GAN 网络入门

坏人能把普通的一张白纸制成假钞，警察可以分辨出假钞和真钞。在一轮一轮的较量中，坏人制假钞的水平不断提高，而警察面对越来越难以辨认的假钞也不得不提高自己的辨别能力。

套用 GAN 网络相关术语，我们可以这样来讲这个故事：生成模型（Generative Model，坏人）可以将一个输入噪音（白纸）生成和真实数据差不多的数据（假钞），判别模型（Discriminative Model，警察）能够判断出真实数据（真钱）和类真实数据（假钞）。在一轮又一轮的博弈中，生成模型（Generative Model，坏人）能够输出非常接近真实数据的数据。

GAN 网络的目标是使得生成的数据和真实数据更接近。为了达到这个目标，一方面，我们要求 G(x)（生成模型网络）能够学习到一组很好的模型参数，使得 D(x)(判别模型网络)判别不出来真实数据和类真实数据的区别，另一方面，我们要求 D(x)(判别模型网络)的判别能力很强，能够完成对数据的真实性做出很好的二分类任务。

来自 <https://www.pianshen.com/article/3628327017/>

生成器就是一个神经网络，或看成一个函数，低维度的向量生成一个高维度的向量。训练完成后的生成器，输入向量每一个元素可以对应图片的一个象征，

来自 <https://www.jianshu.com/p/91b35a8108d2>

如上图所示，x 代表真实数据，z 代表噪音，G(z)代表一个输入噪音通过生成网络后的输出。一方面，我们希望**判别网络**能够准确判断出数据的真实性，即 D(x)尽可能接近 1，D(G(z))尽可能接近于 0；另一方面，我们希望**生成网络**产生的数据非常接近真实数据，即 D(G(z))尽可能接近于 1。

损失函数

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

可以这样理解：损失函数做的是最大化 D 的区分度，最小化 G 输出和真实数据的区别。

损失函数可以拆分为两部分：

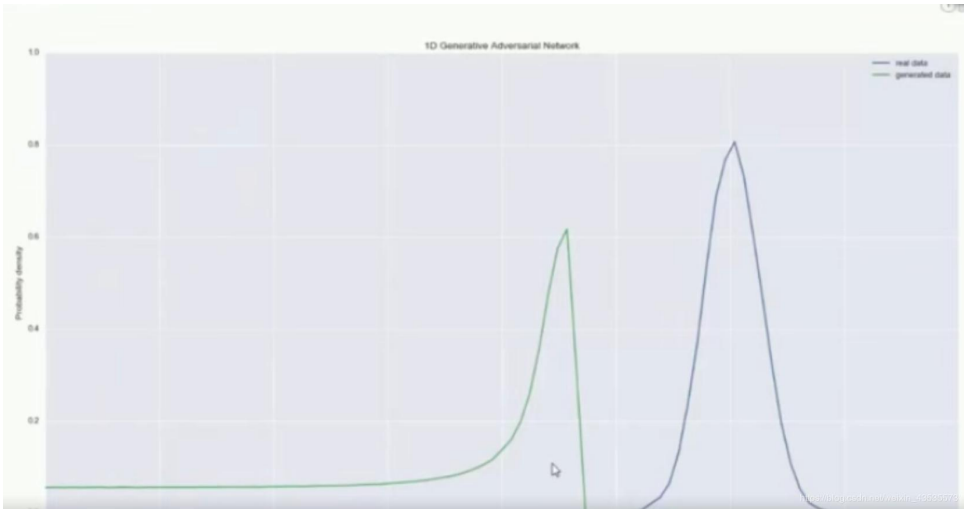
判别模型：log(D1(x))+log(1-D2(G(z)))...(1)

生成模型：log(D2(G(z)))...(2)

当判别模型能力强时，D1(x)->1，D2(G(z))->0,(1)式趋近于 0

当生成模型能力强时，D2(G(z))->1,(2)式趋近于 0

一个简单的 GAN 案例



实现功能：上图中，蓝色线代表真实数据，绿色线代表生成网络输出数据，最终我们希望绿色线能和蓝色线能够比较接近

见详细代码。

一 定义模型

```
def main(args):
    model = GAN(
        DataDistribution(),                         //真实数据真实分布
        GeneratorDistribution(range=8),      //生成数据分布
        args.num_steps,                             //迭代次数，取 1200
        args.batch_size,                            //一次迭代更新 12 个点的数据
        args.log_every,                             //每隔多少次迭代打印一次 loss 值
    )
    model.train()
```

