手写CGAN

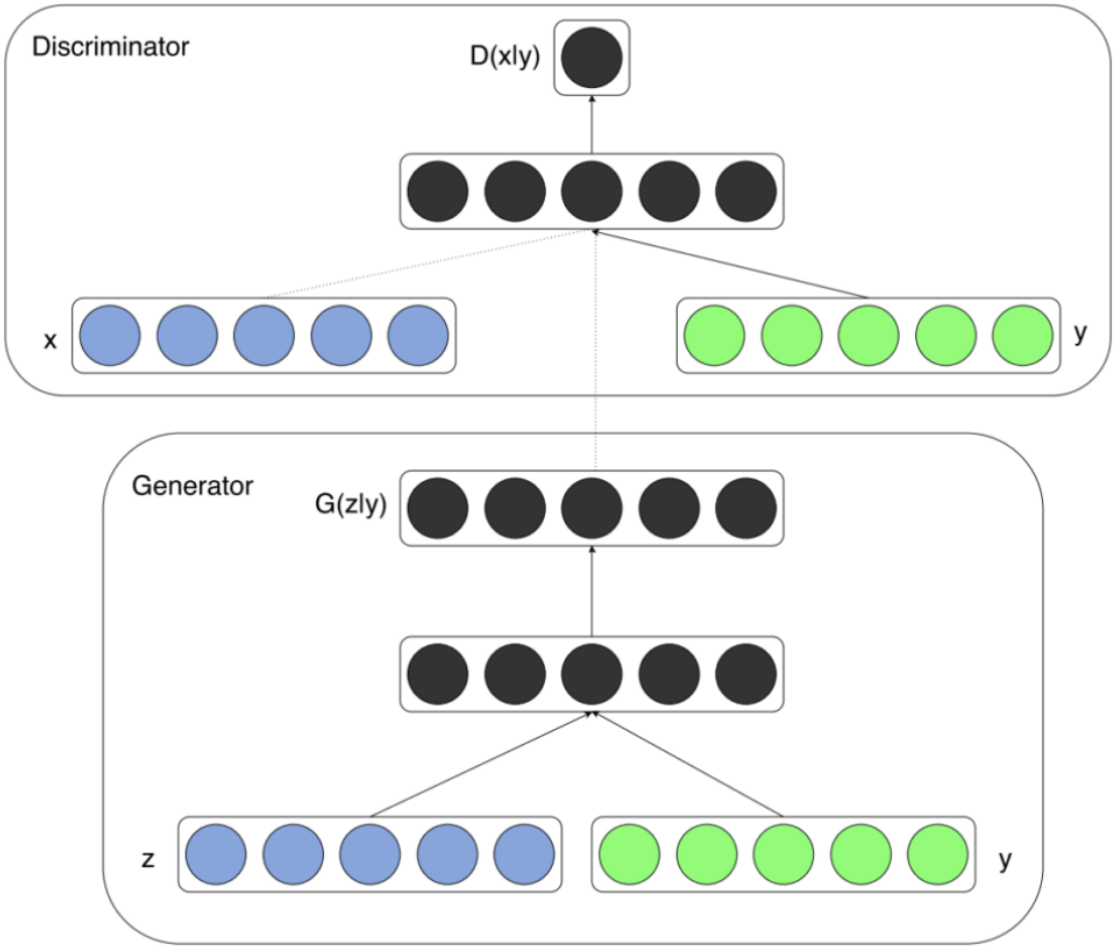
201900161098 马田慧 智能19

基本GAN思路：一个generator生成图片为了欺骗descriminatior，descriminator为了正确分辨出真的图片和假的图片。

> CGAN：

在gan训练基础上加上label信息，为啥要加上label信息呢？实验发现这样训练出的gan效果更好。怎么将label信息加上呢？在训练时将label直接拼接到原始的特征向量上。

