生成对抗网络（GAN）

1. GAN原理介绍

➀引入的第一篇paper是Ian Goodfellow大牛的Generative Adversarial Networks（https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf），这篇paper算是这个领域的开山之作。

➁最新的一篇paper：<https://arxiv.org/abs/1701.00160>

➂Alec Radford、Luke Metz和Soumith Chintala等人的论文，介绍了本教程中我们在生成器上使用的复杂GANs的基本结构：

Paper：https://arxiv.org/abs/1511.06434

**（1）GAN模型：**

GAN，即生成对抗网络，主要包含两个模块：

* **生成器（Generative Model）**
* **判别器（Discriminative Model）**

生成模型和判别模型之间互相博弈、学习产生相当好的输出。以图片为例，生成器的主要任务是学习真实图片集，从而使得自己生成的图片更接近于真实图片，以“骗过”判别器。而判别器的主要任务是找出出生成器生成的图片，区分其与真实图片的不同，进行真假判别。在整个迭代过程中，生成器不断努力让生成的图片越来越像真的，而判别器不断努力识别出图片的真假。这类似生成器与判别器之间的博弈，随着反复迭代，最终二者达到了平衡：生成器生成的图片非常接近于真实图片，而判别器已经很难识别出真假图片的不同了。其表现是对于真假图片，判别器的概率输出都接近 0.5。

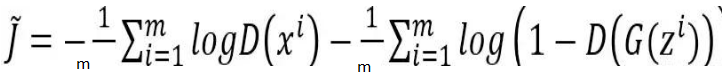
这样我们的目的就达成了：我们得到了一个生成式的模型G，它可以用来生成图片。GAN 基本模型由 输入 Vector、G 网络、D 网络组成。其中，G 和 D 一般都是由神经网络组成。G 的输出是一幅图片，只不过是以全连接形式。G 的输出是 D 的输入，D 的输入还包含真实样本集。这样， D 对真实样本尽量输出 score 高一些，对 G 产生的样本尽量输出 score 低一些。每次循环迭代，G 网络不断优化网络参数，使 D 无法区分真假；而 D 网络也在不断优化网络参数，提高辨识度，让真假样本的 score 有差距。最终，经过多次训练迭代，GAN 模型建立。

最终的 GAN 模型中，G 生成的样本以假乱真，D 输出的 score 接近 0.5，即表示真假样本难以区分，训练成功。

这里，重点要讲解一下输入 vector。输入向量是用来做什么的呢？其实，输入 vector 中的每一维度都可以代表输出图片的某个特征。比如说，输入 vector 的第一个维度数值大小可以调节生成图片的头发颜色，数值大一些是红色，数值小一些是黑色；输入 vector 的第二个维度数值大小可以调节生成图片的肤色；输入 vector 的第三个维度数值大小可以调节生成图片的表情情绪，等等。

GAN 的强大之处也正是在于此，通过调节输入 vector，就可以生成具有不同特征的图片。而这些生成的图片不是真实样本集里有的，而是即合理而又没有见过的图片。是不是很有意思呢？下面这张图反映的是不同的 vector 生成不同的图片。

**（2）GAN算法原理：**

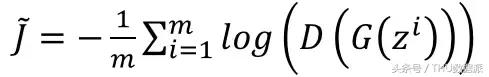
GAN 的算法原理。既然有两个模块：G 和 D，每个模块都有相应的网络参数。先来看 D 模块，它的目标是让真实样本 score 越大越好，让 G 产生的样本 score 越小越好。那么可以得到 D 的损失函数为：

m

m

其中，x 是真实样本，G(z) 是 G 生成样本。我们希望 D(x) 越大越好，D(G(z)) 越小越好，也就是希望 -D(x) 越小越好，-log(1-D(G(z))) 越小越好。从损失函数的角度来说，能够得到上式。

再来看 G 模块，它的目标就是希望其生成的模型能够在 D 中得到越高的分数越好。那么可以得到 G 的损失函数为：



知道了损失函数之后，接下来就可以使用各种优化算法来训练模型了。

1. **GAN模型实例**

基于PyTorch 实现一个简单的 GAN 模型。仍然以绘画创作为例，假设我们要创造如下“名画”（以正弦图形为例）：

环境：win10 + Anaconda + pycharm

安装的库：pytorch、numpy、matplotlib

代码：

import torch  
import torch.nn as nn  
import numpy as np  
import matplotlib.pyplot as plt  
  
