|  |  |
| --- | --- |
| 分数： |  |

**深度学习**

**2020 - 2021 学年度第 二 学期**

实践报告

任课教师： 孔雨秋

# **题目：深度生成模型（**变分自编码器（VAE）与生成对抗网络（GAN））

院(系): 电子信息与电气工程

班级： 电智1901

学号： 201981242

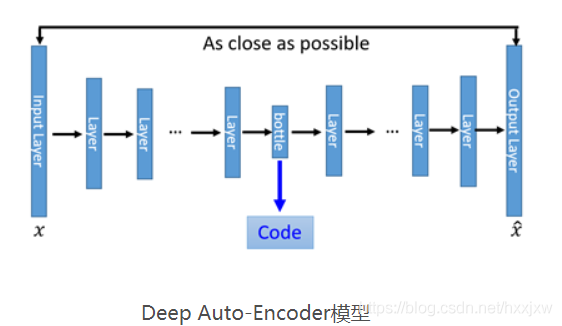
姓名： 李浩

**1. 理论基础**

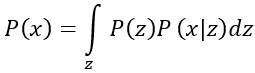
**1.1VAE**

VAE作为一个生成模型，其基本思路是很容易理解的：把一堆真实样本通过编码器网络变换成一个理想的数据分布，然后这个数据分布再传递给一个解码器网络，得到一堆生成样本，生成样本与真实样本足够接近的话，就训练出了一个自编码器模型。

从深度学习的角度来理解就是以下这幅图：

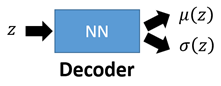


VAE的理论基础就是高斯混合模型（任何一个数据的分布，都可以看作是若干高斯分布的叠加）。我们的编码换成一个连续变量z，我们规定z服从正态分布N(0,1)（实际上并不一定要选用N(0,1)，其他的连续分布都是可行的）。每对于一个采样z，会有两个函数μ和σ，分别决定z对应到的高斯分布的均值和方差，然后在积分域上所有的高斯分布的累加就成为了原始分布P(X),即：

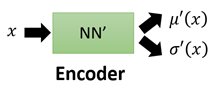


其中z~N(0,1)，x|z ~ N(μ(z),σ(z))。我们需要引入两个神经网络来帮助我们求解。

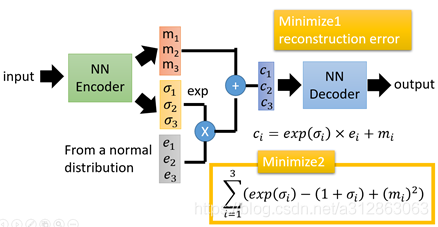
第一个神经网络叫做Decoder，它求解的是μ和σ两个函数，这等价于求解P(x|z)，因为x|z ~ N(μ(z),σ(z))：