在GAN的基础上加上label拼接的过程。



Generator

> ReLU

是将所有的负值都设为零，相反，Leaky ReLU是给所有负值赋予一个非零斜率。Leaky ReLU激活函数是在声学模型（2013）中首次提出的。以数学的方式我们可以表示为：

$$y_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ \dfrac{x_i}{a_i} & \text{if } x_i < 0, \end{cases}$$

ai是（1，+∞)区间内的固定参数。

> Tanh --激活函数

为什么要引入激活函数

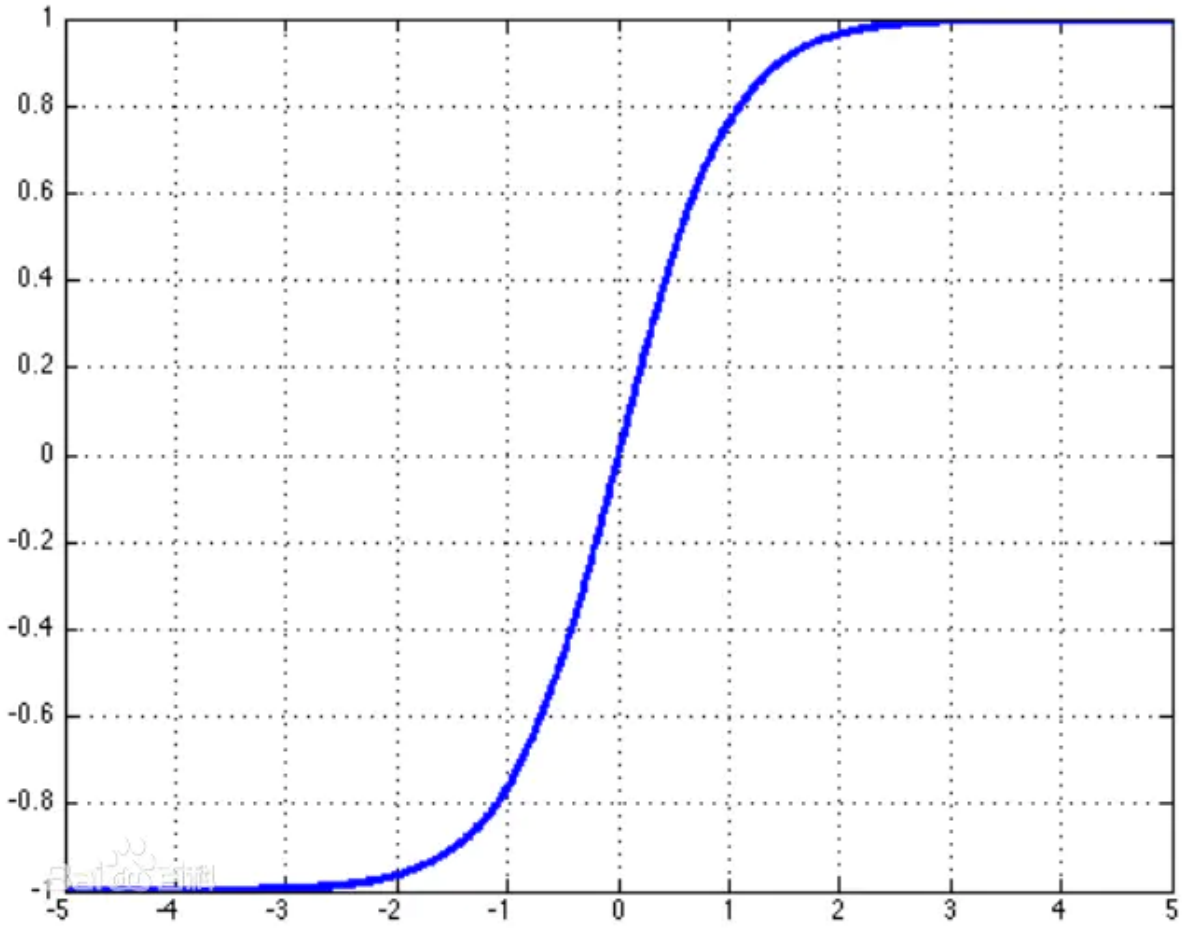
如果不用激励函数（其实相当于激励函数是f(x) = x），在这种情况下你每一层输出都是上层输入的线性函数，很容易验证，**无论你神经网络有多少层，输出都是输入的线性组合，与没有隐藏层效果相当**，这种情况就是最原始的感知机（Perceptron）了。

