计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：RNN 文本生成		学号：201600122057
日期：2019.5.20	班级：　智能 16	姓名：　贝仕成
Email：337263318@qq.com		

实验目的：运用 fashion MNIST 的数据集，采用 conditional Gan 的方式，将其中的 generator 和 discriminator 换成 CNN（即 DCGAN 思想），训练网络使其能够生成衣物图像。

实验软件和硬件环境：

Python 3.5.4
Jupyter notebook 5.0.0
神舟战神 Z7M-KP7S1 Windows10 16GRAM
NVIDIA GTX1050Ti

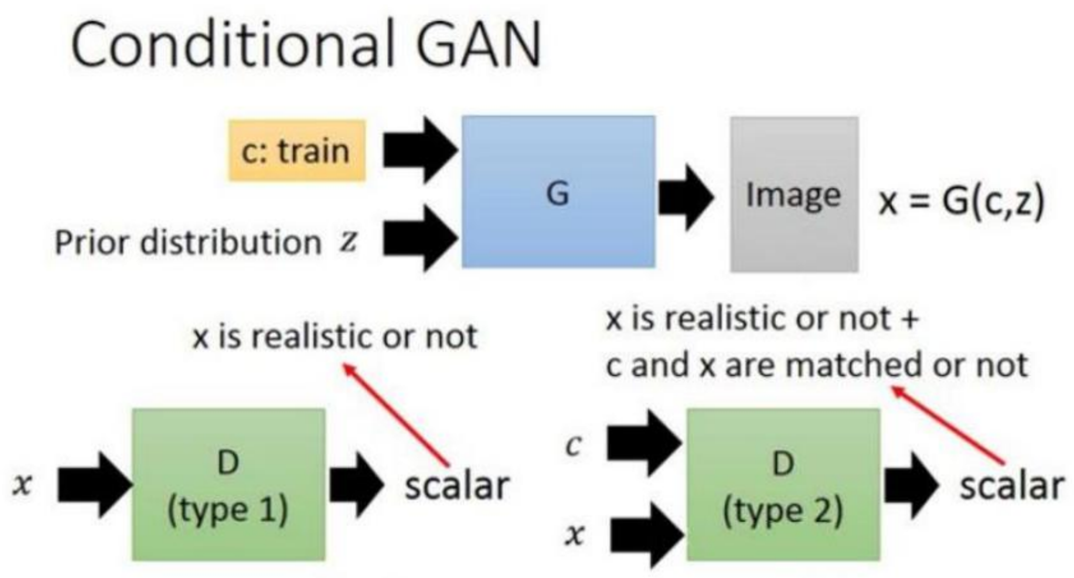
实验原理和方法：

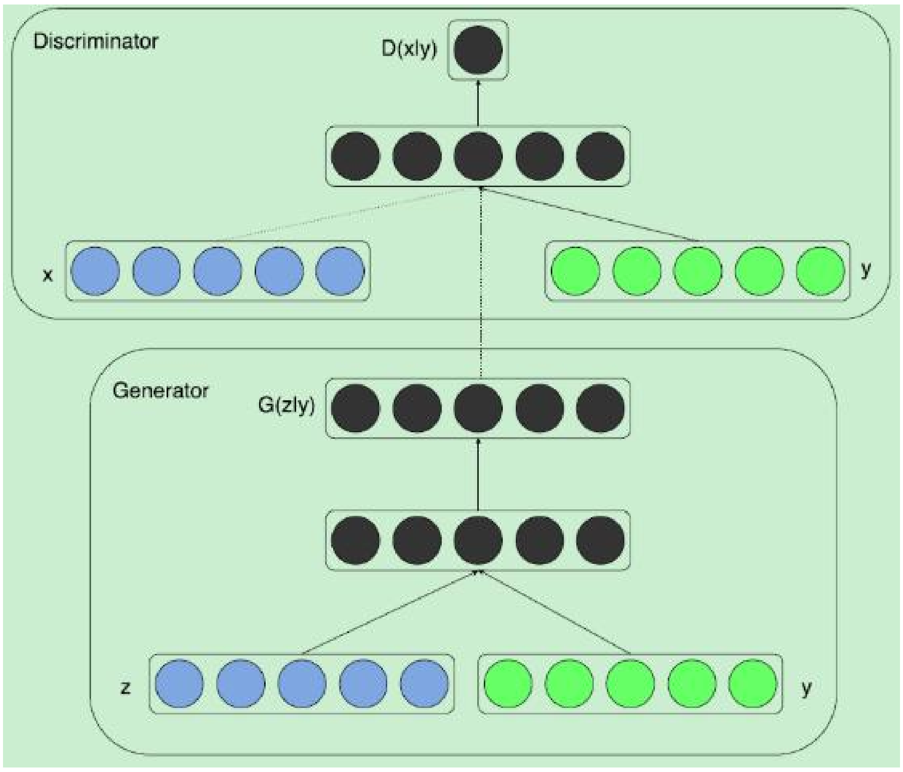
cGAN:

