

认知科学与类脑计算

实验报告八

日期 : 2019/5/31

班级 : 16 人工智能

姓名 : 贾乘兴

学号 : 201600301304

一.需求分析 :

生成式对抗网络 (GAN, Generative Adversarial Networks) 是一种深度学习模型, 是近年来复杂分布上无监督学习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) 和判别模型 (Discriminative) 的互相博弈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。本次实验的目的是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

二.概要设计 :

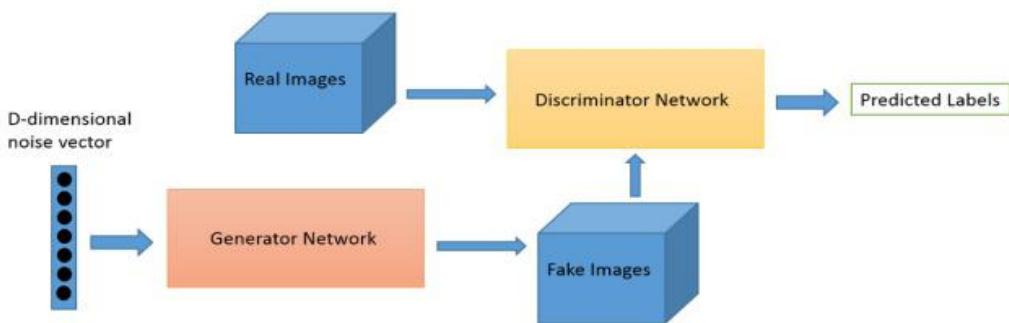
下载 MNIST 数据集, 创建生成对抗网络, 使用 MNIST 数据集训练生成对抗网络, 使网络生成 MNIST 假数据。同时使用其他 gan 的网络结构和其它数据集进行测试

三.详细设计：

1.GAN

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：

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)} [\log(D(x))] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

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：

$$\max_D V(D, G) = E_{x \sim p_{data}(x)} [\log(D(x))] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

优化 G：

$$\min_G V(D, G) = E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

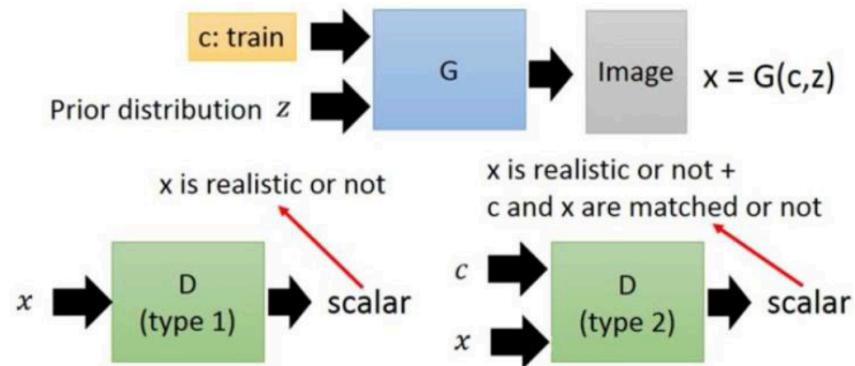
可以看到，优化 D，也就是判别网络时，生成网络并没有参与，后面的 G(z)相当于已经得到的假样本。优化 D 的公式的第一项，使的真样本 x 输入的时候，得到的结果越大越好，可以理解为需要真样本的预测结果越接近于 1 越好。对于假样本，需要优化的是其结果越小越好，也就是 D(G(z))越小越好，因为它的标签为 0。但是优化 D 的公式是取最大值，所以把第二项改成 1-D(G(z))，这样就变成了取最大值，两者合起来也就是取最大值。同样在优化 G 的时候，不需要真样本，所以把第一项直接去掉了。这个时候只有假样本，但是这时候是希望假样本的标签是 1 的，所以是 D(G(z))越大越好，但是为了统一成 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。

2.condition-GAN

在原有的 gan 的基础上，加入了条件概率，公式计算如下

$$\max_D \{ \mathbb{E}_{\sim P_{data}} \log D(G(x|y)) + \mathbb{E}_{x \sim P_G} \log (1 - (D(x|y))) \}$$