正因为上面的原因，我们决定引入**非线性函数作为激励函数，这样深层神经网络就有意义了（不再是输入的线性组合，可以逼近任意函数）**。最早的想法是sigmoid函数或者tanh函数，输出有界，很容易充当下一层输入（以及一些人的生物解释balabala）。激活函数的作用是为了增加神经网络模型的非线性。否则你想想，没有激活函数的每层都相当于矩阵相乘。就算你叠加了若干层之后，无非还是个矩阵相乘罢了。所以你没有非线性结构的话，根本就算不上什么神经网络。

特点

- 函数：y=tanh x；
- 定义域：R
- 值域：(-1,1)。
- y=tanh x是一个奇函数，其函数图像为过原点并且穿越Ⅰ、Ⅲ象限的严格单调递增曲线，其图像被限制在两水平渐近线y=1和y=-1之间。

图像



激活函数

Rectified Linear Unit(ReLU) - 用于隐层神经元输出
Sigmoid - 用于隐层神经元输出
Softmax - 用于多分类神经网络输出
Linear - 用于回归神经网络输出（或二分类问题）

> code

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_embedding = nn.Embedding(opt.n_classes, opt.label_dim)
        ## TODO: There are many ways to implement the model,  one alternative
        ## architecture is (100+50)--->128--->256--->512--->1024--->(1,28,28)

        ### START CODE HERE
        self.model=nn.Sequential(
            nn.Linear(150,128),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Linear(128,256),
            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()#激活函数
        )
```

```
        ### END CODE HERE

    def forward(self, noise, labels):

        ### START CODE HERE
        noise=noise.view(noise.size(0),100)#保证输入维度
        c=self.label_embedding(labels)#此的嵌入表达，将原来的01编码表示成100维的特征表
示
        x=torch.cat([noise,c],1)#cgan将噪声和标签连接作为输入
        out=self.model(x)
        img=out.view(x.size(0),28,28)

        ### END CODE HERE

        return img
```

Discriminator

最终输出对于图片判别的真假，是个分类器

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.label_embedding = nn.Embedding(opt.n_classes, opt.label_dim)
        ## TODO: There are many ways to implement the discriminator,  one
alternative
        ## architecture is (100+784)--->512--->512--->512--->1

        ### START CODE HERE
        self.model=nn.Sequential(
            nn.Linear(834,512),
            nn.LeakyReLU(0.2,inplace=True),#激活函数
            nn.Dropout(0.3), #减少过拟合
            nn.Linear(512,512),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Dropout(0.3),

            nn.Linear(512,512),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Dropout(0.3),

            nn.Linear(512,512),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512,1),#输出nx1表示真或者假
            nn.Sigmoid()#激活函数
        )
        ### END CODE HERE


    def forward(self, img, labels):

        ### START CODE HERE
        img=img.view(img.size(0),784)
        c=self.label_embedding(labels)
        # print(c.size())
```

```
        x=torch.cat([img,c],1)#输入图片与标签结合
        out=self.model(x)
        validity=out.squeeze()#维度压缩，去掉是1的维度
        ### END CODE HERE

        return validity
```

训练

定义优化器，负责更新相应的参数，每次使用之前都要清零梯度，否则会梯度积累；
`optimizer_G.zero_grad()#清零梯度`

这里若是没有优化器需要手动对所有参数更新，复杂。

```
# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr, betas=(opt.b1,
opt.b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr, betas=
(opt.b1, opt.b2))
```

