信息网络专题研究大作业报告（应用层）

一、实验目的

了解 GANs 的基本原理，用 pytorch 实现生成式对抗网络模型。

二、实验原理

此次实验是在笔记本电脑的对应环境下，运行 python 代码以产生一个典型的生成对抗网络，来体会 pytorch 库在机器学习领域的应用。

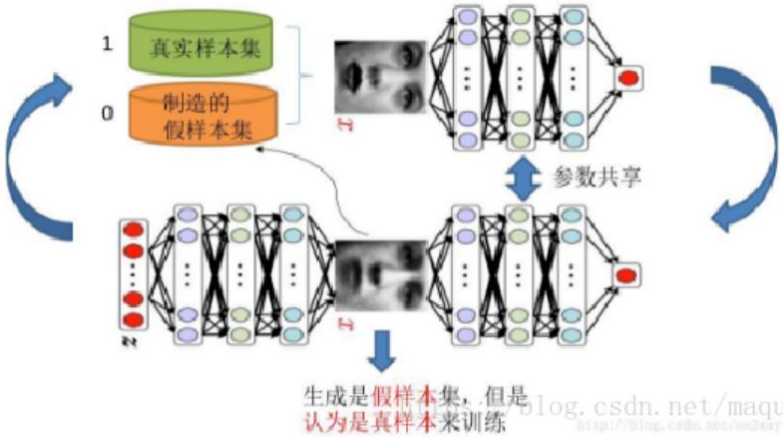
生成式对抗网络（Generative Adversarial Networks，GANs）

【概述】

GANs 中包含了两个模型，一个是生成模型 G，另一个是判别模型 D。

- 生成模型：对联合概率进行建模，从统计的角度表示数据的分布情况，刻画数据是如何生成的，收敛速度快，如朴素贝叶斯，GDA，HMM 等；
- 判别模型：对条件概率 P(Y|X)进行建模，不关心数据如何生成，主要是寻找不同类别之间的最优分类面，例如 LR，SVM 等。

在 GANs 中，**生成模型** G 不断地从训练集中学习真实数据的概率分布，根据误差通过链式求导并更新自己的参数，最终目标是使得 G 生成的图片（即“假”的）通过 D 后最大可能被判为是真的，达到以假乱真的效果。而**判别模型** D 则不断地判断来自训练集和 G 的图片是否为真实的，最终目的是准确地分辨“真”图片和“假”图片。可以看出，这两个模型在训练的过程中是对抗的，相互竞争的，故得名生成式对抗网络。具体的过程如下图所示。



生成对抗模型图示

迭代博弈的过程可以概括为：生成模型生成一些图片—>判别模型学习区分生成的图片和真实图片—>生成模型根据判别模型改进自己，生成新的图片—>判别模型再学习区分生

成的图片和真实图片…如此循环，直到判别模型和生成模型都无法再改进自己，也就达到了双方相互制约的目的，形成比较完美的对抗网络。这时，由生成模型生成的图片被判别模型正确识别的概率为 0.5，而原本真实的图片被判别模型正确识别的概率也为 0.5。

【数学表示&代码讲解】

参考 Ian 论文《Generative Adversarial Nets》中给出的结论如下，

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

其中，生成模型 G 的输入是随机噪声 z，输出是“假”的图片 G(z)，判别模型 D 的输入不区分是“真”图片 x 和“假”图片 G(z)，输出均是对每张图片的判定结果 outputs。

```
# ================================================================== #
#                      Train the discriminator                       #
# ================================================================== #

# Compute BCE_Loss using real images where BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
# Second term of the loss is always zero since real_labels == 1
outputs = D(images)
d_loss_real = criterion(outputs, real_labels)
real_score = outputs

# Compute BCELoss using fake images
# First term of the loss is always zero since fake_labels == 0
z = torch.randn(batch_size, latent_size).to(device)
fake_images = G(z)
outputs = D(fake_images)
d_loss_fake = criterion(outputs, fake_labels)  #根据虚假图片修正D判别器
fake_score = outputs

# Backprop and optimize  优化D判别器
d_loss = d_loss_real + d_loss_fake
reset_grad()
d_loss.backward()
d_optimizer.step() #训练一次，基于反向梯度，更新一次D的参数空间
```

```
# ================================================================ #
#                      Train the generator                         #
# ================================================================ #

# Compute loss with fake images
z = torch.randn(batch_size, latent_size).to(device)
fake_images = G(z)
outputs = D(fake_images)

# We train G to maximize log(D(G(z)) instead of minimizing log(1-D(G(z)))
# For the reason, see the last paragraph of section 3. https://arxiv.org/pdf/1406.2661.pdf
g_loss = criterion(outputs, real_labels)  #根据真实图片修正G生成器

# Backprop and optimize  优化G生成器
reset_grad()
g_loss.backward()
g_optimizer.step() #更新G的参数空间
```