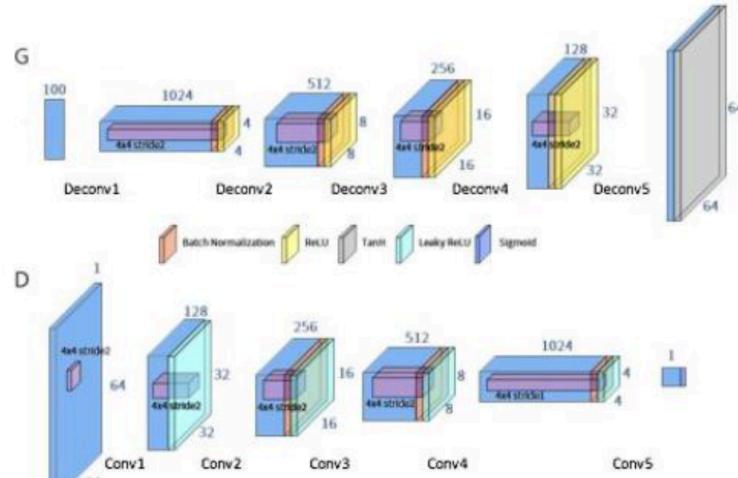
其结构如下



在实际的计算中，对条件的加入则较为简单，直接对输入进行拼接

3.DCGAN

在判别模型和生成模型上都是用卷积神经网络，结构如下



4.图像归一化处理

标准化处理：

$$\text{standardization} = \frac{x - \mu}{\max\left(\sigma, \frac{1.0}{\sqrt{N}}\right)}$$