第二个神经网络叫做Encoder，它求解的结果是q(z|x)，q可以代表任何分布：



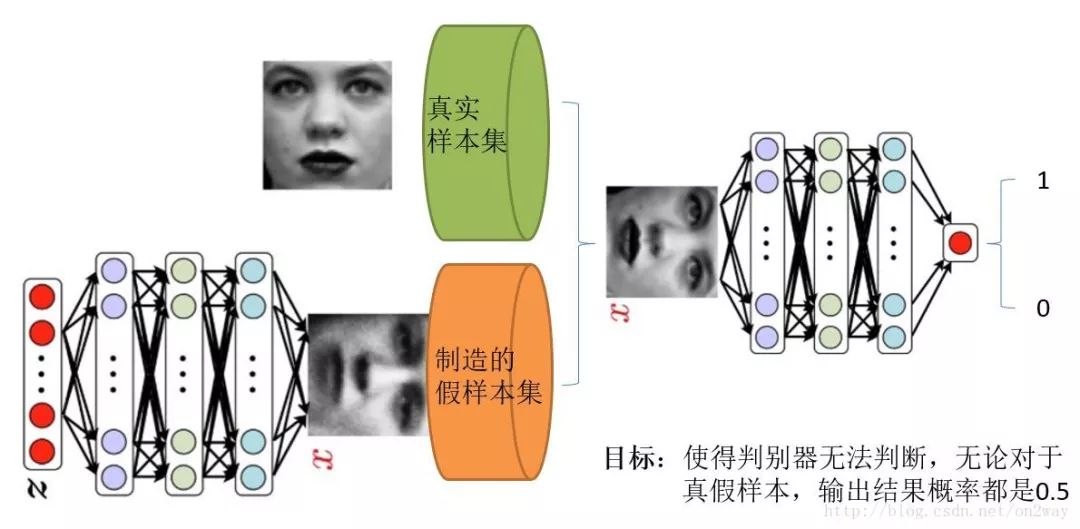
# 整个VAE的模型架构为：



**1.2 GAN**

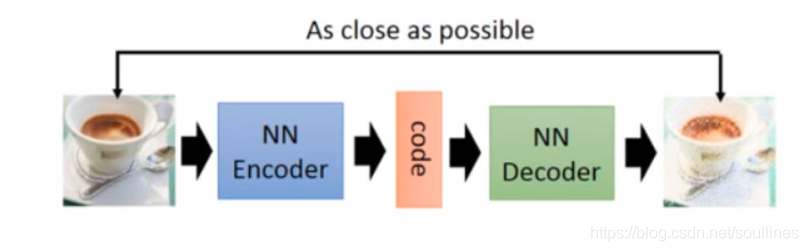
GAN的思想是是一种二人零和博弈思想（two-player game），博弈双方的利益之和是一个常数，比如两个人掰手腕，假设总的空间是一定的，你的力气大一点，那你就得到的空间多一点，相应的我的空间就少一点，相反我力气大我就得到的多一点，但有一点是确定的就是，我两的总空间是一定的，这就是二人博弈，但是呢总利益是一定的。

引申到GAN里面就是可以看成，GAN中有两个这样的博弈者，一个人名字是生成模型（G），另一个人名字是判别模型（D）。他们各自有各自的功能。



##### 1.2.1 生成器(Generation)

就是利用模型对图片的学习,最终达到可以自己生成图片的目的



第一步:将图片传入解码器 NN-Encoder 转化为机器可以识别的array形式,然后通过 NN-Decoder生成图片 P i c f a k e

第二步:已知真实图片 P i c r e a l,通过 l o s s loss loss函数 MSE,计算真实图片和生成图片的 l o s s,进而反馈网络

这样看起来好像是没有什么问题,但是需要注意一个问题,这里的 l o s s loss loss仅仅计算数据之间的差异,图片的像素 v a l val val不仅仅是数据的堆叠那么简单,同样相对位置(数据之间的相关性)也是很重要的一个部分,由于G网络没有办法学习到位置的相关性

所以 G e n e r a t i o n Generation Generation不能生成高还原度的图片

1.2.2 判别器(Discriminator)

简单来说就是一个判断 r e a l图片和 f a k e图片的二分类模型

Discriminator是一个卷积的神经网络,所以可以有效的区分图片的相对位置(即注重数据的相关性),但是由于Discriminator只对真实数据奖励(此时的 o u t p u t output output大),对伪造的数据惩罚(此时的 o u t p u t output output小)

所以随机数据的选取比较困难

这样来看 G网络和 D网络各有优缺点,但是刚刚好可以互补,所以GAN网络顺势而生

**2. 程序实现**

**2.1 VAE数字生成MINST程序实现**

**2.1.1 引入的包**

**import** torch  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**import** torchvision  
**import** torchvision.transforms **as** transforms  
**from** torchvision.utils **import** save\_image

**2.1.2 损失函数**

**def** loss\_function(recon\_x, x, mu, logvar):  
 *"""* **:param** *recon\_x: generated image* **:param** *x: original image* **:param** *mu: latent mean of z* **:param** *logvar: latent log variance of z  
 """* BCE\_loss = nn.BCELoss(reduction=**'sum'**)  
 reconstruction\_loss = BCE\_loss(recon\_x, x)  
 KL\_divergence = -0.5 \* torch.sum(1+logvar-torch.exp(logvar)-mu\*\*2)**return** reconstruction\_loss + KL\_divergence, reconstruction\_loss, KL\_divergence

