GAN生成MNIST手写体

1. 数据采集

这里是通过GAN网络生成MNIST手写体图片，所以我们只需要通过tensorflow mnist api来调取MNIST图片即可，无需调取标识。这里使用tensorflow库来读取数据，并储存到一个叫MNIST_data的文件夹中。

```
from tensorflow.examples.tutorials.mnist import input_data

def load_data():
    return input_data.read_data_sets('MNIST_data/')
```

2. 识别网络架构 (discriminator)

识别网络的作用是负责识别输入的图片是来自真正的MNIST数据源还是由生成网络产生的，其本质是一个卷积网络，最后的输出是一个概率。当这个概率等于0的时候，识别网络认为输入的图片是生成的；当这个概率等于1的时候，识别网络认为输入的图片是来自真正的数据源。

识别网络的架构如下：
卷积层1：32个5x5的filter，relu activation和平均化池(average pool)
卷积层2：64个5x5的filter，relu activation和平均化池(average pool)
全联接层1: 输入[7x7x64, 1]的卷积结果，输出1024个节点，relu activation
全联接层2: 输入1024个节点，输出1个节点（概率）

```
def discriminator(image, reuse=False):
    with tf.variable_scope(tf.get_variable_scope(), reuse=reuse) as scope:
        # input 28x28x1
        # convolution layer 5x5x32
        weight1 = tf.get_variable('d_weight1', [5, 5, 1, 32], initializer=tf.truncated_normal_initializer(stddev=0.01))
        bias1 = tf.get_variable('d_bias1', [32], initializer=tf.constant_initializer(0))
        tensor1 = tf.nn.conv2d(input=image, filter=weight1, strides=[1, 1, 1, 1], padding='SAME')
        tensor1 = tensor1 + bias1
        # activation
        tensor1 = tf.nn.relu(tensor1)
        # average pooling
        tensor1 = tf.nn.avg_pool(tensor1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        # output 14x14x32

        # convolution layer 5x5x64
        weight2 = tf.get_variable('d_weight2', [5, 5, 32, 64], initializer=tf.truncated_normal_initializer(stddev=0.01))
        bias2 = tf.get_variable('d_bias2', [64], initializer=tf.constant_initializer(0))
        tensor2 = tf.nn.conv2d(input=tensor1, filter=weight2, strides=[1, 1, 1, 1], padding='SAME')
        tensor2 = tensor2 + bias2
        # activation
        tensor2 = tf.nn.relu(tensor2)
        # average pooling
        tensor2 = tf.nn.avg_pool(tensor2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        # output 7x7x64

        # fully connected layer
        weight3 = tf.get_variable('d_weight3', [7*7*64, 1024], initializer=tf.truncated_normal_initializer(stddev=0.01))
        bias3 = tf.get_variable('d_bias3', [1024], initializer=tf.constant_initializer(0))
        tensor2_flat = tf.reshape(tensor2, [-1, 7*7*64])
        tensor3 = tf.matmul(tensor2_flat, weight3) + bias3
        tensor3 = tf.nn.relu(tensor3)

        # fully connected layer
        weight4 = tf.get_variable('d_weight4', [1024, 1], initializer=tf.truncated_normal_initializer(stddev=0.01))
        bias4 = tf.get_variable('d_bias4', [1], initializer=tf.constant_initializer(0))
        tensor4 = tf.matmul(tensor3, weight4) + bias4
        return tensor4
```

这里有几点说明：

在程序的开头用到了variable scope。由于我们在训练过程中，我们需要识别网络既要接收生成的图片，也要接收真正的图片，所以我们必须定义识别网络两次。但我们要保证这两次定义的是一个网络，也就是说，这里的variable scope和reuse是保证网络的变量在这两次定义中互相分享，从而避免创建两个不同的识别网络。