conditional GAN 是最早的 GAN 的变种之一，把原生 GAN 中的概率全改成条件概率，

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

这个条件可以使图片，标注等，结构如下图：

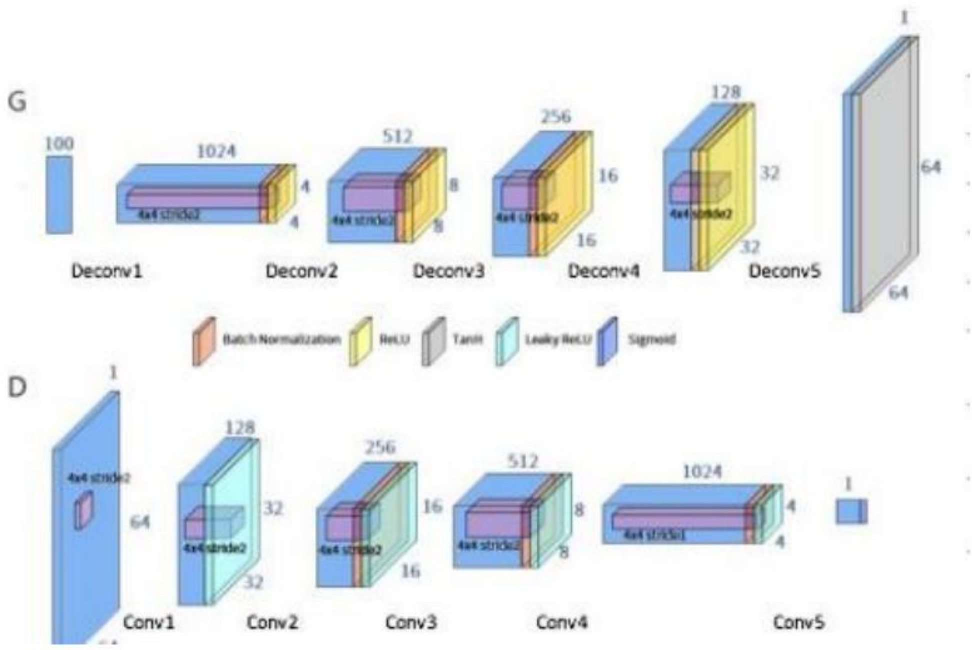




这个条件 c（y）可以是任何类型的数据，目的是有条件地监督生成器生成的数据，使得生成器生成结果的方式不是完全自由无监督的。

DCGAN:

同样的，GAN 中的 generator 和 discriminator 也可以换成 CNN，最先提出这个思想的就是 DCGAN。G 和 D 的结构如下：



图像标准化处理：

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

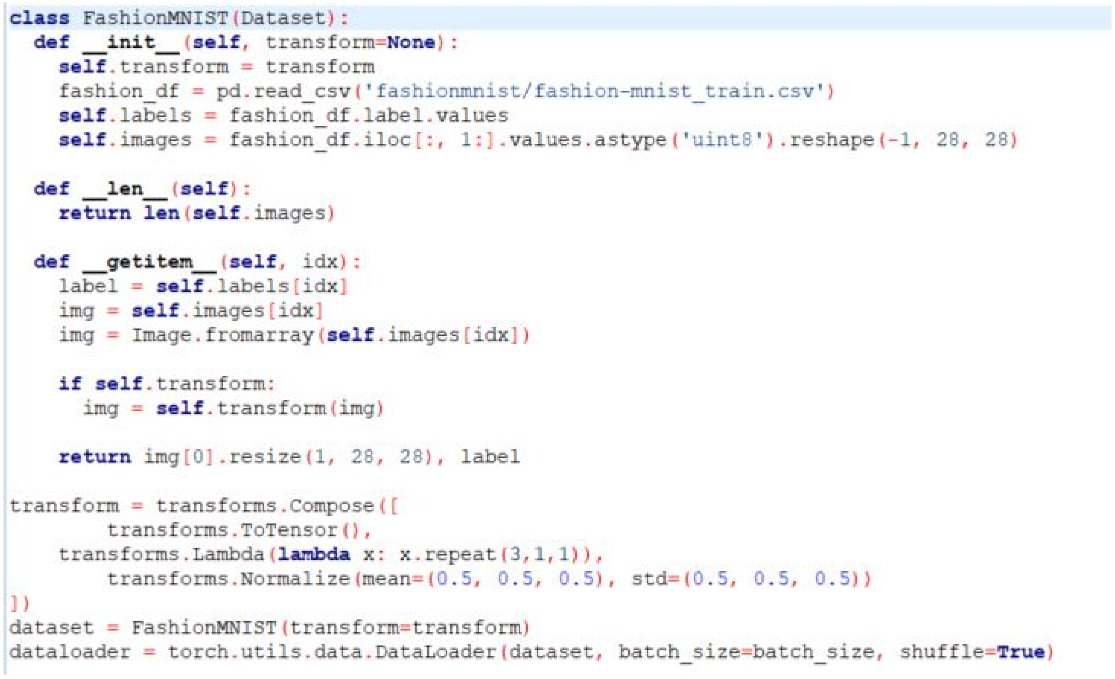
图像归一化处理：

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

实验步骤：（不要求罗列完整源代码）

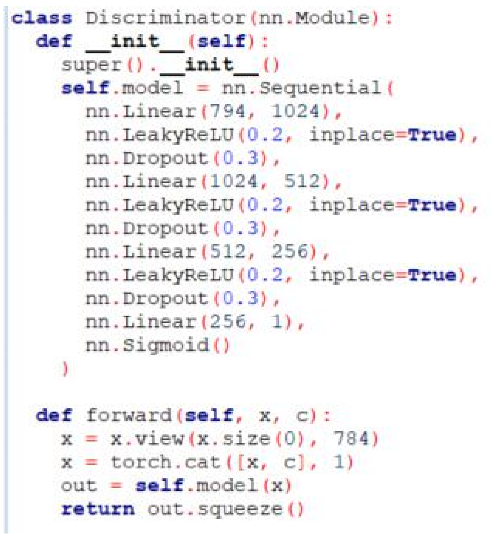
1. 补完 cgan.py

①读取数据并构建数据集

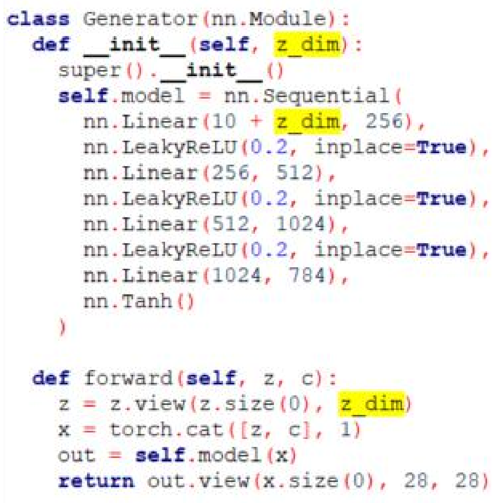


```
class FashionMNIST(Dataset):
  def __init__(self, transform=None):
    self.transform = transform
    fashion_df = pd.read_csv('fashionmnist/fashion-mnist_train.csv')
    self.labels = fashion_df.label.values
    self.images = fashion_df.iloc[:, 1:].values.astype('uint8').reshape(-1, 28, 28)

  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(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)
```