**2.1.3 VAE模型**

**class** VAE(nn.Module):  
 **def** \_\_init\_\_(self):  
 super(VAE, self).\_\_init\_\_()  
 self.fc1 = nn.Linear(784, 400) *# 线性层* self.fc2\_mean = nn.Linear(400, 20)  
 self.fc2\_logvar = nn.Linear(400, 20)  
 self.fc3 = nn.Linear(20, 400)  
 self.fc4 = nn.Linear(400, 784)  
  
 *# 编码，学习高斯分布均值与方差* **def** encode(self, x):  
 h1 = F.relu(self.fc1(x))  
 **return** self.fc2\_mean(h1), self.fc2\_logvar(h1)  
  
 **def** reparametrization(self, mu, logvar):  
 *# sigma = 0.5\*exp(log(sigma^2))= 0.5\*exp(log(var))* std = 0.5 \* torch.exp(logvar)  
 *# N(mu, std^2) = N(0, 1) \* std + mu* z = torch.randn(std.size()) \* std + mu  
 **return** z  
  
 *# 解码隐变量z* **def** decode(self, z):  
 h3 = F.relu(self.fc3(z))  
 **return** torch.sigmoid(self.fc4(h3))  
 *# 计算重构值和隐变量z的分布参数* **def** forward(self, x):  
 mu, logvar = self.encode(x) *# 从原始样本x中学习隐变量z的分布，即学习服从高斯分布均值与方差* z = self.reparametrization(mu, logvar) *# 将高斯分布均值与方差参数重表示，生成隐变量z* **return** self.decode(z), mu, logvar *# 解码隐变量z，生成重构x’，并返回重构值和隐变量的分布参数*

**2.1.4 数据处理**

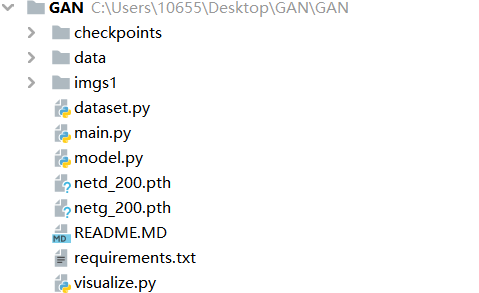
transform = transforms.Compose([  
 transforms.ToTensor(),  
 *# transforms.Normalize([0.5], [0.5]),*])  
  
trainset = torchvision.datasets.MNIST(root=**'./data'**, train=**True**, download=**True**, transform=transform) *# 读取数据集*trainloader = torch.utils.data.DataLoader(trainset, batch\_size=128, shuffle=**True**)  
  
testset = torchvision.datasets.MNIST(root=**'./data'**, train=**False**, download=**True**, transform=transform) *# 生成模型没有用到测试集*testloader = torch.utils.data.DataLoader(testset, batch\_size=100, shuffle=**False**)

**2.1.5 训练函数**

vae = VAE()  
optimizer = torch.optim.Adam(vae.parameters(), lr=0.0003) *# 选用Adam优化器  
  
# Training***def** train(epoch):  
 vae.train()  
 all\_loss = 0.  
 all\_recon\_loss = 0.  
 all\_kl\_loss = 0.  
 **for** batch\_idx, (inputs, targets) **in** enumerate(trainloader):  
 inputs, targets = inputs.to(**'cpu'**), targets.to(**'cpu'**) *# 使用cpu进行训练* real\_imgs = torch.flatten(inputs, start\_dim=1) *# 将batch\_size\*1\*28\*28  
  