# torch.manual\_seed(1) # reproducible  
# np.random.seed(1)  
  
# Hyper Parameters  
BATCH\_SIZE = 64  
LR\_G = 0.0001 # learning rate for generator  
LR\_D = 0.0001 # learning rate for discriminator  
N\_IDEAS = 5 # think of this as number of ideas for generating an art work(Generator)  
ART\_COMPONENTS = 15 # it could be total point G can drew in the canvas  
PAINT\_POINTS = np.vstack([np.linspace(-1, 1, ART\_COMPONENTS) for \_ in range(BATCH\_SIZE)])  
  
def artist\_works(): # 生成艺术画作  
 # a = np.random.uniform(1, 2, size=BATCH\_SIZE)[:, np.newaxis]  
 r = 0.02 \* np.random.randn(1, ART\_COMPONENTS)  
 paintings = np.sin(PAINT\_POINTS \* np.pi) + r  
 paintings = torch.from\_numpy(paintings).float()  
 return paintings

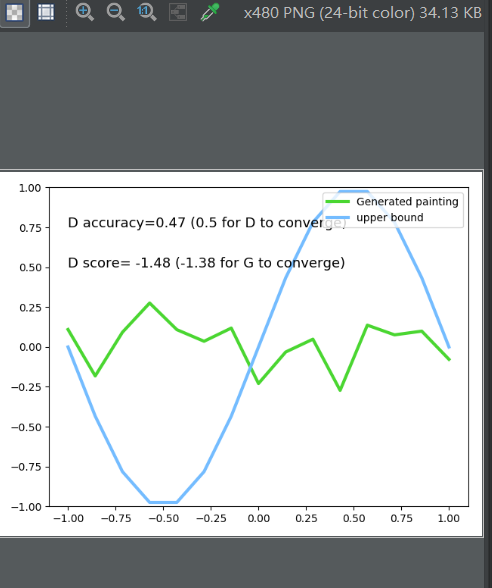
#定义G网络模型  
G = nn.Sequential( # Generator  
 nn.Linear(N\_IDEAS, 128), # random ideas (could from normal distribution)  
 nn.ReLU(),  
 nn.Linear(128, ART\_COMPONENTS), # making a painting from these random ideas  
)

#定义D网络模型  
D = nn.Sequential( # Discriminator  
 nn.Linear(ART\_COMPONENTS, 128), # receive art work either from the famous artist or a newbie like G  
 nn.ReLU(),  
 nn.Linear(128, 1),  
 nn.Sigmoid(), # tell the probability that the art work is made by artist  
)

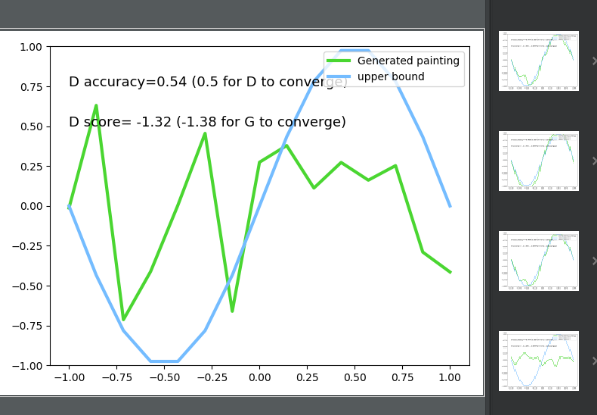
#设置Adam算法进行优化  
opt\_D = torch.optim.Adam(D.parameters(), lr=LR\_D)  
opt\_G = torch.optim.Adam(G.parameters(), lr=LR\_G)

#构建GAN训练迭代过程  
plt.ion() # something about continuous plotting  
  
D\_loss\_history = []  
G\_loss\_history = []  
for step in range(1):  
 artist\_paintings = artist\_works() # real painting from artist  
 G\_ideas = torch.randn(BATCH\_SIZE, N\_IDEAS) # random ideas  
 G\_paintings = G(G\_ideas) # fake painting from G (random ideas)  
  
 prob\_artist0 = D(artist\_paintings) # D try to increase this prob  
 prob\_artist1 = D(G\_paintings) # D try to reduce this prob  
  
 D\_loss = - torch.mean(torch.log(prob\_artist0) + torch.log(1. - prob\_artist1))  
 G\_loss = torch.mean(torch.log(1. - prob\_artist1))  
  