②编写 Generation 模块与 Discrimination 模块



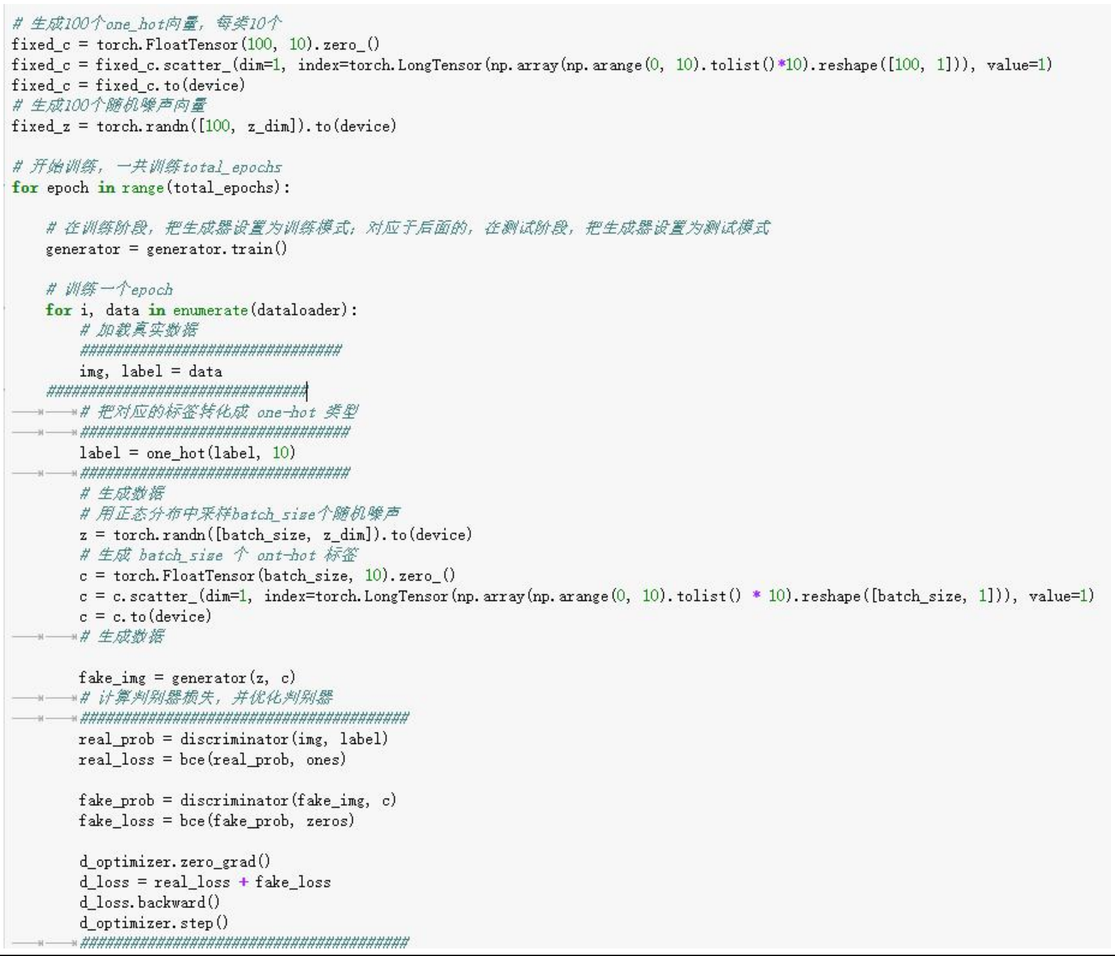
```
class Discriminator(nn.Module):
  def __init__(self):
    super().__init__()
    self.model = nn.Sequential(
      nn.Linear(794, 1024),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Dropout(0.3),
      nn.Linear(1024, 512),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Dropout(0.3),
      nn.Linear(512,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