 # Train Discriminator* gen\_imgs, mu, logvar = vae(real\_imgs) *# 将batch\_size\*748的x输入模型进行前向传播计算,重构值和服从高斯分布的隐变量z的分布参数（均值和方差）* loss, recon\_loss, kl\_loss = loss\_function(gen\_imgs, real\_imgs, mu, logvar) *# 计算重构损失和KL散度  
 # 重构损失  
 # 反向传播与优化  
 # 清空上一步的残余更新参数值* optimizer.zero\_grad()  
 *# 误差反向传播, 计算参数更新值* loss.backward()  
 *# 将参数更新值施加到VAE model的parameters上* optimizer.step()  
  
 all\_loss += loss.item()  
 all\_recon\_loss += recon\_loss.item()  
 all\_kl\_loss += kl\_loss.item()  
 *# 每迭代一定步骤，打印结果值* **if** batch\_idx % 1000 == 0:  
 print(**'Epoch {}, Iter {}, loss: {:.2f}'**.format(epoch, batch\_idx, all\_loss/(batch\_idx+1)))  
 print(**'======== Reconstruction Loss: {:.2f}'**.format(all\_recon\_loss/(batch\_idx+1)))  
 print(**'======== KL Divergence Loss: {:.2f}'**.format(all\_kl\_loss/(batch\_idx+1)))  
 *# Save generated images for every epoch* fake\_images = gen\_imgs.view(-1, 1, 28, 28)  
 save\_image(fake\_images, **'MNIST\_FAKE/fake\_images-{}.png'**.format(epoch + 1))*# 保存图片*

**2.2 GAN程序实现**

**2.2.1代码结构：**



checkpoints/ # 无代码，用来保存训练过程的模型

Imgs1/ # 无代码，用来保存生成的图片

data/ # 无代码，用来保存训练所需的图片

main.py # 训练和生成

model.py # 模型定义

visualize.py # 可视化工具visdom的封装

requirements.txt # 程序中用到的第三方库

README.MD # 说明

netd\_200.pth # 预训练模型

netg\_200.pth # 预训练模型

**2.2.2 生成器模型NetG：**

**class** NetG(nn.Module):  
 *"""  
 生成器定义  
 """* **def** \_\_init\_\_(self, opt):  
 super(NetG, self).\_\_init\_\_()  
 ngf = opt.ngf *# 生成器feature map数* self.main = nn.Sequential(  
 *# 输入是一个nz维度的噪声，我们可以认为它是一个1\*1\*nz的feature map* nn.ConvTranspose2d(opt.nz, ngf \* 8, 4, 1, 0, bias=**False**),  
 nn.BatchNorm2d(ngf \* 8),  
 nn.ReLU(**True**),  
 *# 上一步的输出形状：(ngf\*8) x 4 x 4*

nn.ConvTranspose2d(ngf \* 8, ngf \* 4, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ngf \* 4),  
 nn.ReLU(**True**),  
 *# 上一步的输出形状： (ngf\*4) x 8 x 8* nn.ConvTranspose2d(ngf \* 4, ngf \* 2, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ngf \* 2),  
 nn.ReLU(**True**),  
 *# 上一步的输出形状： (ngf\*2) x 16 x 16* nn.ConvTranspose2d(ngf \* 2, ngf, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ngf),  
 nn.ReLU(**True**),  
 *# 上一步的输出形状：(ngf) x 32 x 32* nn.ConvTranspose2d(ngf, 3, 5, 3, 1, bias=**False**),  
 nn.Tanh() *# 输出范围 -1~1 故而采用Tanh  
 # 输出形状：3 x 96 x 96* )  
 **def** forward(self, input):  
 **return** self.main(input)

**2.2.3 判别器模型NetD：**

**class** NetD(nn.Module):  
 *"""  
 判别器定义  
 """* **def** \_\_init\_\_(self, opt):  
 super(NetD, self).\_\_init\_\_()  
 ndf = opt.ndf  
 self.main = nn.Sequential(  
 *# 输入 3 x 96 x 96* nn.Conv2d(3, ndf, 5, 3, 1, bias=**False**),  
 nn.LeakyReLU(0.2, inplace=**True**),  
 *# 输出 (ndf) x 32 x 32* nn.Conv2d(ndf, ndf \* 2, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ndf \* 2),  
 nn.LeakyReLU(0.2, inplace=**True**),  
 *# 输出 (ndf\*2) x 16 x 16* nn.Conv2d(ndf \* 2, ndf \* 4, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ndf \* 4),  
 nn.LeakyReLU(0.2, inplace=**True**),  
 *# 输出 (ndf\*4) x 8 x 8* nn.Conv2d(ndf \* 4, ndf \* 8, 4, 2, 1, bias=**False**),  
 nn.BatchNorm2d(ndf \* 8),  
 nn.LeakyReLU(0.2, inplace=**True**),  
 *# 输出 (ndf\*8) x 4 x 4* nn.Conv2d(ndf \* 8, 1, 4, 1, 0, bias=**False**),  
 nn.Sigmoid() *# 输出一个数(概率)* )  
  