这里的所有weight和bias全部有命名(比如d_weight1),之所以命名是因为我们在后面训练识别网络和生成网络时候，我们需要针对这个网络的变量进行训练。换句话说，在训练识别网络时候我们只更新识别网络的变量。这里命名方便我们区分两个网络的变量。

3. 生成网络的架构 (generator)

生成网络的目的是将输入的随机噪音向量变成类似MNIST手写体。与CNN不同，生成网络的输出是图片。其基本概念是将输入不停缩小和放大，提取重要的特征。

由于MNIST手写体是28x28的图片，生成网络一开始将输入的噪音向量变成56x56的图片。这里通过矩阵乘法，将[1, noise_dim]的向量与[noise_dim, 3136]的向量相乘，并将结果reshape成[56,56]的图片，使用batch normalization保证数据的variance不变，方便后面层次训练。使用relu激活层。

```
filter_num = 56*56

def generator(noise, noise_dim):
    weight1 = tf.get_variable('g_weight1', [noise_dim, filter_num], dtype=tf.float32,
                              initializer=tf.truncated_normal_initializer(stddev=0.01))
    bias1 = tf.get_variable('g_bias1', [filter_num], dtype=tf.float32,
                            initializer=tf.truncated_normal_initializer(stddev=0.01))
    tensor1 = tf.matmul(noise, weight1) + bias1
    tensor1 = tf.reshape(tensor1, [-1, 56, 56, 1])
    tensor1 = tf.contrib.layers.batch_norm(tensor1, epsilon=1e-5, scope='g_bias1')
    tensor1 = tf.nn.relu(tensor1)
```

随后生成网络会进行两次卷积和反卷积，每一次卷积都是使用3x3的filter，卷积出的结果是28x28. 第二次卷积用的filter数量比第一次少一半，以此循循渐进找到重要特征。每次卷积过后都有一个反卷积层，将图片放大回56x56. 放大时空缺的点使用bicubic方法复原，bicubic计算周围16个点的加权平均。注意的是我们也可以使用更快的bilinear，但bicubic方法对放大是更好的。

```
# convolution 3x3 kernel size
# use noise_dim / 2 filters
weight2 = tf.get_variable('g_weight2', [3, 3, 1, noise_dim/2],
                          initializer=tf.truncated_normal_initializer(stddev=0.01))
bias2 = tf.get_variable('g_bias2', [noise_dim/2], initializer=tf.truncated_normal_initializer(stddev=0.01))
tensor2 = tf.nn.conv2d(input=tensor1, filter=weight2, strides=[1, 2, 2, 1], padding='SAME')
tensor2 = tensor2 + bias2
tensor2 = tf.contrib.layers.batch_norm(tensor2, epsilon=1e-5, scope='g_bias2')
tensor2 = tf.nn.relu(tensor2)
# use bicubic to enlarge
tensor2 = tf.image.resize_images(tensor2, [56, 56], method=tf.image.ResizeMethod.BICUBIC)

# convolution 3x3 kernel size
# use noise_dim / 4 filters
weight3 = tf.get_variable('g_weight3', [3, 3, noise_dim/2, noise_dim/4],
                          initializer=tf.truncated_normal_initializer(stddev=0.01))
bias3 = tf.get_variable('g_bias3', [noise_dim/4], initializer=tf.truncated_normal_initializer(stddev=0.01))
tensor3 = tf.nn.conv2d(input=tensor2, filter=weight3, strides=[1, 2, 2, 1], padding='SAME')
tensor3 = tensor3 + bias3
tensor3 = tf.contrib.layers.batch_norm(tensor3, epsilon=1e-5, scope='g_bias3')
tensor3 = tf.nn.relu(tensor3)
# use bicubic to enlarge
tensor3 = tf.image.resize_images(tensor3, [56, 56], method=tf.image.ResizeMethod.BICUBIC)
```