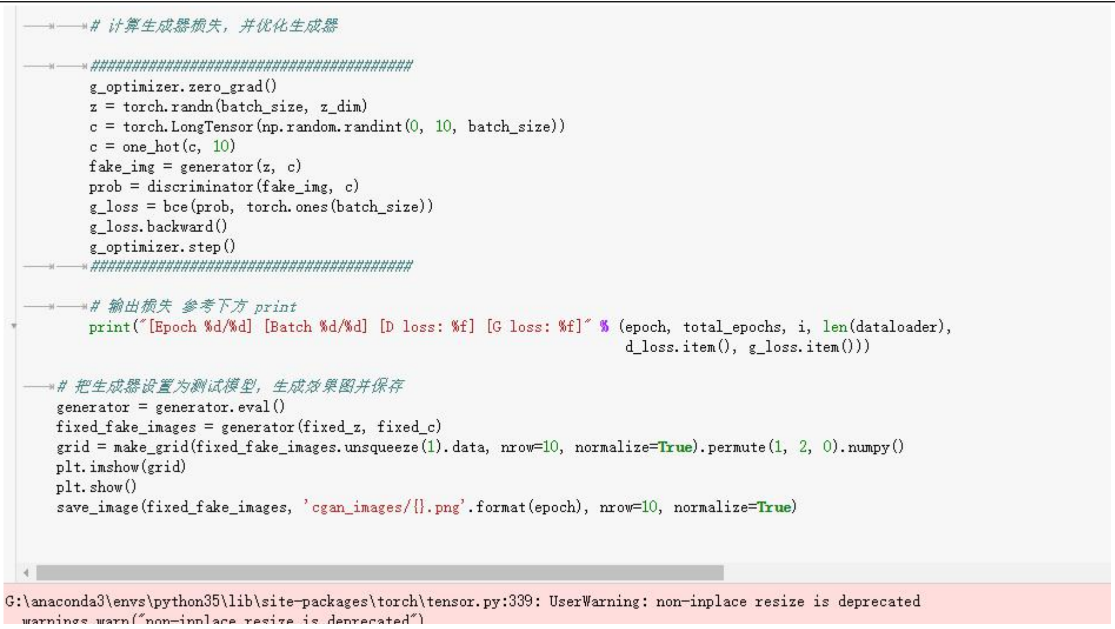
```
class Generator(nn.Module):
  def __init__(self, z_dim):
    super().__init__()
    self.model = nn.Sequential(
      nn.Linear(10 + z_dim, 256),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Linear(256, 512),
      nn.LeakyReLU(0.2, inplace=True),
      nn.Linear(512, 1024),
      nn.LeakyReLU(0.2,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

