人工神经网络模型与算法

2025-5-4

姓名：　　李丰克
专业：　自动化（控制）
年级：　　大二
学号：　　3230105182

Chapter 3 基于对抗生成网络GAN实现图象生成

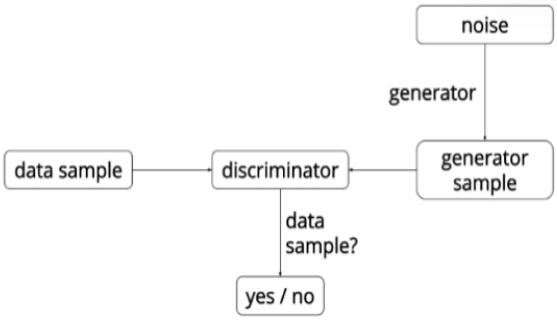
3.1 问题简介

本次实验尝试用对抗生成网络GAN实现生成图象。

3.2 模型介绍

3.2.1 模型原理

生成对抗网络由两个网络组成：生成网络（Generator）和判别网络（Discriminator），G、D两个网络互相博弈学习产生输出结果。



生成器G负责接收一个随机的噪声，通过这个噪声生成图片，将生成的图片送给判别器。判别器D是一个二分类器，用于判断图片是真是假，图片越真，输出的值越接近1，反之接近0。

在训练过程中，G尽量使生成的图片能够欺骗过D，即提高经过D网络判别后的输出概率值。D的目标是提高区分G的假图片和真实图片的能力。这样，G和D就形成了一种博弈过程。

判别器D的训练过程为：随机取样噪声，将噪声输入生成器G得到假图片，其lable为0；再随机采样训练数据，即真图片，其label为1。以此作为训练集进行输入，用输出值与原始标签计算损失函数，反向传播更新网络参数。

生成器G的训练过程为：随机取样噪声，输入生成器，输出假图片，将假图片输入判别器，用输出的概率值与目标输出值1计算损失函数，反向传播更新网络参数。

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.

两个网络训练时是互相独立的，即一个网络训练时，另一个网络的参数冻结，两者互不干扰。

损失函数用的是交叉熵函数。

$$H(p, q) := -\sum_{i} p_i \log q_i$$

图象的特征并没有分别用各种标签区分出来，仅仅根据真假标签，G和D可以自行提取出训练数据图象的特征，并更新到网络参数。两者相互博弈训练，最终达到一个动态的平衡（纳什均衡），此时生成器已经提取出了图象的特征，获取了训练数据的分布，判别模型此时也提取出了图象的特征，但是根据这样的“评判标准”，判别器已经判断不出图象的真假（输出为0.5），生成器已经能够以假乱真，双方达到平衡。

3.2.2 代码实现

```
#定义生成器Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        def block(in_feat, out_feat, normalize=True):          #block(in, out )
            layers = [nn.Linear(in_feat, out_feat)]            #线性变换将输入映射到out维
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))   #正则化
            layers.append(nn.LeakyReLU(0.2, inplace=True))     #非线性激活函数
            return layers
        self.model = nn.Sequential(
            *block(opt.latent_dim, 128, normalize=False),      #线性变化将输入映射 100 to 128, 正则化, LeakyReLU
            *block(128, 256),                                  #线性变化将输入映射 128 to 256, 正则化, LeakyReLU
            *block(256, 512),                                  #线性变化将输入映射 256 to 512, 正则化, LeakyReLU
            *block(512, 1024),                                 #线性变化将输入映射 512 to 1024, 正则化, LeakyReLU
            nn.Linear(1024, img_area),                         #线性变化将输入映射 1024 to image_size
            nn.Tanh()
        )
    def forward(self, z):                                      #输入的是(64,  100)的噪声数据
        imgs = self.model(z)                                    #噪声数据通过生成器模型
        imgs = imgs.view(imgs.size(0), *img_shape)              #reshape
        return imgs
```

此为生成器的网络，每个中间层都由一个linear线性层、一个正则化、一个LeakyRele激活函数组成。将随机的(100，)的噪声转化为(image_size，)的图象一维展开向量。

```
#定义判别器 Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(img_area, 512),                  #输入特征数为image_size，输出为512
            nn.LeakyReLU(0.2, inplace=True),           #进行非线性映射
            nn.Linear(512, 256),                       #输入特征数为512，输出为256
            nn.LeakyReLU(0.2, inplace=True),           #进行非线性映射
            nn.Linear(256, 1),                         #输入特征数为256，输出为1
            nn.Sigmoid(),                              #输入sigmoid函数二分类
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)           #鉴别器输入是一个被view展开的image_size的一维图像
        validity = self.model(img_flat)
        return validity                                #返回[0, 1]间的概率
```