μ 表示图像均值， N 表示图像像素个数， σ 表示标准差。

归一化处理：

$$\text{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

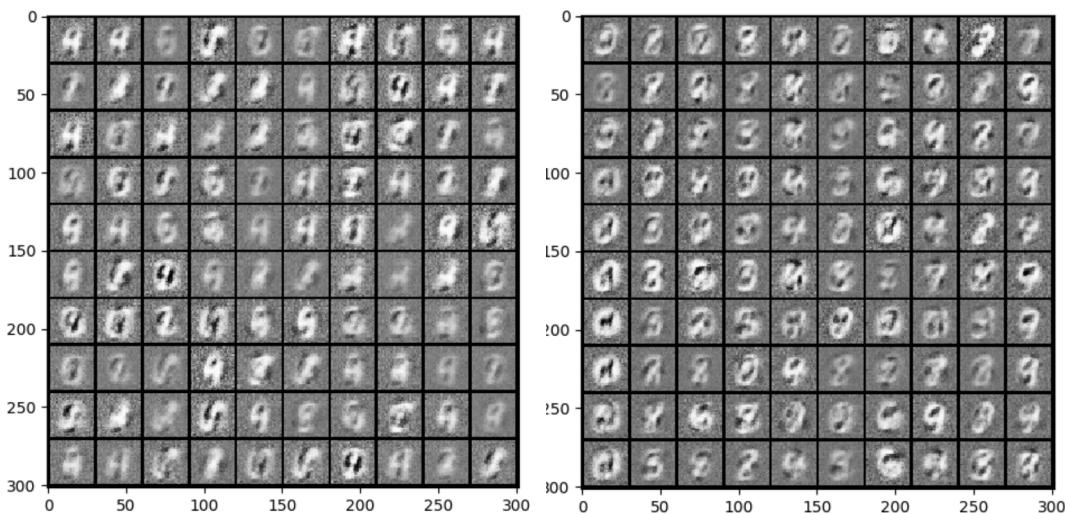
其中， \min ， \max 分别是图像 x 的像素最小，最大值。