③完成训练代码

1. 补充代码



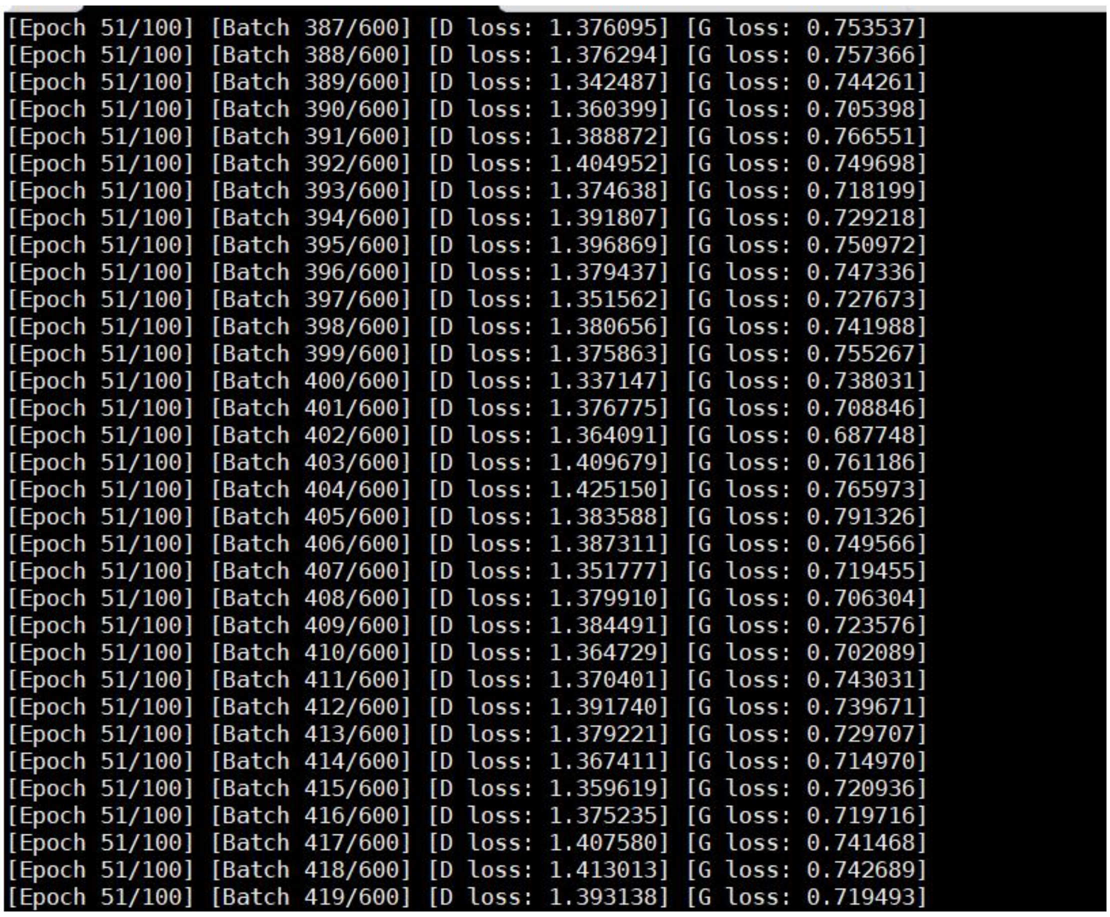
```
# 生成100个one_hot向量，每类10个
fixed_c = torch.FloatTensor(100, 10).zero_()
fixed_c = fixed_c.scatter_(dim=1, index=torch.LongTensor(np.array(np.arange(0, 10).tolist()*10).reshape([100, 1])), value=1)
fixed_c = fixed_c.to(device)
# 生成100个随机噪声向量
fixed_z = torch.randn([100, z_dim]).to(device)

# 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 ont-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

        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)
```

```
G:\anaconda3\envs\python35\lib\site-packages\torch\tensor.py:339: UserWarning: non-inplace resize is deprecated
  warnings.warn("non-inplace resize is deprecated")
```

