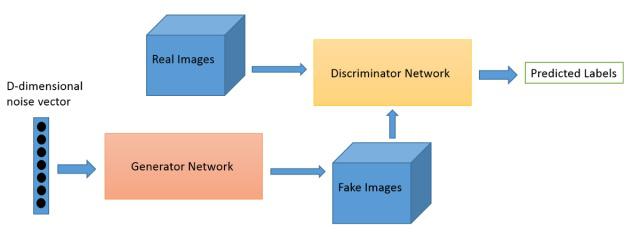
### 实验九 生成对抗网络实现

生成式对抗网络（[GAN](https://baike.baidu.com/item/GAN/22181905" \t "_blank), Generative Adversarial Networks）是一种[深度学习](https://baike.baidu.com/item/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0/3729729" \t "_blank)模型，是近年来复杂分布上[无监督学习](https://baike.baidu.com/item/%E6%97%A0%E7%9B%91%E7%9D%A3%E5%AD%A6%E4%B9%A0/810193)最具前景的方法之一。模型通过框架中（至少）两个模块：生成模型（Generative Model）和判别模型（Discriminative Model）的互相[博弈](https://baike.baidu.com/item/%E5%8D%9A%E5%BC%88/4669968" \t "_blank)学习产生相当好的输出。原始 GAN 理论中，并不要求 G 和 D 都是神经网络，只需要是能拟合相应生成和判别的函数即可。但实用中一般均使用深度神经网络作为 G 和 D 。一个优秀的GAN应用需要有良好的训练方法，否则可能由于神经网络模型的自由性而导致输出不理想。

**实验目的：**加深对生成对抗网络的理解，能够使用生成对抗网络解决简单问题

**实验原理：**在生成对抗网络中，一共建立了两个神经网络。第一个网络是典型的分类神经网络，称为判别器，我们训练这个网络对图像进行识别，以区别真假的图像。另一个网络称之为生成器，它将随机的噪声作为输入，输出经过生成器生成的图像，它的目的是混淆判别器，使其认为它生成的图像是真的。真的图片在训练集当中，而假的则不在。在这个设置中，两个网络参与了一场竞争游戏，并试图超越对方，同时，帮助对方完成自己的任务。经过数千次迭代后，如果一切顺利，生成器网络在生成逼真的假图像方面变得完美，而判别器网络在判断显示给它的图像是假的还是真的方面变得完美。

最基本的生成对抗网络为vanilla GAN网络。首先，从潜在空间采样D维噪声向量并馈送到生成器网络。生成器网络将该噪声向量转换为图像。然后将该生成的图像馈送到判别器网络以进行分类。判别器网络不断地从真实数据集和由生成器网络生成的图像获得图像。它的工作是区分真实和虚假的图像。所有GAN架构都遵循相同的设计。这是GAN的诞生。



原始论文中的目标公式如下：



上述这个公式就是一个最大最小优化问题，对应的也就是上述的两个优化过程。这个公式既然是最大最小的优化，那就不是一步完成的，其实对比我们的分析过程也是这样的，这里我们先优化D，然后再去优化G，本质上是两个优化问题，把目标公式拆解，就如同下面两个公式：

优化D：



优化G：



可以看到，优化D，也就是判别网络时，生成网络并没有参与，后面的G(z)相当于已经得到的假样本。优化D的公式的第一项，使的真样本x输入的时候，得到的结果越大越好，可以理解为需要真样本的预测结果越接近于1越好。对于假样本，需要优化是的其结果越小越好，也就是D(G(z))越小越好，因为它的标签为0。但是优化D的公式是取最大值，所以把第二项改成1-D(G(z))，这样就变成了取最大值，两者合起来也就是取最大值。同样在优化G的时候，不需要真样本，所以把第一项直接去掉了。这个时候只有假样本，但是这时候是希望假样本的标签是1的，所以是D(G(z))越大越好，但是为了统一成1-D(G(z))的形式，那么只能是最小化1-D(G(z))，本质上没有区别，只是为了形式的统一。之后这两个优化模型可以合并起来写，就变成了最开始的那个最大最小目标函数了。

**实验内容：**根据生成对抗网络的相关知识，使用Python语言实现一个简单的生成对抗网络。简单的程序示例如下：

import torch

from torch import nn

from torch.autograd import Variable

import torchvision.transforms as tfs

from torch.utils.data import DataLoader, sampler

from torchvision.datasets import MNIST

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec

%matplotlib inline

plt.rcParams['figure.figsize'] = (10.0, 8.0) # 设置画图的尺寸

plt.rcParams['image.interpolation'] = 'nearest'

plt.rcParams['image.cmap'] = 'gray'

def show\_images(images): # 定义画图工具

images = np.reshape(images, [images.shape[0], -1])

sqrtn = int(np.ceil(np.sqrt(images.shape[0])))

sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

fig = plt.figure(figsize=(sqrtn, sqrtn))

gs = gridspec.GridSpec(sqrtn, sqrtn)

gs.update(wspace=0.05, hspace=0.05)

for i, img in enumerate(images):

ax = plt.subplot(gs[i])

plt.axis('off')

ax.set\_xticklabels([])

ax.set\_yticklabels([])

ax.set\_aspect('equal')

plt.imshow(img.reshape([sqrtimg,sqrtimg]))

return

def preprocess\_img(x):

x = tfs.ToTensor()(x)

return (x - 0.5) / 0.5

def deprocess\_img(x):

return (x + 1.0) / 2.0

class ChunkSampler(sampler.Sampler): # 定义一个取样的函数

"""Samples elements sequentially from some offset.