生成网络的最后一层是一个卷积层，使用1个1x1的filter卷积，得到一个28x28的图片。最后使用sigmoid函数保证图片里面的数值在0和1之间(图片pixel的允许值)。

```
# final convolution to generate image
weight4 = tf.get_variable('g_weight4', [1, 1, noise_dim/4, 1],
                          initializer=tf.truncated_normal_initializer(stddev=0.01))
bias4 = tf.get_variable('g_bias4', [1], initializer=tf.truncated_normal_initializer(stddev=0.01))
tensor4 = tf.nn.conv2d(input=tensor3, filter=weight4, strides=[1, 2, 2, 1], padding='SAME')
tensor4 = tensor4 + bias4
tensor4 = tf.sigmoid(tensor4)
return tensor4
```

4.训练

建立好模型后我们要开始训练步骤，首先定义hyperparameters，这里的hyperparameters我们需要：generator的learning rate，discriminator的learning rate，噪音向量大小，批尺寸，识别网络预热训练loop，训练loop。我们会在后文提到识别网络预热问题。

```
# hyperparameters
learning_rate_g = 0.0001
learning_rate_d = 0.0003
noise_dim = 100
batch_size = 50
discriminator_pre_train_loop = 300
train_loop = 2000
```

接着我们需要定义placeholder和模型。这里我们需要两个placeholder，一个放噪音向量，一个放真正的MNIST数据源图片。在前文提到过，我们需要让识别网络既接收噪音向量，又接收真正的数据源图片，因此我们需要定义两个识别网络的模型，但第二个模型的reuse为true，意味着和第一个模型同时分享网络中的变量。

```
# placeholder for noise
noise_placeholder = tf.placeholder(tf.float32, [None, noise_dim], name='noise_placeholder')
# placeholder for real image
x_placeholder = tf.placeholder(tf.float32, [None, 28, 28, 1], name='x_placeholder')

# generate new images
generated = generator(noise_placeholder, noise_dim)

# discriminator for real MNIST dataset
d_real = discriminator(x_placeholder)
# discriminator for generated data
d_fake = discriminator(generated, reuse=True)
```

紧接着我们需要定义loss函数，在这里我们定义三个loss。第一个loss是识别网络对真正MNIST数据源图片的loss，我们希望识别网络这个时候的输出越接近1越好。第二个loss是识别网络对人工生产的图片的loss，我们希望识别网络这个时候的输出越接近0越好。第三个loss是生成网络的loss，生成网络的目的是生成让识别网络觉得是来自真正的数据源图片的人工图片，即让识别网络接收人工图片的输出越接近1越好。所以我们的定义如下。

```
# define loss function
# real loss -> probability as close to 1 as possible
d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_real, labels=tf.ones_like(d_real)))
# generated loss -> probability as close to 0 as possible
d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_fake, labels=tf.zeros_like(d_fake)))
# generator loss -> wants to let discriminator output 1 as much as possible for generated images
g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_fake, labels=tf.ones_like(d_fake)))
```

最后我们需要定义优化器，在优化器中我们需要区分出来优化哪些变量，这也是为什么识别网络的变量都是d开头，生成网络变量都是g开头。我们通过var_list参数保证识别网络不会更新生成网络的参数，生成网络也不会更新识别网络的参数。

```
# get trainable variables
variables = tf.trainable_variables()
discriminator_var = [var for var in variables if 'd_' in var.name]
generator_var = [var for var in variables if 'g_' in var.name]

# optimizer
discriminator_train_real = tf.train.AdamOptimizer(learning_rate=learning_rate_d)\
    .minimize(d_loss_real, var_list=discriminator_var)
discriminator_train_fake = tf.train.AdamOptimizer(learning_rate=learning_rate_d)\
    .minimize(d_loss_fake, var_list=discriminator_var)
generator_train = tf.train.AdamOptimizer(learning_rate=learning_rate_g).minimize(g_loss, var_list=generator_var)
```