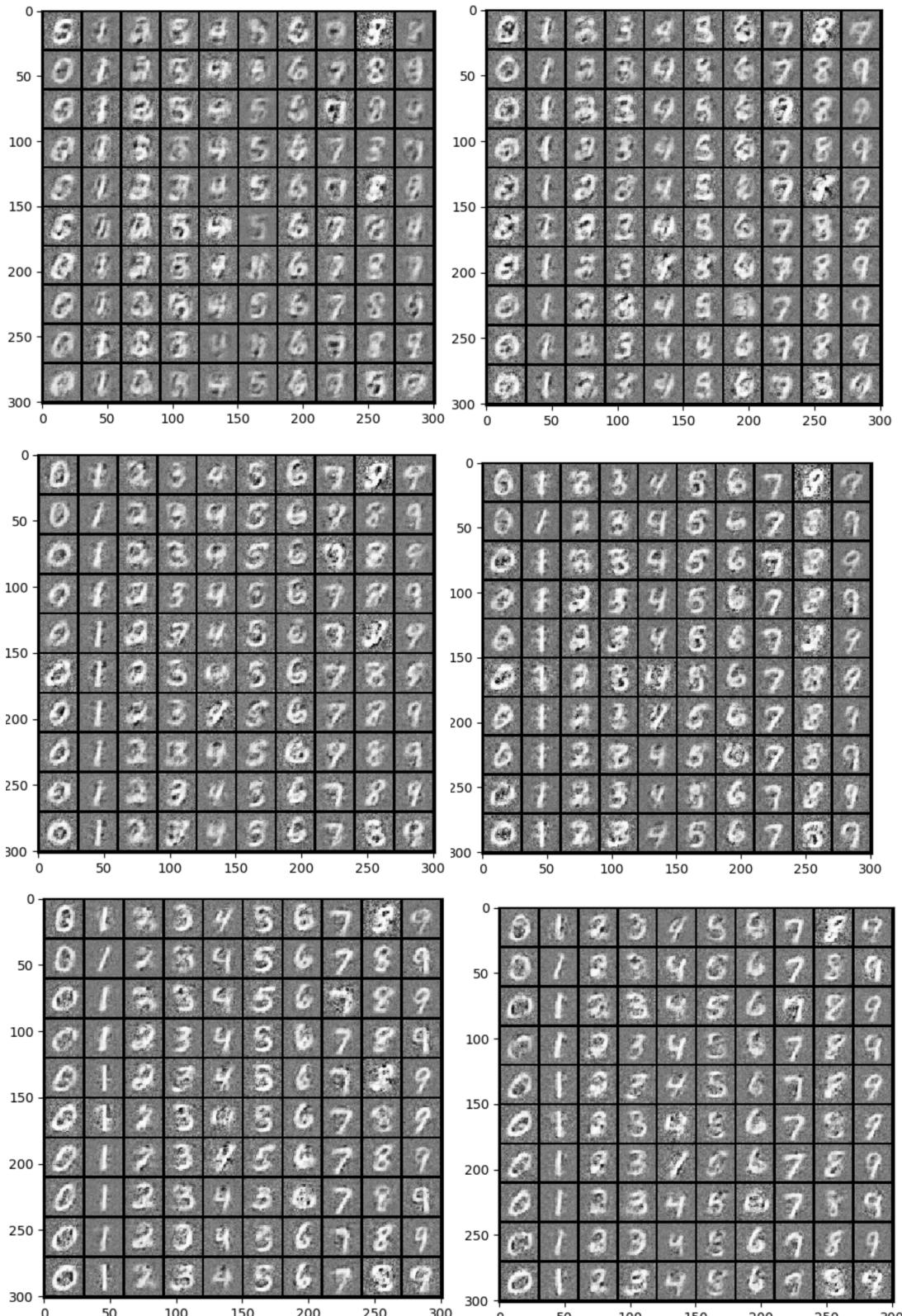
四. 调试分析：

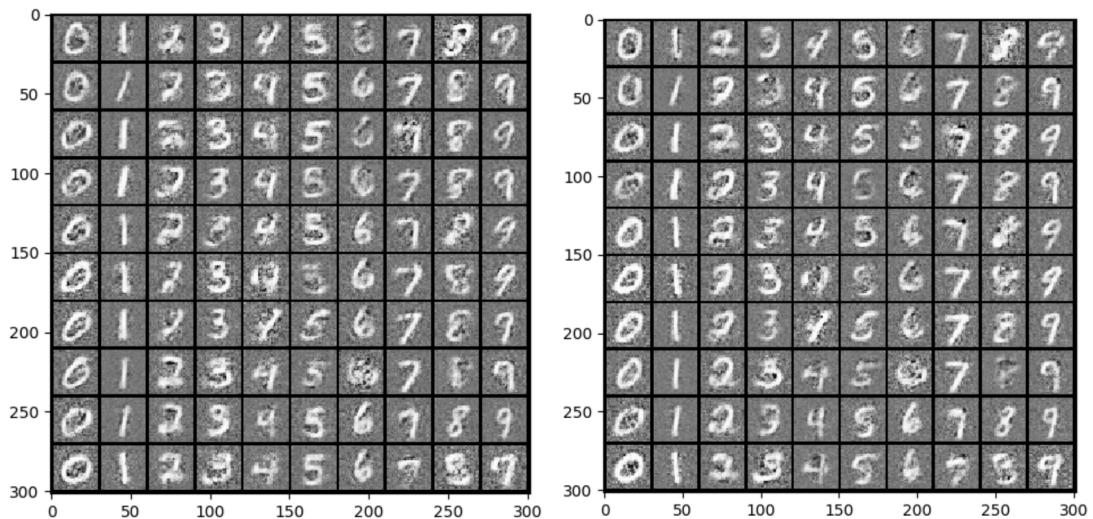
首先使用了 mnist 数据集进行生成，然后使用了相同的网络结构，使用了 fashion-mnist 数据集，该数据集在大小、类别等与 mnist 没有区别