控制所有新的变量在cuda中，是float或者long：

```
FloatTensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if cuda else torch.LongTensor
```

损失函数

```
adversarial_loss = torch.nn.MSELoss()
# Adversarial ground truths
valid = FloatTensor(batch_size, 1).fill_(1.0)
fake = FloatTensor(batch_size, 1).fill_(0.0)
```

训练时对于每一个batch，先训练生成器，然后训练判别器；

训练判别器时，loss分为两块，一部分是将真的输入进去的损失，这时候**期望判别器全部识别为真**，因此loss：`real_loss=adversarial_loss(real_validity,valid)` 通过与**groud truth**对比得出loss；

假的图片还是通过生成器产生，然后ground truth为fake；

```


## TODO: implement the training process

for epoch in range(opt.n_epochs):
    for i, (imgs, labels) in enumerate(dataloader):

        batch_size = imgs.shape[0]

        # Adversarial ground truths
        valid = FloatTensor(batch_size, 1).fill_(1.0)
        fake = FloatTensor(batch_size, 1).fill_(0.0)

        # Configure input
        real_imgs = imgs.type(FloatTensor)
        labels = labels.type(LongTensor)

        # -----------------
```

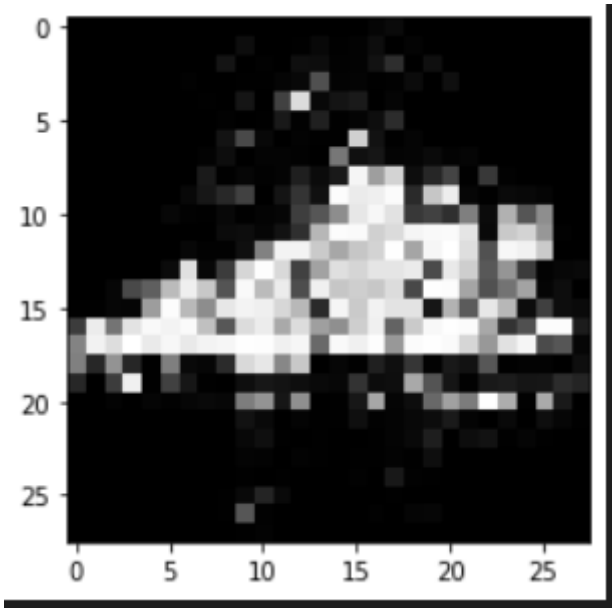
```
        #  Train Generator
        # -----------------

        ### START CODE HERE
        #训练生成器
        optimizer_G.zero_grad()#清零梯度
        noise=FloatTensor(np.random.randn(batch_size,100))#随机噪声图片
        fake_labels=LongTensor(np.random.randint(0,10,batch_size))#随机标签
        fake_imgs=generator(noise,fake_labels)#generate one
        validity=discriminator(fake_imgs,fake_labels)#用判别器判别假的图片
        g_loss=adversarial_loss(validity,valid)#最优：判别器全部认为为真，依照最优情况
计算损失
        g_loss.backward()#反向传播
        optimizer_G.step()#参数更新
        ### END CODE HERE

        # ---------------------
        #  Train Discriminator
        # ---------------------

        ### START CODE HERE
        #训练判别器 分类器
        optimizer_D.zero_grad()
        real_validity=discriminator(real_imgs,labels)
        real_loss=adversarial_loss(real_validity,valid)#真图片损失

        noise=FloatTensor(np.random.randn(batch_size,100))
        fake_labels=LongTensor(np.random.randint(0,10,batch_size))
        fake_imgs=generator(noise,fake_labels)
        fake_validity=discriminator(fake_imgs,fake_labels)
        fake_loss=adversarial_loss(fake_validity,fake)#假图片损失

        d_loss=real_loss+fake_loss
        d_loss.backward()
        optimizer_D.step()#更新
        ### END CODE HERE

        print(
            "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
            % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(),
g_loss.item())
        )
    if (epoch+1) % 20 ==0:
        torch.save(generator.state_dict(), "./cgan_generator %d.pth" % (epoch))
```

