卷积神经网络（CNN）引入到GAN的架构中，以便更好地处理图像数据。传统的GAN中使用的是全连接层，这在处理高维图像数据时存在一些问题，例如显存消耗大和模型难以收敛等。DCGAN通过使用卷积层和反卷积层，充分利用了图像的空间结构特征，提高了生成图像的质量和多样性。下面是DCGAN的一般思路和关键要点：

1. 生成器（Generator）：

* 生成器接收一个低维的随机噪声向量作为输入，通常是一个服从均匀分布或正态分布的向量。
* 利用反卷积层（转置卷积层）逐渐增加特征图的尺寸，从而将低维噪声转换为高维的图像数据。
* 在卷积层之间通常使用批归一化（Batch Normalization）来加速收敛并稳定训练过程。
* 使用修正线性单元（ReLU）或其他激活函数来引入非线性特征。
* 最后一层的激活函数通常使用Tanh函数，将生成的图像像素值范围映射到[-1, 1]之间。

1. 判别器（Discriminator）：

* 判别器接收一个图像作为输入，判别图像是真实图像还是生成器生成的假图像。
* 利用卷积层逐渐降低图像的尺寸，从而将图像数据转换为更高级的特征表示。
* 在卷积层之间通常使用批归一化和LeakyReLU激活函数，增加模型的表达能力和稳定性。
* 最后一层使用全连接层或卷积层输出一个概率值，表示输入图像是真实图像的概率。

1. 对抗训练：

* 生成器和判别器通过对抗训练来互相学习和优化。
* 在训练过程中，生成器生成假图像，判别器评估真实图像和生成的假图像的真实性，并给出相应的概率预测。
* 生成器的目标是通过生成逼真的假图像来欺骗判别器，使其将假图像误判为真实图像。
* 判别器的目标是准确地区分真实图像和生成的假图像，尽可能使真实图像被判别为真，假图像被判别为假。
* 生成器和判别器通过交替训练，使用梯度下降方法优化自己的参数，逐渐提高生成图像的质量和判别的准确性。

根据GCDAN的思路实现生成器：

|  |
| --- |
| class Generator(nn.Module):      def \_\_init\_\_(self, ngpu):          super(Generator, self).\_\_init\_\_()          self.ngpu = ngpu          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, input):          return self.main(input) |

判别器：

|  |
| --- |
| class Discriminator(nn.Module):      def \_\_init\_\_(self, ngpu):          super(Discriminator, self).\_\_init\_\_()          self.ngpu = ngpu          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),              # state size. (1) x 1 x 1              nn.Sigmoid()          )      def forward(self, input):          return self.main(input) |

网络结构如下：

|  |
| --- |
| Discriminator(  (main): Sequential(  (0): Conv2d(1, 64, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  (1): LeakyReLU(negative\_slope=0.2, inplace=True)  (2): Conv2d(64, 128, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  (4): LeakyReLU(negative\_slope=0.2, inplace=True)  (5): Conv2d(128, 256, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  (7): LeakyReLU(negative\_slope=0.2, inplace=True)  (8): Conv2d(256, 512, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  (10): LeakyReLU(negative\_slope=0.2, inplace=True)  (11): Conv2d(512, 1, kernel\_size=(4, 4), stride=(1, 1), bias=False)  (12): Sigmoid()  )  ) |