 **def** forward(self, input):  
 **return** self.main(input).view(-1)

**2.2.4 超参数定义Config函数：**

**class** Config(object):  
 data\_path = **'./data'** *# 数据集存放路径* num\_workers = 2 *# 多进程加载数据所用的进程数* image\_size = 96 *# 图片尺寸* batch\_size = 256  
 max\_epoch = 20  
 lr1 = 2e-4 *# 生成器的学习率* lr2 = 2e-4 *# 判别器的学习率* beta1 = 0.5 *# Adam优化器的beta1参数* gpu = **True** *# 是否使用GPU* nz = 100 *# 噪声维度* ngf = 64 *# 生成器feature map数* ndf = 64 *# 判别器feature map数* save\_path = **'imgs1/'** *# 生成图片保存路径* vis = **False** *# 是否使用visdom可视化* env = **'GAN'** *# visdom的env* plot\_every = 20 *# 每间隔20 batch，visdom画图一次* debug\_file = **'debug/debug.txt'** *# 存在该文件则进入debug模式* d\_every = 1 *# 每1个batch训练一次判别器* g\_every = 5 *# 每5个batch训练一次生成器* save\_every = 10 *# 没10个epoch保存一次模型* netd\_path = **"netd\_200.pth"** *# 'checkpoints/netd\_.pth' #预训练模型* netg\_path = **"netg\_200.pth"** *# 'checkpoints/netg\_211.pth'  
  
 # 只测试不训练* gen\_img = **'result.png'** *# 从512张生成的图片中保存最好的64张* gen\_num = 64  
 gen\_search\_num = 512  
 gen\_mean = 0 *# 噪声的均值* gen\_std = 1 *# 噪声的方差*

**2.2.5 数据读取（但是源码包含在了训练函数中）**

*# 数据*transforms = tv.transforms.Compose([  
 tv.transforms.Resize(opt.image\_size),  
 tv.transforms.CenterCrop(opt.image\_size),  
 tv.transforms.ToTensor(),  
 tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
])  
  
dataset = tv.datasets.ImageFolder(opt.data\_path, transform=transforms)  
dataloader = torch.utils.data.DataLoader(dataset,  
 batch\_size=opt.batch\_size,  
 shuffle=**True**,  
 num\_workers=opt.num\_workers,  
 drop\_last=**True** )

**2.2.6 定义模型、优化器、噪声等变量：**

*# 网络* netg, netd = NetG(opt), NetD(opt)  
 map\_location = **lambda** storage, loc: storage  
 **if** opt.netd\_path:  
 netd.load\_state\_dict(t.load(opt.netd\_path, map\_location=map\_location))  
 **if** opt.netg\_path:  
 netg.load\_state\_dict(t.load(opt.netg\_path, map\_location=map\_location))  
 netd.to(device)  
 netg.to(device)  
  
 *# 定义优化器和损失* optimizer\_g = t.optim.Adam(netg.parameters(), opt.lr1, betas=(opt.beta1, 0.999))  
 optimizer\_d = t.optim.Adam(netd.parameters(), opt.lr2, betas=(opt.beta1, 0.999))  
 criterion = t.nn.BCELoss().to(device)  
  
 *# 真图片label为1，假图片label为0  
 # noises为生成网络的输入* true\_labels = t.ones(opt.batch\_size).to(device)  
 fake\_labels = t.zeros(opt.batch\_size).to(device)  
 fix\_noises = t.randn(opt.batch\_size, opt.nz, 1, 1).to(device)  
 noises = t.randn(opt.batch\_size, opt.nz, 1, 1).to(device)  
  
 errord\_meter = AverageValueMeter()  
 errorg\_meter = AverageValueMeter()