 D\_loss\_history.append(D\_loss)  
 G\_loss\_history.append(G\_loss)  
  
 opt\_D.zero\_grad()  
 D\_loss.backward(retain\_graph=True) # reusing computational graph  
 opt\_D.step()  
  
 opt\_G.zero\_grad()  
 G\_loss.backward()  
 opt\_G.step()  
  
 if step % 50 == 0: # plotting  
 plt.cla()  
 plt.plot(PAINT\_POINTS[0], G\_paintings.data.numpy()[0], c='#4AD631', lw=3, label='Generated painting', )  
 plt.plot(PAINT\_POINTS[0], np.sin(PAINT\_POINTS[0] \* np.pi), c='#74BCFF', lw=3, label='upper bound')  
 plt.text(-1, 0.75, 'D accuracy=%.2f (0.5 for D to converge)' % prob\_artist0.data.numpy().mean(),  
 fontdict={'size': 13})  
 plt.text(-1, 0.5, 'D score= %.2f (-1.38 for G to converge)' % -D\_loss.data.numpy(), fontdict={'size': 13})  
 plt.ylim((-1, 1));  
 plt.legend(loc='upper right', fontsize=10);  
 plt.draw();  
 plt.pause(0.01)  
  
plt.ioff()  
plt.show()

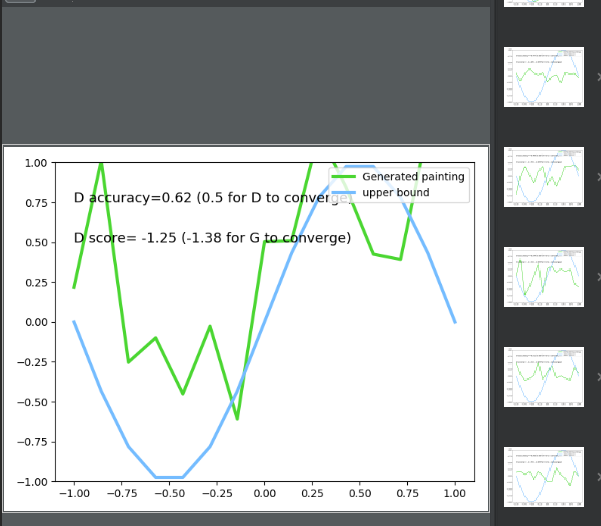
**迭代1次时：**



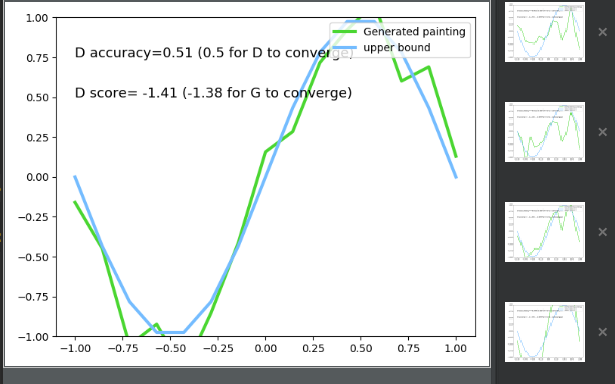
**迭代100次时：**



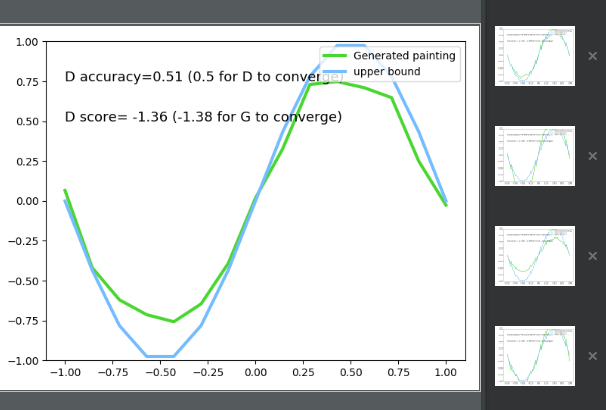
**迭代200次时：**



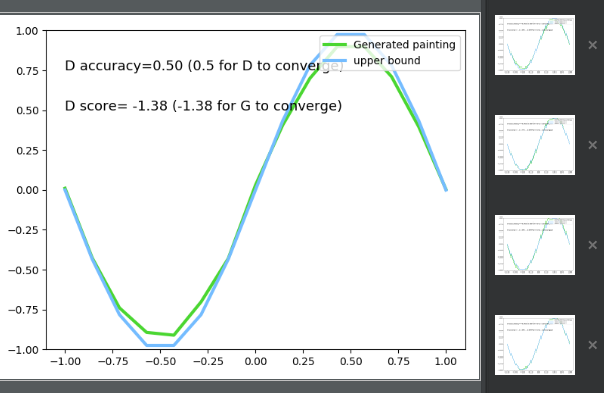
**迭代500次时：**



**迭代1000次时：**



**迭代3000次时：**



迭代10000次时出现内存不足情况