2. 实验结果：

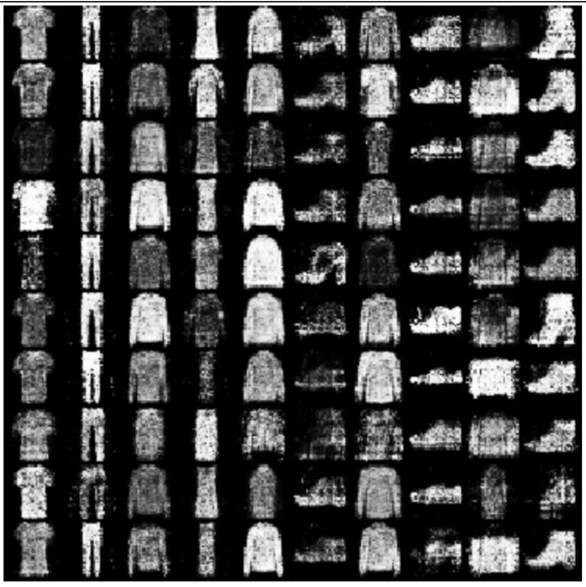


```
[Epoch 51/100] [Batch 387/600] [D loss: 1.376095] [G loss: 0.753537]
[Epoch 51/100] [Batch 388/600] [D loss: 1.376294] [G loss: 0.757366]
[Epoch 51/100] [Batch 389/600] [D loss: 1.342487] [G loss: 0.744261]
[Epoch 51/100] [Batch 390/600] [D loss: 1.360399] [G loss: 0.705398]
[Epoch 51/100] [Batch 391/600] [D loss: 1.388872] [G loss: 0.766551]
[Epoch 51/100] [Batch 392/600] [D loss: 1.404952] [G loss: 0.749698]
[Epoch 51/100] [Batch 393/600] [D loss: 1.374638] [G loss: 0.718199]
[Epoch 51/100] [Batch 394/600] [D loss: 1.391807] [G loss: 0.729218]
[Epoch 51/100] [Batch 395/600] [D loss: 1.396869] [G loss: 0.750972]
[Epoch 51/100] [Batch 396/600] [D loss: 1.379437] [G loss: 0.747336]
[Epoch 51/100] [Batch 397/600] [D loss: 1.351562] [G loss: 0.727673]
[Epoch 51/100] [Batch 398/600] [D loss: 1.380656] [G loss: 0.741988]
[Epoch 51/100] [Batch 399/600] [D loss: 1.375863] [G loss: 0.755267]
[Epoch 51/100] [Batch 400/600] [D loss: 1.337147] [G loss: 0.738031]
[Epoch 51/100] [Batch 401/600] [D loss: 1.376775] [G loss: 0.708846]
[Epoch 51/100] [Batch 402/600] [D loss: 1.364091] [G loss: 0.687748]
[Epoch 51/100] [Batch 403/600] [D loss: 1.409679] [G loss: 0.761186]
[Epoch 51/100] [Batch 404/600] [D loss: 1.425150] [G loss: 0.765973]
[Epoch 51/100] [Batch 405/600] [D loss: 1.383588] [G loss: 0.791326]
[Epoch 51/100] [Batch 406/600] [D loss: 1.387311] [G loss: 0.749566]
[Epoch 51/100] [Batch 407/600] [D loss: 1.351777] [G loss: 0.719455]
[Epoch 51/100] [Batch 408/600] [D loss: 1.379910] [G loss: 0.706304]
[Epoch 51/100] [Batch 409/600] [D loss: 1.384491] [G loss: 0.723576]
[Epoch 51/100] [Batch 410/600] [D loss: 1.364729] [G loss: 0.702089]
[Epoch 51/100] [Batch 411/600] [D loss: 1.370401] [G loss: 0.743031]
[Epoch 51/100] [Batch 412/600] [D loss: 1.391740] [G loss: 0.739671]
[Epoch 51/100] [Batch 413/600] [D loss: 1.379221] [G loss: 0.729707]
[Epoch 51/100] [Batch 414/600] [D loss: 1.367411] [G loss: 0.714970]
[Epoch 51/100] [Batch 415/600] [D loss: 1.359619] [G loss: 0.720936]
[Epoch 51/100] [Batch 416/600] [D loss: 1.375235] [G loss: 0.719716]
[Epoch 51/100] [Batch 417/600] [D loss: 1.407580] [G loss: 0.741468]
[Epoch 51/100] [Batch 418/600] [D loss: 1.413013] [G loss: 0.742689]
[Epoch 51/100] [Batch 419/600] [D loss: 1.393138] [G loss: 0.719493]
```