然后，对于判别模型，将“真”图片的 outputs 和 1 输入 criterition，得到对“真”图片的分辨误差 d_loss_real，同时也把“假”图片的 outputs 和 0 输入得到 d_loss_fake，两者之和作为总误差，经过 backprop 反向传播和优化器更新网络模型的参数。对于生成模型，只考虑“假”图片的 outputs 和 1 之间的 g_loss，用相同方法改进该网络模型参数。每更新一次参数记为一步 step，每遍历完一遍训练集数据即为一个周期 epoch，实验设置为 epoch=200，进行 200 次遍历。

```
# Binary cross entropy loss and optimizer   交叉熵损失函数，用于二分类，输出>=0
criterion = nn.BCELoss()      #设定评判准则
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0002)  #待优化参数、学习率
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0002)   #将网络的参数放到各自的优化器里面
```

Criterition 是一个交叉熵损失函数，在判别模型中，

$$\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

优化器以最大化“真”图片的 outputs 和最小化“假”图片的 outputs 为目标，在生成模型中，

$$\underset{G}{\text{maximize}}\ \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

优化器以最大化“假”图片的 outputs 为目标。

三、实验数据

torchvision 库中的 MNIST 数据集，是一个包含 60,000 个用于训练的示例和 10,000 个用于测试的示例的手写数字数据集。这些数字已经过尺寸标准化并位于图像中心，图像是固定大小(28x28 像素)，其值为 0 到 1。为简单起见，每个图像都被平展并转换为 784(28 * 28)个特征的一维 numpy 数组。

四、实验步骤

1、 首先准备好硬件（一台笔记本电脑）

2、 配置好进行实验所需的环境，选择为 vscode+anaconda3

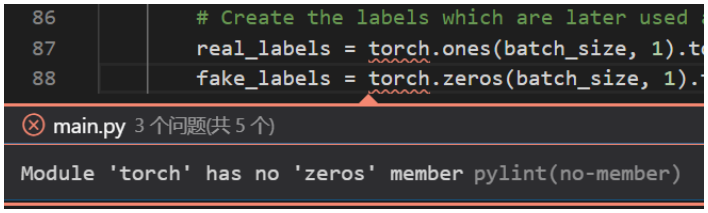
最开始选择的是 Pycharm+anaconda3，后由于内存不够，便用更轻便的 vscode 替代了 pycharm，且经过实验感受到 vscode 的显著优点，体验感佳）