五. 测试结果：

mnist 的生成结果如下（10 个 epoch）

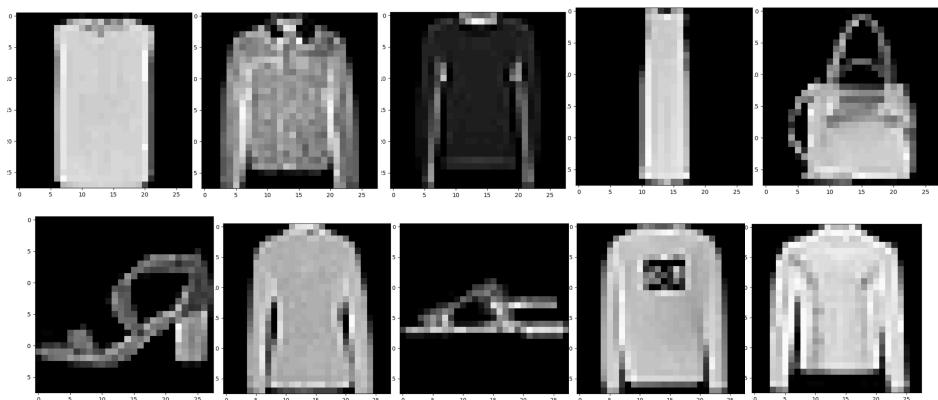




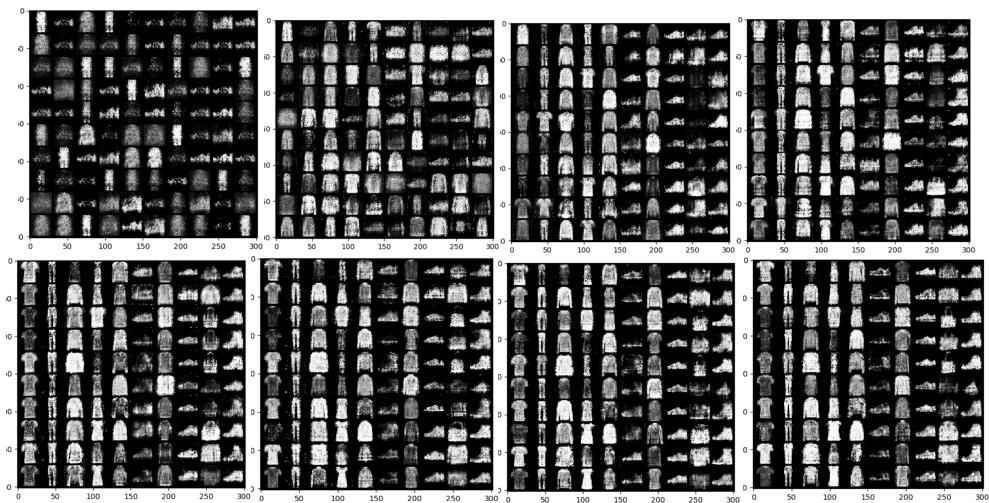


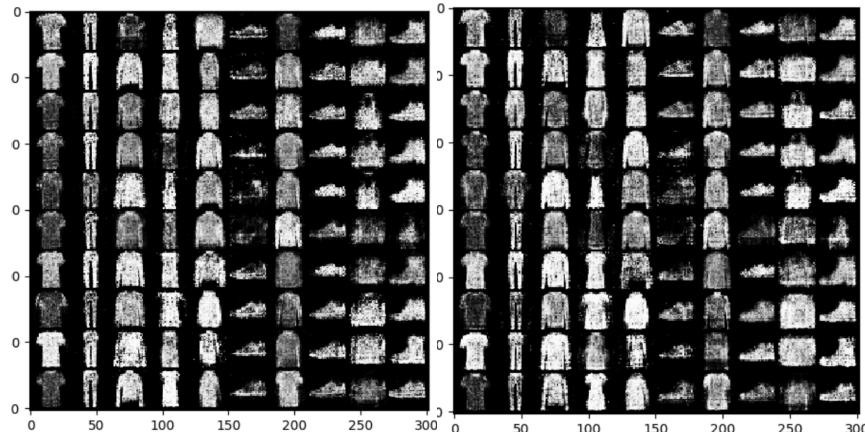
使用 fashion-mnist 数据集的效果如下

对原始数据的随机取样可视化如下



10 个 epoch 生成的图像如下





可见生成效果也较好

六.附录：

附录代码如下 文件为 condition_gan.py, cgan.py

```
condition_gan.py
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import save_image

import numpy as np
import os
from torchvision.utils import make_grid
import matplotlib.pyplot as plt

# 超参数
gpu_id = None
if gpu_id is not None:
    os.environ['CUDA_VISIBLE_DEVICES'] = gpu_id
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
if os.path.exists('cgan_images') is False:
    os.makedirs('cgan_images')
```

```

z_dim = 100 #paras.z_dim
batch_size = 100 #paras.batch_size
learning_rate = 0.0003 #paras.learning_rate
total_epochs = 10 #paras.total_epochs

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(794, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x, c):
        x = x.view(x.size(0), 784)
        x = torch.cat([x, c], 1)
        out = self.model(x)
        return out.squeeze()

class Generator(nn.Module):
    def __init__(self, z_dim):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(10 + z_dim, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, z, c):

```

```
z = z.view(z.size(0), z_dim)
x = torch.cat([z, c], 1)
out = self.model(x)
return out.view(x.size(0), 28, 28)

def one_hot(labels, class_num):
    batch_size = labels.size(0)
    one_hot_label = torch.zeros(batch_size, class_num).scatter_(1,
labels.reshape(-1, 1), 1)
    return one_hot_label

# 初始化构建判别器和生成器
discriminator = Discriminator().to(device)
generator = Generator(z_dim=z_dim).to(device)

# 初始化二值交叉熵损失
bce = torch.nn.BCELoss().to(device)
ones = torch.ones(batch_size).to(device)
zeros = torch.zeros(batch_size).to(device)

# 初始化优化器，使用Adam 优化器
g_optimizer = optim.Adam(generator.parameters(), lr=learning_rate,
betas=[0.5, 0.999])
d_optimizer = optim.Adam(discriminator.parameters(), lr=learning_rate,
betas=[0.5, 0.999])

# 加载 fashion 数据集

#####
train_dataset = datasets.MNIST(root='./data/', train=True,
transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data/', train=False,
transform=transforms.ToTensor())

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)

#####
#用于生成效果图
```

```

# 生成 100 个 one-hot 向量, 每类 10 个
fixed_c = torch.FloatTensor(100, 10).zero_()
fixed_c = fixed_c.scatter_(dim=1,
index=torch.LongTensor(np.array(np.arange(0, 10).tolist()*10).reshape([100, 1])), value=1)
fixed_c = fixed_c.to(device)
# 生成 100 个随机噪声向量
fixed_z = torch.randn([100, z_dim]).to(device)

# 开始训练, 一共训练 total_epochs
for epoch in range(total_epochs):

    # 在训练阶段, 把生成器设置为训练模式; 对应于后面的, 在测试阶段, 把生成器设置为测试模式
    generator = generator.train()

    # 训练一个 epoch
    for i, data in enumerate(train_loader):

        # 加载真实数据
        #####
        img, label = data
        #####

        # 把对应的标签转化成 one-hot 类型
        #####
        label = one_hot(label.reshape(-1,1), 10)
        #####

        # 生成数据
        # 用正态分布中采样 batch_size 个随机噪声
        z = torch.randn([batch_size, z_dim]).to(device)
        # 生成 batch_size 个 one-hot 标签
        c = torch.FloatTensor(batch_size, 10).zero_()
        c = c.scatter_(dim=1, index=torch.LongTensor(np.array(np.arange(0, 10).tolist() * 10).reshape([batch_size, 1])), value=1)
        c = c.to(device)
        # 生成数据

        # your code
        fake_img = generator(z, c)

        # 计算判别器损失, 并优化判别器
        #####