定义完之后就要开一个tensorflow的session开始训练，由于GAN训练速度很慢，最好办法是储存训练的结果，在下次训练时调出继续训练。这里实现是通过session saver，具体方法如下。

```
tf.get_variable_scope().reuse_variables()
# open a session
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
# restore model from previous training - we can train 100 loops at first, then
# we run another 100 loops. With following restore code we train a total of 200 loops
# COMMENT OUT IF YOU WANT TO TRAIN FROM BEGINNING
saver.restore(sess, 'model/mnist_gan.ckpt')
```

这里是假设之前训练了一个模型出存在model/mnist_gan.ckpt里面，在这调出继续训练。

在正式开始训练之前，一般会预热识别网络。我们希望在训练生成网络时候识别网络能有一些识别能力。所以这里用到识别网络预热loop，先训练一下识别网络。噪音向量是通过随机函数构造的。

```
print('pre-train discriminator...')
# pre-train discriminator
for i in range(0, discriminator_pre_train_loop):
    # create noise vector
    noise = np.random.normal(0, 1, size=[batch_size, noise_dim])
    real_images = mnist.train.next_batch(batch_size)[0].reshape([batch_size, 28, 28, 1])
    # pretrain on both real and fake images
    _, __, d_loss_real, d_loss_fake = sess.run([discriminator_train_real, discriminator_train_fake,
                                                d_loss_real, d_loss_fake],
                                               feed_dict={x_placeholder: real_images, noise_placeholder: noise})
    # need to cast back to tensor, otherwise there would be an error
    d_loss_real = tf.cast(d_loss_real, tf.float32)
    d_loss_fake = tf.cast(d_loss_fake, tf.float32)
```

一般这个loop比较少，在300左右即可。

最后我们开始整个训练，在一个loop中我们先训练识别网络，然后训练生成网络，形成两个网络的对抗。

```
print('training...')
# training
for i in range(0, train_loop):
    noise = np.random.normal(0, 1, size=[batch_size, noise_dim])
    real_images = mnist.train.next_batch(batch_size)[0].reshape([batch_size, 28, 28, 1])
    _, __, d_loss_real, d_loss_fake = sess.run([discriminator_train_real, discriminator_train_fake,
                                                d_loss_real, d_loss_fake],
                                               feed_dict={x_placeholder: real_images, noise_placeholder: noise})
    # train generator
    noise = np.random.normal(0, 1, size=[batch_size, noise_dim])
    sess.run(generator_train, feed_dict={x_placeholder: real_images, noise_placeholder: noise})
    d_loss_real = tf.cast(d_loss_real, tf.float32)
    d_loss_fake = tf.cast(d_loss_fake, tf.float32)
    # save each 100 loops
    if i % 100 == 0 and i != 0:
        print(str(i))
        saver.save(sess, 'model/mnist_gan.ckpt')
```

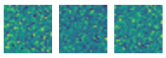
训练结束后切记储存训练好的模型，方便下次使用。

5.结果

在累积训练了1万个循环后（近4个小时），我们得到一下结果。第一行是真正MNIST的图，第二行是训练前生成网络产生的图，第三行是训练后生成网络产生的图。

Real MNIST Images

Generated MNIST Images Before Training



Generated MNIST Images After Training

可以看出，生成网络还是有点缺陷，但可以通过增加训练次数获得更好结果。

常见问题：

识别网络击败生成网络：当识别网络完全能分辨出图片来源时，即输出永远是1或者0，我们无法计算网络的gradient，则不能更新和优化网络，造成训练失败。一个解决办法是我们不将识别网络的输出规定在0和1，在识别网络的最后一层不使用sigmoid函数，保证输出的值有gradient可以计算。

生成网络击败识别网络：当生成网络发现识别网络的弱点，生成网络会产生相似的图，这些图识别网络都无法正确识别这是来自人工生成的。一个解决办法是加强识别网络，增加层数或者调节learning rate。