此为判别器的网络，由三层线性层+LeakyRelu激活函数的组合构成，最后输入sigmoid激活函数进行二分类。

```
#训练判别器
imgs = imgs.view(imgs.size(0), -1)                      #展开图片为一维
real_img = Variable(imgs).cuda()                        #将tensor变成Variable放入计算图中
real_label = Variable(torch.ones(imgs.size(0), 1)).cuda()     #定义真实的图片label为1
fake_label = Variable(torch.zeros(imgs.size(0), 1)).cuda()    #定义假的图片的label为0
#计算真的图片的损失
real_out = discriminator(real_img)                      #将真实图片放入判别器中
loss_real_D = criterion(real_out, real_label)           #得到真实图片的loss
real_scores = real_out                                  #得到真实图片的判别值，输出的值越接近1越好
#计算假的图片的损失
z = Variable(torch.randn(imgs.size(0), opt.latent_dim)).cuda()     #随机生成一些噪声, 大小为(128, 100)
fake_img = generator(z).detach()                        #随机噪声放入生成网络中，生成一张假的图片。
fake_out = discriminator(fake_img)                      #判别器判断假的图片
loss_fake_D = criterion(fake_out, fake_label)           #得到假的图片的loss
fake_scores = fake_out                                  #得到假图片的判别值，对于判别器来说，假图片的损失越接近0越好
#损失函数和优化
loss_D = loss_real_D + loss_fake_D                 #损失包括判真损失和判假损失
optimizer_D.zero_grad()                            #反向传播之前，先将梯度归0
loss_D.backward()                                  #将误差反向传播
optimizer_D.step()                                 #更新参数
```

```
#训练生成器
z = Variable(torch.randn(imgs.size(0), opt.latent_dim)).cuda()     #得到随机噪声
fake_img = generator(z)                                            #随机噪声输入到生成器中，得到一副假的图片
output = discriminator(fake_img)                                   #经过判别器得到的结果
#损失函数和优化
loss_G = criterion(output, real_label)                             #得到的假的图片与真实的图片的label的loss
optimizer_G.zero_grad()                                            #梯度归0
loss_G.backward()                                                  #进行反向传播
optimizer_G.step()                                                 #step()一般用在反向传播后面,用于更新生成网络的参数

if (i + 1) % 100 == 0:
    print(
        "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f] [D real: %f] [D fake: %f]"
        % (epoch, opt.n_epochs, i, len(dataloader), loss_D.item(), loss_G.item(), real_scores.data.mean(), fake_scores.data.mean(
    )
#保存训练过程中的图像
batches_done = epoch * len(dataloader) + i
if batches_done % opt.sample_interval == 0:
    save_image(fake_img.data[:25], "./images/gan2/%d.png" % batches_done, nrow=5, normalize=True)
```

以上两者的训练函数，与原理基本对应。判别器是获取随机噪声经过生成器后用输出在经过判别器计算假图片的损失函数；获取随机训练数据经过判别器计算真图片的损函数，用两个损失函数更新参数；生成器是用随机噪声生成图片经过判别器后的输出计算损失函数，再更新参数。

3.3 数据集介绍

一开始用了上次报告的猫狗数据集，后来发现数据集图片太混乱了，包含大量的环境干扰，导致最后的生成效果不好，如果是纯猫狗面部的图象数据集，效果会很好。

本次用了三个数据集，人脸数据集lfw，动漫头像数据集anime-data以及英雄联盟原画数据集。

lfw有13233张人脸数据集，分别来自5749个人。anime-data有21551张动漫面部头像图片。英雄联盟原画数据集来自其官网，共有8000张不同截图区域的原画，其中有部分重复。