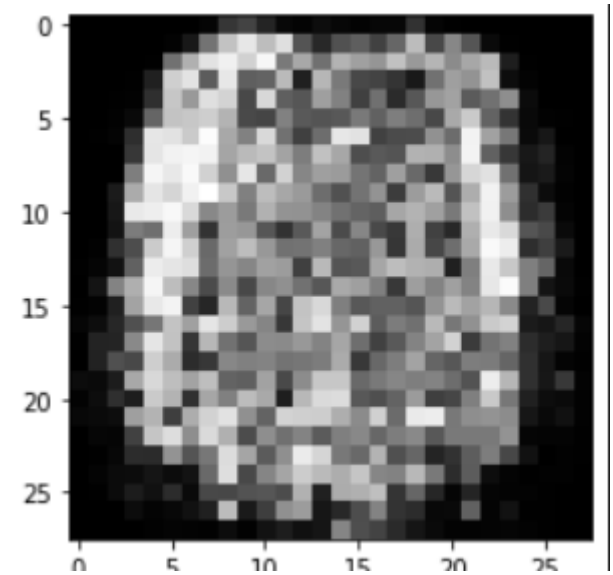
取样查看最终生成器效果

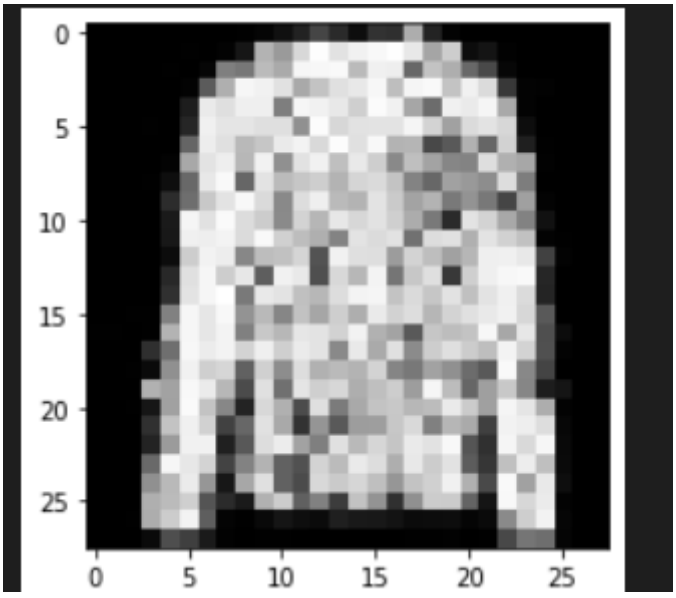
```
def generate_latent_points(latent_dim, n_samples, n_classes):
    # Sample noise

    ### START CODE HERE
    z=FloatTensor(np.random.randn(n_samples,latent_dim))
    labels=LongTensor(np.random.randint(0,n_classes,n_samples))
    ### END CODE HERE

    return z,labels
```