```

```

real_prob = discriminator(img, label)
real_loss = bce(real_prob, ones)

fake_prob = discriminator(fake_img, c)
fake_loss = bce(fake_prob, zeros)

d_optimizer.zero_grad()
d_loss = real_loss + fake_loss
d_loss.backward()
d_optimizer.step()
#####
# 计算生成器损失, 并优化生成器

#####
g_optimizer.zero_grad()
z = torch.randn(batch_size, z_dim)
c = torch.LongTensor(np.random.randint(0, 10, batch_size))
c = one_hot(c, 10)
fake_img = generator(z, c)
prob = discriminator(fake_img, c)
g_loss = bce(prob, torch.ones(batch_size))
g_loss.backward()
g_optimizer.step()

#####
# 输出损失 参考下方 print
print("[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]" % (epoch,
total_epochs, i, len(train_loader), d_loss.item(), g_loss.item()))

# 把生成器设置为测试模型, 生成效果图并保存
generator = generator.eval()
fixed_fake_images = generator(fixed_z, fixed_c)
grid = make_grid(fixed_fake_images.unsqueeze(1).data, nrow=10,
normalize=True).permute(1, 2, 0).numpy()
plt.imshow(grid)
plt.show()

#save_image(fixed_fake_images, 'cgan_images/{}.png'.format(epoch),
nrow=10, normalize=True)

```

cgan.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import Dataset
from torchvision.utils import save_image

import numpy as np
import os
# import paras
import pandas as pd
from PIL import Image
from torchvision.utils import make_grid
import matplotlib.pyplot as plt

# 超参数
gpu_id = None
if gpu_id is not None:
    os.environ['CUDA_VISIBLE_DEVICES'] = gpu_id
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
if os.path.exists('cgan_images') is False:
    os.makedirs('cgan_images')

z_dim = 100 #paras.z_dim
batch_size = 100 #paras.batch_size
learning_rate = 0.0003 #paras.learning_rate
total_epochs = 1 #paras.total_epochs

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(794, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
```

```

        nn.Sigmoid()
    )

def forward(self, x, c):
    x = x.view(x.size(0), 784)
    x = torch.cat([x, c], 1)
    out = self.model(x)
    return out.squeeze()

class Generator(nn.Module):
    def __init__(self, z_dim):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(10 + z_dim, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, z, c):
        z = z.view(z.size(0), z_dim)
        x = torch.cat([z, c], 1)
        out = self.model(x)
        return out.view(x.size(0), 28, 28)

def one_hot(labels, class_num):
    batch_size = labels.size(0)
    one_hot_label = torch.zeros(batch_size, class_num).scatter_(1,
    labels.reshape(-1, 1), 1)
    return one_hot_label

# 初始化构建判别器和生成器
discriminator = Discriminator().to(device)
generator = Generator(z_dim=z_dim).to(device)

# 初始化二值交叉熵损失
bce = torch.nn.BCELoss().to(device)

```

```

ones = torch.ones(batch_size).to(device)
zeros = torch.zeros(batch_size).to(device)