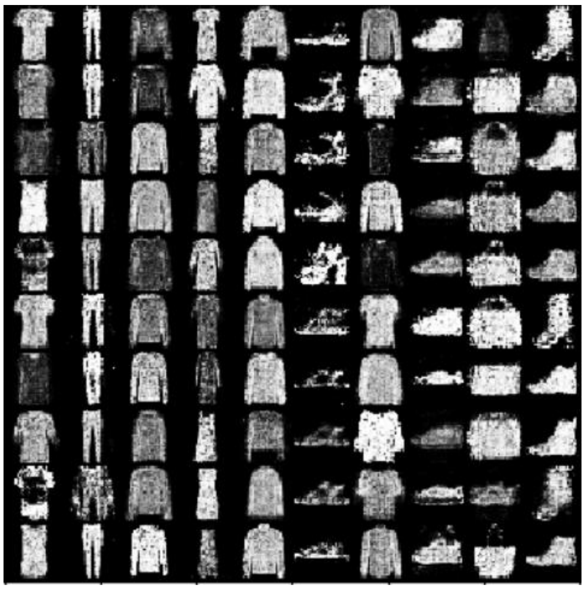
第一次迭代：

第五次迭代：

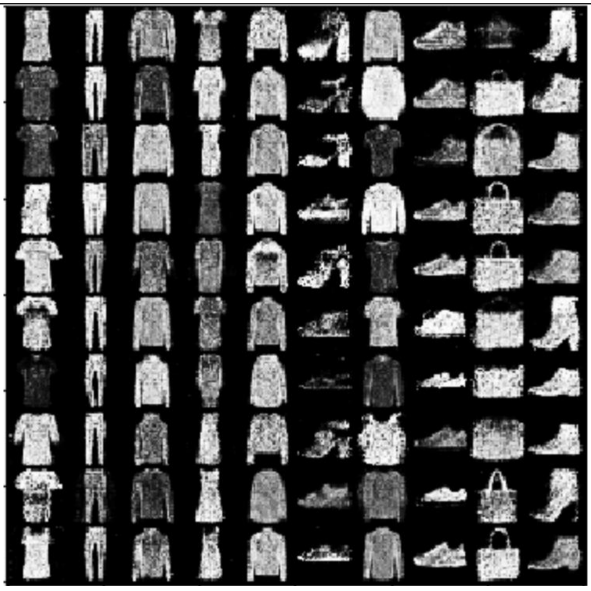
第十次迭代：



第五十次迭代：



第一百次迭代：



效果在二十次迭代后其实已经趋于稳定，后面的效果差别不大。

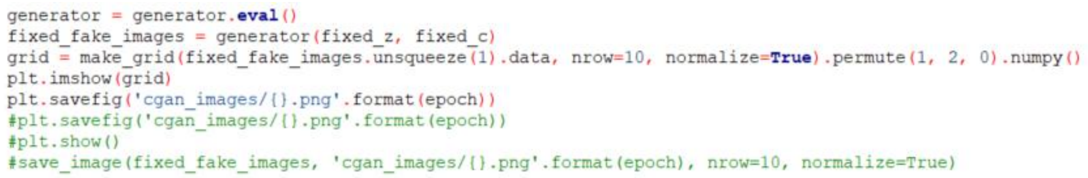
结论分析与体会：

使用 GAN 有条件地生成图片非常有趣,通过反复训练生成器和判别器使生成器生成尽可能接近真实情况的图片，判别器尽可能甄别出生成器生成的 fake images，使这两者在反复博弈中不断进化。该网络也经常被用来生成人物头像，或许不久后一些插画师就要下岗了……

就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1－3 道问答题：

1. 原始代码中的 save_image(fixed_fake_images, 'cgan_images/{}.png'.format(epoch), nrow=10, normalize=True)总是报错，该如何解决？

不去存 fixed_fake_images,直接去存 make_grid 产生的 grid。



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

改成这样就可以了。

2. 本任务中 Torchvision.transforms 是怎样的函数？

torchvision.transforms.Compose（）用来将多个变换组合在一起

torchvision.transforms.ToTensor 将像素值为[0,255]的 PIL 图像或 shape 为（H,W,C）的 np.ndarray 转化为形状为[C,H,W]，范围取值为[0,1]的 torch.FloatTensor

torchvision.transforms.Normalize（*mean*，*std*，*inplace = False* ）用给定均值和标准差来归一化张量图像