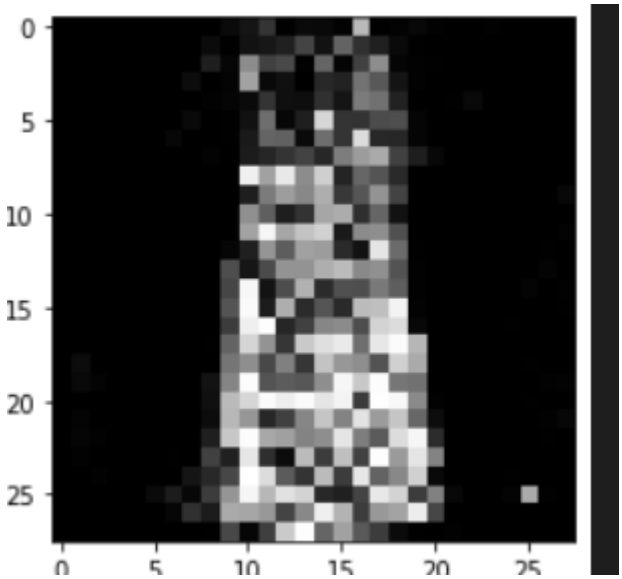
训练效果

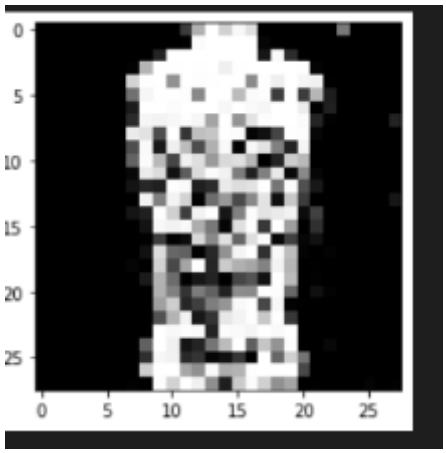
使用epoch 20次时generator生成：



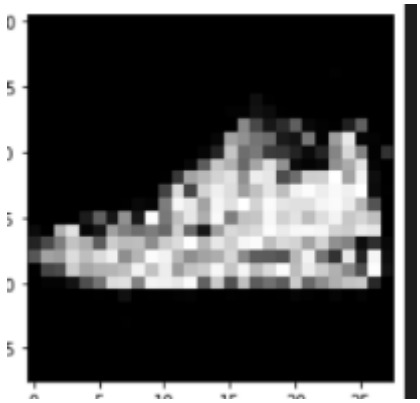


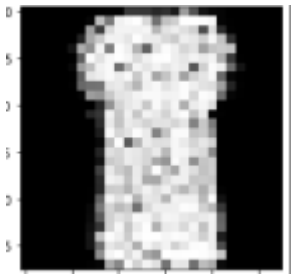


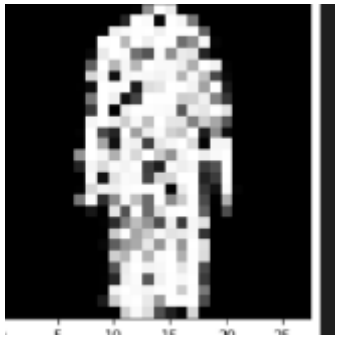


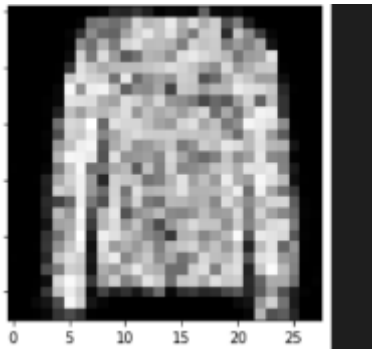


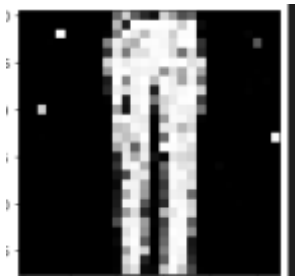
epoch200生成：

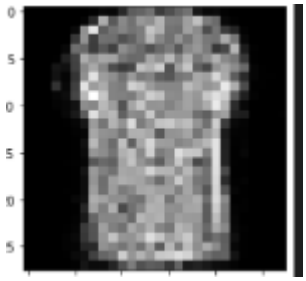












看出epoch=200生成的比19时轮廓更清楚，看起来更 像是真的衣服。

总结反思

若是尝试更多的epoch、batch可能最后将会产生更为逼真的结果