**2.2.7 训练Train函数（除去上文已经出现过的部分）：**

**def** train(opt):  
 *# for k\_, v\_ in kwargs.items():  
 # setattr(opt, k\_, v\_)  
 # pdb.set\_trace()* device=torch.device(**'cuda:0'**) **if** opt.gpu **else** torch.device(**'cpu'**)  
 **if** opt.vis:  
 **from** visualize **import** Visualizer  
 vis = Visualizer(opt.env)  
  
 epochs = range(opt.max\_epoch)  
 **for** epoch **in** iter(epochs):  
 **for** ii, (img, \_) **in** tqdm.tqdm(enumerate(dataloader)):  
 real\_img = img.to(device)  
  
 **if** ii % opt.d\_every == 0:  
 *# 训练判别器* optimizer\_d.zero\_grad()  
 *## 尽可能的把真图片判别为正确* output = netd(real\_img)  
 error\_d\_real = criterion(output, true\_labels)  
 error\_d\_real.backward()  
 *## 尽可能把假图片判别为错误* noises.data.copy\_(torch.randn(opt.batch\_size, opt.nz, 1, 1))  
 fake\_img = netg(noises).detach() *# 根据噪声生成假图* output = netd(fake\_img)  
 error\_d\_fake = criterion(output, fake\_labels)  
 error\_d\_fake.backward()  
 optimizer\_d.step()  
 error\_d = error\_d\_fake + error\_d\_real  
 errord\_meter.add(error\_d.item())  
 **if** ii % opt.g\_every == 0:  
 *# 训练生成器* optimizer\_g.zero\_grad()  
 noises.data.copy\_(torch.randn(opt.batch\_size, opt.nz, 1, 1))  
 fake\_img = netg(noises)  
 output = netd(fake\_img)  
 error\_g = criterion(output, true\_labels)  
 error\_g.backward()  
 optimizer\_g.step()  
 errorg\_meter.add(error\_g.item())  
 **if** opt.vis **and** ii % opt.plot\_every == opt.plot\_every - 1:  
 *## 可视化* **import** os  
 **if** os.path.exists(opt.debug\_file):  
 ipdb.set\_trace()  
 fix\_fake\_imgs = netg(fix\_noises)  
 vis.images(fix\_fake\_imgs.detach().cpu().numpy()[:64] \* 0.5 + 0.5, win=**'fixfake'**)  
 vis.images(real\_img.data.cpu().numpy()[:64] \* 0.5 + 0.5, win=**'real'**)  
 vis.plot(**'errord'**, errord\_meter.value()[0])  
 vis.plot(**'errorg'**, errorg\_meter.value()[0])  
 *# 保存模型、图片* tv.utils.save\_image(fix\_fake\_imgs.data[:64], **'%s/%s.png'** % (opt.save\_path, epoch), normalize=**True**,  
 range=(-1, 1))  
 torch.save(netd.state\_dict(), **'checkpoints/netd\_%s.pth'** % epoch)  
 torch.save(netg.state\_dict(), **'checkpoints/netg\_%s.pth'** % epoch)  
 errord\_meter.reset()  
 errorg\_meter.reset()

**2.2.8 生成generate函数：**

@torch.no\_grad()  
**def** generate(opt):  
 *"""  
 随机生成动漫头像，并根据netd的分数选择较好的  
 """  
 # pdb.set\_trace()* device=torch.device(**'cuda'**) **if** opt.gpu **else** torch.device(**'cpu'**)  
 netg, netd = NetG(opt).eval(), NetD(opt).eval()  
 noises = torch.randn(opt.gen\_search\_num, opt.nz, 1, 1).normal\_(opt.gen\_mean, opt.gen\_std)  
 noises = noises.to(device)  
 map\_location = **lambda** storage, loc: storage  
 netd.load\_state\_dict(torch.load(opt.netd\_path, map\_location=map\_location))  
 netg.load\_state\_dict(torch.load(opt.netg\_path, map\_location=map\_location))  
 netd.to(device)  
 netg.to(device)  
 *# 生成图片，并计算图片在判别器的分数* fake\_img = netg(noises)  
 scores = netd(fake\_img).detach()  
 *# 挑选最好的某几张* indexs = scores.topk(opt.gen\_num)[1]  
 result = []  
 **for** ii **in** indexs:  
 result.append(fake\_img.data[ii])  
 *# 保存图片* tv.utils.save\_image(torch.stack(result), opt.gen\_img, normalize=**True**, range=(-1, 1))