Arguments:

num\_samples: # of desired datapoints

start: offset where we should start selecting from

"""

def \_\_init\_\_(self, num\_samples, start=0):

self.num\_samples = num\_samples

self.start = start

def \_\_iter\_\_(self):

return iter(range(self.start, self.start + self.num\_samples))

def \_\_len\_\_(self):

return self.num\_samples

NUM\_TRAIN = 50000

NUM\_VAL = 5000

NOISE\_DIM = 96

batch\_size = 128

train\_set = MNIST('./mnist', train=True, download=True, transform=preprocess\_img)

train\_data = DataLoader(train\_set, batch\_size=batch\_size, sampler=ChunkSampler(NUM\_TRAIN, 0))

val\_set = MNIST('./mnist', train=True, download=True, transform=preprocess\_img)

val\_data = DataLoader(val\_set, batch\_size=batch\_size, sampler=ChunkSampler(NUM\_VAL, NUM\_TRAIN))

imgs = deprocess\_img(train\_data.\_\_iter\_\_().next()[0].view(batch\_size, 784)).numpy().squeeze() # 可视化图片效果

show\_images(imgs)

def discriminator():

net = nn.Sequential(

nn.Linear(784, 256),

nn.LeakyReLU(0.2),

nn.Linear(256, 256),

nn.LeakyReLU(0.2),

nn.Linear(256, 1)

)

return net

def generator(noise\_dim=NOISE\_DIM):

net = nn.Sequential(

nn.Linear(noise\_dim, 1024),

nn.ReLU(True),

nn.Linear(1024, 1024),

nn.ReLU(True),

nn.Linear(1024, 784),

nn.Tanh()

)

return net

bce\_loss = nn.BCEWithLogitsLoss()

def discriminator\_loss(logits\_real, logits\_fake): # 判别器的 loss

size = logits\_real.shape[0]

true\_labels = Variable(torch.ones(size, 1)).float().cuda()

false\_labels = Variable(torch.zeros(size, 1)).float().cuda()

loss = bce\_loss(logits\_real, true\_labels) + bce\_loss(logits\_fake,

false\_labels)

return loss

def generator\_loss(logits\_fake): # 生成器的 loss

size = logits\_fake.shape[0]

true\_labels = Variable(torch.ones(size, 1)).float().cuda()

loss = bce\_loss(logits\_fake, true\_labels)

return loss

# 使用 adam 来进行训练，学习率是 3e-4, beta1 是 0.5, beta2 是 0.999

def get\_optimizer(net):

optimizer = torch.optim.Adam(net.parameters(), lr=3e-4, betas=(0.5, 0.999))

return optimizer

def train\_a\_gan(D\_net, G\_net, D\_optimizer, G\_optimizer, discriminator\_loss,

generator\_loss, show\_every=250, noise\_size=96, num\_epochs=10):

iter\_count = 0

for epoch in range(num\_epochs):

for x, \_ in train\_data:

bs = x.shape[0]

# 判别网络

real\_data = Variable(x).view(bs, -1).cuda() # 真实数据

logits\_real = D\_net(real\_data) # 判别网络得分

# -1 ~ 1 的均匀分布

sample\_noise = (torch.rand(bs, noise\_size) - 0.5) / 0.5

g\_fake\_seed = Variable(sample\_noise).cuda()

fake\_images = G\_net(g\_fake\_seed) # 生成的假的数据

logits\_fake = D\_net(fake\_images) # 判别网络得分

# 判别器的 loss

d\_total\_error = discriminator\_loss(logits\_real, logits\_fake)

D\_optimizer.zero\_grad()

d\_total\_error.backward()

D\_optimizer.step() # 优化判别网络

# 生成网络

g\_fake\_seed = Variable(sample\_noise).cuda()

fake\_images = G\_net(g\_fake\_seed) # 生成的假的数据

gen\_logits\_fake = D\_net(fake\_images)

g\_error = generator\_loss(gen\_logits\_fake) # 生成网络的 loss

G\_optimizer.zero\_grad()

g\_error.backward()

G\_optimizer.step() # 优化生成网络

if (iter\_count % show\_every == 0):

print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter\_count,

d\_total\_error.data[0], g\_error.data[0]))

imgs\_numpy = deprocess\_img(fake\_images.data.cpu().numpy())

show\_images(imgs\_numpy[0:16])

plt.show()

print()

iter\_count += 1

D = discriminator().cuda()

G = generator().cuda()

D\_optim = get\_optimizer(D)

G\_optim = get\_optimizer(G)

train\_a\_gan(D, G, D\_optim, G\_optim, discriminator\_loss, generator\_loss)

**实验要求：**

1. 下载MNIST数据集。
2. 创建生成对抗网络。
3. 使用MNIST数据集训练生成对抗网络，使网络生成MNIST假数据。