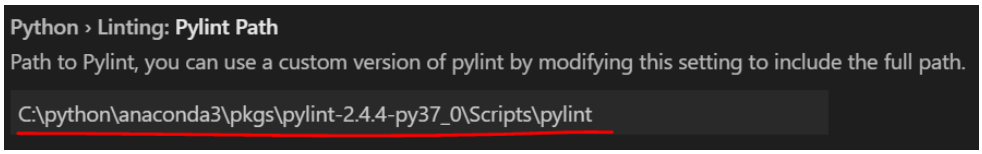
3、 安装 pytorch 机器学习专用库

这一部分和上一部分花费时间较长，主要原因是对使用命令行下载不太熟练

4、 从 GitHub 上下载程序的代码并导入 vscode，进行程序的调试，此时还未运行代码

出现的问题和解决办法如下：



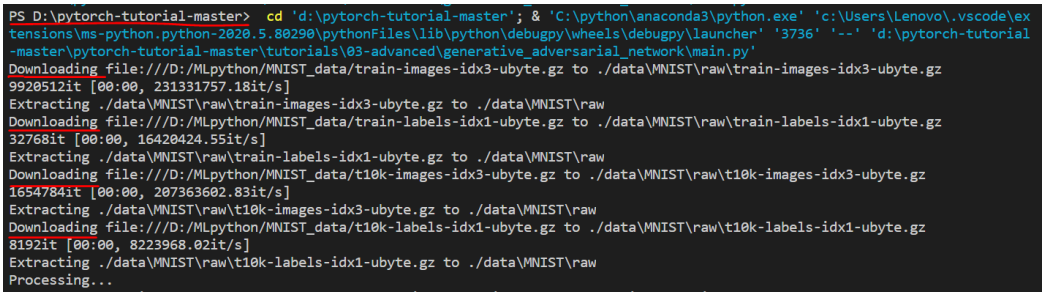


5、 根据自己掌握的部分 python 基础和学习机器学习课程所获知识，看懂理解代码

6、 运行代码，等待大约 2 小时，观察实验结果

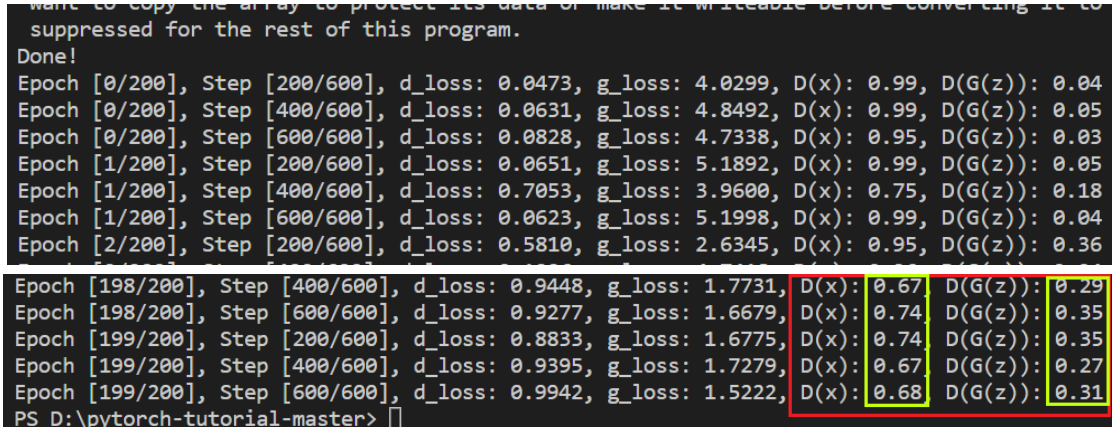
五、实验结果

运行代码后，在终端一栏显示如下内容。首先从本地资源库里下载用于训练的 MNIST 数据包，原始路径为国外的官网，但下载速度很慢且总是中断，多次尝试却出现“远程主机强迫关闭了一个现有的连接”的情况，所以选择先将数据下载到本地，再更改获取链接。



接着，就可以看到网络开始训练了。每完成 200 次参数更新就会显示一条进度，说明训练在正常进行中，一段时间过后，随着训练周期次数增加，得到的结果也有所变化。

D(x)的值表明了判别器将真图片判为真的概率，D(G(z))表明了判别器把假图片判为真



的概率，经过和最开始时候的值对比，可以发现前者有所下降 0.99—>0.70 左右，后者有所上升 0.04—>0.30 左右。因此，可以推测，如果延长训练时间，增加训练的周期数，那么两个概率会逐渐向中间 0.50 靠拢，最终基本不变，形成相互制约的局面。当然，要到达这样结果不仅需要更长的时间，训练数据的要求也可能会更多，这对于此次基于 CPU 运算的实验来说，计算机的任务负载会很大，所以还需要更强大的硬件设备来支持这样的机器学习任务。

六、实验体会

这次实验，我不仅学习到了 GAN 的基本原理和核心思想，也体会到了如何借助强大的 pytorch 库完成看上去具有如此复杂数学表达式的网络构建，真的就像是为了让计算机做这些学习工作而为它量身定制的一样，不同的函数都能灵活使用，不仅加快了编程人员开发程序的速度，也为读者提供了更高效和更方便的阅读体验，确实很不错，很值得学习。

（附）实验代码

本实验所用代码参考以下链接：

https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/03-advanced/generative_adversarial_network/main.py#L41-L57