1. **实验**
   1. **数据库介绍**

VAE使用的MINST数据集为手写数字集包含了60000张手写数字图片以及图片标签，是计算机网络的入门数据集

GAN使用的是二次元头像数据集，利用爬虫软件爬取原始图片，并利用OpenCV从中截取头像，将图像分辨率处理为3 \* 96 \* 96，形成大约2w张二次元头像的数据集。

**3.2 定量实验结果展示**

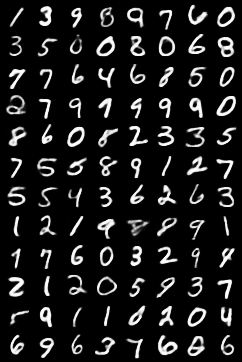
**VAE所生成的FAKE图像为：**

**Epoch1： Epoch2： Epoch3：**

**  **

**......**

**Epoch98： Epoch99：**

** **

**可见在训练的过程中生成的假的手写数字集愈加清晰和准确。**

**GAN所生成的FAKE二次元头像为：**

**Epoch1：**

****

**第一次的对抗训练生成的结果看起来有些脸有点扭曲...**

**Epoch20：**

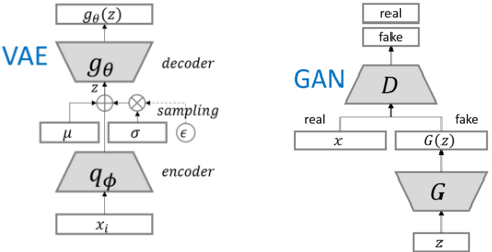
****

**在第20个epoch时生成的二次元头像虽然仍然有些出现比较扭曲的现象但整体来说效果变得好了一些。**

**其实如果epoch达到100以上生成的效果会更加可观但是由于时间和设备问题不能很快的运行出结果所以只截取了20次epoch的结果。**

**3.4 分析实验结果**

**VAE与GAN的结构差别:**



VAE训练完全依靠一个假设的loss函数和[KL-divergence](https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence)逼近真实分布:

[IMG_256](http://nooverfit.com/wp/wp-content/uploads/2017/10/Screenshot-from-2017-10-08-115443.png)

由于没有使用对抗网络，对于生成复杂的图像会更为困难而且可能会变得较为模糊。

GAN则没有假设单个loss函数, 而是让判别器D和生成器G之间进行一种零和博弈, 一方面, 生成器G要以生成假样本为目的(loss评估), 欺骗判别器D误认为是真实样本:

IMG_256

另一方面, 判别器D要以区分真实样本x和假样本G(z)为最终目的(loss评估):

[IMG_256](http://nooverfit.com/wp/wp-content/uploads/2017/10/Screenshot-from-2017-10-08-120305.png)一般, 判别器D在GAN训练中是比生成器G更强的网络, 毕竟, 网络G要从D的判别过程中学到”以假乱真”的方法. 所以, 很大程度上, G是跟着D学习的.

 所以相比其他所有模型, GAN可以产生更加清晰，真实的样本，而且GAN采用的是一种无监督的学习方式训练，可以被广泛用在无监督学习和半监督学习领域， GAN应用到一些场景上，比如图片风格迁移，超分辨率，图像补全，去噪，避免了损失函数设计的困难，不管三七二十一，只要有一个的基准，直接上判别器，剩下的就交给对抗训练了。

当然, 生成对抗网络也有一些问题, 比如经常很难训练(DCGAN试图解决), 有时候(特别是高像素图像), GAN生成图像不清晰, 还有时候, 生成图片多样性太差(只是对真实样本的简单改动).，训练GAN需要达到纳什均衡,有时候可以用梯度下降法做到,有时候做不到.我们还没有找到很好的达到纳什均衡的方法,所以训练GAN相比VAE或者PixelRNN是不稳定的,但我认为在实践中它还是比训练玻尔兹曼机稳定的多。