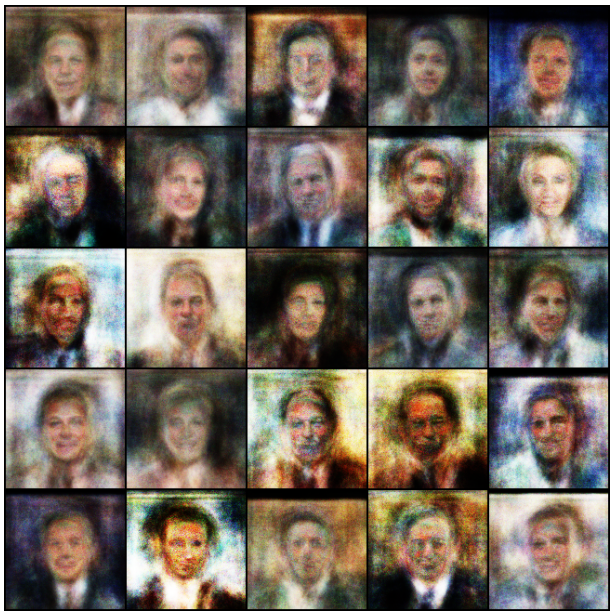
3.4 训练以及结果

参数设置为：学习率0.0002，每次采样数量64，优化器ADAM，其参数b1为0.5，b2为0.999，噪声空间维度100，图象长宽为128，图象通道数为3(彩色图象RGB)，采样间隔500，训练轮数50。全部数据封装在超参数中。

```
#超参数配置
parser = argparse.ArgumentParser()
parser.add_argument("--n_epochs", type=int, default=50, help="训练轮数")
parser.add_argument("--batch_size", type=int, default=64, help="每次采样数量")
parser.add_argument("--lr", type=float, default=0.0002, help="ADAM学习率")
parser.add_argument("--b1", type=float, default=0.5, help="ADAM参数")
parser.add_argument("--b2", type=float, default=0.999, help="ADAM参数")
parser.add_argument("--n_cpu", type=int, default=2)
parser.add_argument("--latent_dim", type=int, default=100, help="噪声空间维度")
parser.add_argument("--img_size", type=int, default=128, help="图象大小")
parser.add_argument("--channels", type=int, default=3, help="图象通道数，彩色图片为三色RGB")
parser.add_argument("--sample_interval", type=int, default=500, help="采样间隔")
opt = parser.parse_args()
print(opt)
```

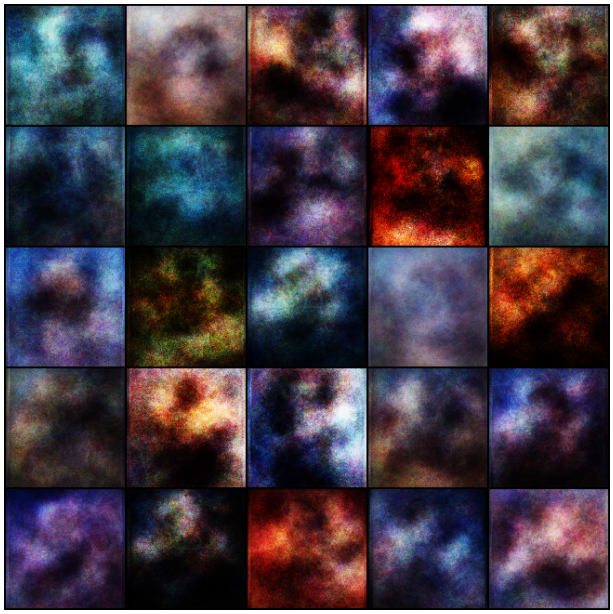
将训练过程中的图象存在文件夹images中，将训练后的网络参数存在文件夹save中。

对数据集lfw的训练结果如下图，共训练了50轮。



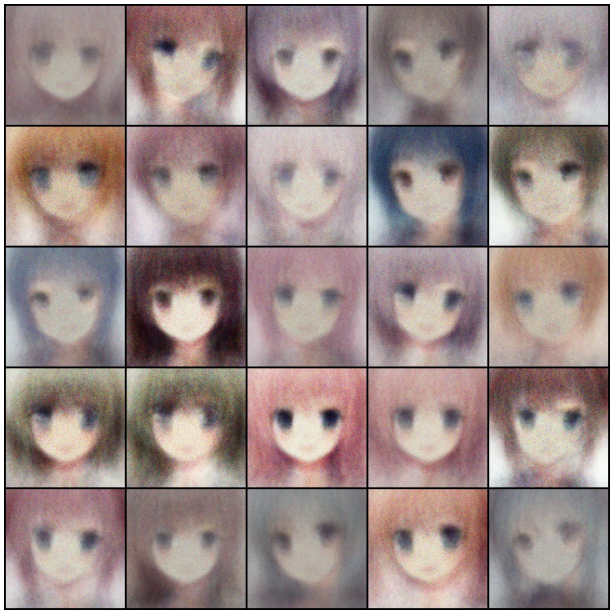
对anime-data的训练结果如下图，训练50轮。

对英雄联盟原画数据集的训练结果如下图，训练50轮。



可见第三个数据集训练结果一般，这也是合理的，原因与猫狗数据集类似，特征不明显或者特征太多，导致没有特别明显的规律，环境比较杂乱。

前两个数据集效果尚可，但是在训练过程中出现了一个问题，就是随着训练轮数的增多，生成图象的饱和度与对比度会逐渐增大。以第二个数据集为例，以下分别为采样第1000与采样第7000时的生成图。



确实存在饱和度与对比度逐渐升高的现象，可能是因为在训练过程中，原始数据分布特征逐渐提纯，最后出现过拟合。

3.5 代码附录

代码的zjugit网址：https://git.zju.edu.cn/3230105182/gan