# 初始化优化器，使用Adam 优化器
g_optimizer = optim.Adam(generator.parameters(), lr=learning_rate,
betas=[0.5, 0.999])
d_optimizer = optim.Adam(discriminator.parameters(), lr=learning_rate,
betas=[0.5, 0.999])

# 加载 fashion 数据集

#####
class FashionMNIST(Dataset):
    def __init__(self, transform=None):
        self.transform = transform
        fashion_df = pd.read_csv('fashionmnist/fashion-mnist_train.csv')
        self.labels = fashion_df.label.values
        self.images = fashion_df.iloc[:, 1: ].values.astype('uint8').reshape(-1,
28, 28)

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        label = self.labels[idx]
        img = self.images[idx]
        img = Image.fromarray(self.images[idx])

        if self.transform:
            img = self.transform(img)

        return img[0].resize(1, 28, 28), label

    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(x: x.repeat(3,1,1)),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
    ])
    dataset = FashionMNIST(transform=transform)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
shuffle=True)

def show_sample(idx):
    sample = np.array(dataset[idx][0])

```

```
sample = torch.from_numpy(sample)
grid = make_grid(sample.unsqueeze(1).data, nrow=1,
normalize=True).permute(1, 2, 0).numpy()
plt.imshow(grid)
plt.show()

import random
for i in range(10):
    a = random.randint(0, 10000)
    show_sample(a)
#####
#用于生成效果图
# 生成 100 个 one-hot 向量, 每类 10 个
fixed_c = torch.FloatTensor(100, 10).zero_()
fixed_c = fixed_c.scatter_(dim=1,
index=torch.LongTensor(np.array(np.arange(0, 10).tolist())*10).reshape([100,
1])), value=1)
fixed_c = fixed_c.to(device)
# 生成 100 个随机噪声向量
fixed_z = torch.randn([100, z_dim]).to(device)

# 开始训练, 一共训练 total_epochs
for epoch in range(total_epochs):

    # 在训练阶段, 把生成器设置为训练模式; 对应于后面的, 在测试阶段, 把生成器设置为测试模式
    generator = generator.train()

    # 训练一个 epoch
    for i, data in enumerate(dataloader):

        # 加载真实数据
        #####
        img, label = data
        #####
        # 把对应的标签转化成 one-hot 类型
        #####
        label = one_hot(label, 10)
        #####
        # 生成数据
        # 用正态分布中采样 batch_size 个随机噪声
        z = torch.randn([batch_size, z_dim]).to(device)
```

```

# 生成 batch_size 个 one-hot 标签
c = torch.FloatTensor(batch_size, 10).zero_()
c = c.scatter_(dim=1, index=torch.LongTensor(np.array(np.arange(0,
10).tolist() * 10).reshape([batch_size, 1])), value=1)
c = c.to(device)
# 生成数据

#your code
fake_img = generator(z, c)

# 计算判别器损失, 并优化判别器

#####
real_prob = discriminator(img, label)
real_loss = bce(real_prob, ones)

fake_prob = discriminator(fake_img, c)
fake_loss = bce(fake_prob, zeros)

d_optimizer.zero_grad()
d_loss = real_loss + fake_loss
d_loss.backward()
d_optimizer.step()
#####

# 计算生成器损失, 并优化生成器

#####
g_optimizer.zero_grad()
z = torch.randn(batch_size, z_dim)
c = torch.LongTensor(np.random.randint(0, 10, batch_size))
c = one_hot(c, 10)
fake_img = generator(z, c)
prob = discriminator(fake_img, c)
g_loss = bce(prob, torch.ones(batch_size))
g_loss.backward()
g_optimizer.step()

#####

# 输出损失 参考下方 print
print("[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]" % (epoch,
total_epochs, i, len(dataloader), d_loss.item(), g_loss.item()))

```

```
# 把生成器设置为测试模型，生成效果图并保存
generator = generator.eval()
fixed_fake_images = generator(fixed_z, fixed_c)
grid = make_grid(fixed_fake_images.unsqueeze(1).data, nrow=10,
normalize=True).permute(1, 2, 0).numpy()
plt.imshow(grid)
plt.show()

#save_image(fixed_fake_images, 'cgan_images/{}.png'.format(epoch),
nrow=10, normalize=True)
```