二 参数初始化

```
def __init__(self, data, gen, num_steps, batch_size, log_every):
        self.data = data
        self.gen = gen
        self.num_steps = num_steps
        self.batch_size = batch_size
        self.log_every = log_every
        self.mlp_hidden_size = 4                    //神经网络中隐藏层个
数设置为 4
        self.learning_rate = 0.03                          //学习率
        self._create_model()
```

三 *D_pre* 网络

判别网络 *D* 的参数不能随机初始化，必须具有一定的判别能力，所以预先使用 *D_pre* 网络对 *D* 的参数进行训练。

```
    with tf.variable_scope('D_pre'):
        self.pre_input = tf.placeholder(tf.float32,
shape=(self.batch_size, 1))
        self.pre_labels = tf.placeholder(tf.float32,
shape=(self.batch_size, 1))
        D_pre = discriminator(self.pre_input, self.mlp_hidden_size)
        self.pre_loss = tf.reduce_mean(tf.square(D_pre -
self.pre_labels))
        self.pre_opt = optimizer(self.pre_loss, None,
self.learning_rate)
```

四 定义 *G,D* 网络

G 网络

```
        with tf.variable_scope('Gen'):
            self.z = tf.placeholder(tf.float32,
shape=(self.batch_size, 1))    //输入，为一个随机输入
```

```
            self.G = generator(self.z, self.mlp_hidden_size)
```

```
//产生输出数据
```

D 网络

D 网络有两个输入，一个是真实数据 x,另一个是生成网络的输出数据 G(z)，

所以定义两个:D1 和 D2。

```
        with tf.variable_scope('Disc') as scope:
            self.x = tf.placeholder(tf.float32,
shape=(self.batch_size, 1))
            self.D1 = discriminator(self.x, self.mlp_hidden_size)
            scope.reuse_variables()
            self.D2 = discriminator(self.G, self.mlp_hidden_size)
```

五 损失函数的定义（非常重要）

```
self.loss_d = tf.reduce_mean(-tf.log(self.D1) - tf.log(1 - self.D2))
```

```
//见(1)式，对(1)式取反
```

```
self.loss_g = tf.reduce_mean(-tf.log(self.D2))
```

```
//见(2)式
```

```
self.opt_d = optimizer(self.loss_d, self.d_params, self.learning_rate)
```

```
//使用优化器对两者的损失函数进行优化
```

```
self.opt_g = optimizer(self.loss_g, self.g_params,
self.learning_rate)
```

六 训练模型

```
 def train(self):
        with tf.Session() as session:
            tf.global_variables_initializer().run()
# pretraining discriminator
            num_pretrain_steps = 1000
            for step in range(num_pretrain_steps):
                d = (np.random.random(self.batch_size) - 0.5) * 10.0
                labels = norm.pdf(d, loc=self.data.mu,
```

```
scale=self.data.sigma)
                pretrain_loss, _ = session.run([self.pre_loss,
self.pre_opt], {
                    self.pre_input: np.reshape(d, (self.batch_size,
1)),
                    self.pre_labels: np.reshape(labels,
(self.batch_size, 1))
                })
            self.weightsD = session.run(self.d_pre_params)     //将
d_pre 网络的训练结果赋值
            # copy weights from pre-training over to new D network
            for i, v in enumerate(self.d_params):         //将 d_pre 网
络的训练结果拷贝给 self.d_params
                session.run(v.assign(self.weightsD[i]))

for step in range(self.num_steps):
                # update discriminator
                x = self.data.sample(self.batch_size)
//定义 x 真实数据
                z = self.gen.sample(self.batch_size)
//定义 z 噪音输入
                loss_d, _ = session.run([self.loss_d, self.opt_d], {
                    self.x: np.reshape(x, (self.batch_size, 1)),
                    self.z: np.reshape(z, (self.batch_size, 1))
                })

# update generator
                z = self.gen.sample(self.batch_size)
                loss_g, _ = session.run([self.loss_g, self.opt_g], {
                    self.z: np.reshape(z, (self.batch_size, 1))
                })

if step % self.log_every == 0:
                    print('{}: {}\t{}'.format(step, loss_d, loss_g))
                if step % 100 == 0 or step==0 or step == self.num_steps
-1 :
                    self._plot_distributions(session)
```

训练结果

经过 1200 次迭代，可以看出结果是非常好的。