DCGAN 手写数字生成

这个文档里会列出代码里最重要的一些部分进行讲解。

代码主要分为两部分: (1)计算图构建 `build_model`; (2)模型训练 `train`。

计算图构建

构建输入节点

```
input_ph = tf.placeholder(
  dtype=tf.float32,
  shape=[hp.batch_size, hp.image_height, hp.image_width, 1],
  name="input_image")
z_ph = tf.placeholder(
  tf.float32, [hp.batch_size, hp.z_dim], name="z")
```

`input_ph`为输入图片入口，作为判别器D的真实图片输入。

`z_ph`为输入噪音入口，作为生成器G的输入。

`tf.placeholder(dtype, shape=None, name=None)` 函数在计算图中定义一个占位符，三个参数分别是数据类型，tensor的形状，和名字。

构建判别器D

D的模型结构见讲义，具体实现见如下代码：

```
def _discriminator(x, is_training, hp):
  with tf.variable_scope("D", reuse=tf.AUTO_REUSE):

    # layer_1
    _x = _leaky_relu(_conv(x, 64, "D_conv_1"))

    # layer_2
    _x = _conv(_x, 128, name="D_conv_2")
    _x = _leaky_relu(_batch_norm(
      _x, is_training, scope="D_bn_2"
    ))

    # layer_3
    _x = tf.reshape(_x, [hp.batch_size, -1])
    _x = _linear(_x, 1024, name="D_linear_3")
    _x = _leaky_relu(_batch_norm(
      _x, is_training, scope="D_bn_3"
    ))
```

```
    # layer_4
    out_logit = _linear(_x, 1, name="D_linear4")
    return out_logit
```

is_training为一个boolean，表示训练状态，True表示正在训练，False表示工作状态。

with tf.variable_scope(name="D", reuse=tf.AUTO_REUSE): 是一个常用语句，在它的作用域里定义的参数会在前面加上一个name的前缀，便于管理参数。 例如，在这个作用域里定义一个变量名为"x"，那么它的名字最终为"..../D/x"，类似于文件系统的目录，比如把判别器D的参数都放到"D"目录里。

在判别器中，输入图片通过卷积、线性变换等操作，最终输出节点 out_logit，Sigmoid(out_logit) 表示它是真实图片的概率。

_conv是一个自定义函数，表示一个卷积操作，包含了参数定义和卷积操作的计算图连接:

```
def _conv(_input, out_dim, name):
  with tf.variable_scope(name):
    w = tf.get_variable(
      name="w",
      shape=[4, 4, _input.get_shape()[-1], out_dim],
      initializer=tf.truncated_normal_initializer(stddev=0.02)
    )
    res = tf.nn.conv2d(_input, w, strides=[1,2,2,1], padding="SAME")
    bias = tf.get_variable("bias", [out_dim],
      initializer=tf.constant_initializer(0.0))
    res = tf.nn.bias_add(res, bias)
    return res
```

注意，这里scope名字设置为变量，scope名字相同就意味着参数共享。因为模型里只有一个判别器，所以D的scope可以写死为"D"，而卷积有很多层，就需要设置为变量。

tf.get_variable是一个创建变量的函数，主要用于创建模型参数；这个函数一般要给定这三个参数：

- name： 变量的名字，也是变量的id
- shape： 变量的维度
- initializer: 初始化函数，告诉变量如何初始化

对于2维卷积操作，我们需要构建一个[卷积核高度,卷积核宽度,输入维度，输出维度]的参数矩阵；并且用一个截断的正态分布来初始化参数。

initializer=tf.truncated_normal_initializer(stddev=0.02) 表示一个截断的正态分布来初始化函数，一般使用两个参数：

- mean： 表示均值
- stddev： 表示标准差

定义完参数后再定义卷积操作：

```
res = tf.nn.conv2d(input=_input, filter=w, strides=[1,2,2,1], padding="SAME")
```

- input： 表示输入节点
- filter： 表示卷积核
- strides： 表示在输入上的移动窗口的移动步长
- padding： 表示使用何种padding算法

最后再定义一个偏置参数bias，卷积后结果加上bias就是最终的卷积操作， 最后返回的结果就是卷积操作的计算图的输出节点。

_linear是一个线性变换操作:

```
def _linear(_input, out_dim, name):
  shape = _input.get_shape().as_list()
  with tf.variable_scope(name):
    w = tf.get_variable("w", [shape[-1], out_dim],
      initializer=tf.truncated_normal_initializer(stddev=0.02))
    b = tf.get_variable("bias", [out_dim],
      initializer=tf.constant_initializer(out_dim))
    return tf.matmul(_input, w) + b
```

这个实现非常简单，就是与一个矩阵相乘进行线性变换。

构建生成器G

生成器G与判别器D正好相反，即用线性变换和反卷积将一个低维的随机变量z（从噪声里采样）生成一个28x28x1的手写图片，即长宽为28个像素的黑白图片。

```
def _generator(z, is_training, hp):
  with tf.variable_scope("G", reuse=tf.AUTO_REUSE):

    # layer_1
    _x = _linear(z, 1024, name="G_linear_1")
    _x = tf.nn.relu(_batch_norm(
      _x, is_training, scope="G_bn_1"
    ))

    # layer_2
    _x = _linear(_x, 128*7*7, name="G_linear_2")
    _x = tf.nn.relu(_batch_norm(
      _x, is_training, scope="G_bn_2"
    ))

    # layer_3
    _x = tf.reshape(_x, [hp.batch_size, 7, 7, 128])
    _x = _dconv(_x, [hp.batch_size, 14, 14, 64], name="G_dconv_3")
    _x = tf.nn.relu(_batch_norm(
      _x, is_training, scope="G_bn_3"
    ))
```

```
    # layer_4
    out = tf.nn.sigmoid(_dconv(
      _x,
      [hp.batch_size, hp.image_height, hp.image_width, 1],
      name="G_dconv_4"))

    return out
```

其中_dconv是自定义的反卷积函数：

```
def _dconv(_input, out_dim, name):
  with tf.variable_scope(name):
    w = tf.get_variable(
      name="w",
      shape=[4, 4, out_dim[-1], _input.get_shape()[-1]],
      initializer=tf.truncated_normal_initializer(stddev=0.02)
    )
    res = tf.nn.conv2d_transpose(
      _input, w, output_shape=out_dim, strides=[1,2,2,1])
    bias = tf.get_variable("bias", [out_dim[-1]],
      initializer=tf.constant_initializer(0.0))
    res = tf.nn.bias_add(res, bias)
    return res
```

构建DCGAN模型

根据生成器和判别器的输出，计算出它们的目标函数值。

```
# output of D for real images
D_real_logits = _discriminator(input_ph, is_training=True, hp=hp)
# output of D for fake images
fake_input = _generator(z_ph, is_training=True, hp=hp)
D_fake_logits = _discriminator(fake_input, is_training=True, hp=hp)

D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
  logits=D_real_logits, labels=tf.ones_like(D_real_logits)))
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
  logits=D_fake_logits, labels=tf.zeros_like(D_fake_logits)))
D_loss = D_loss_real + D_loss_fake

G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
  logits=D_fake_logits, labels=tf.ones_like(D_fake_logits)))
```

使用Adam优化器构建更新操作节点：

```

```

```
D_vars = [var for var in tf.trainable_variables() if
var.name.startswith("D")]
G_vars = [var for var in tf.trainable_variables() if
var.name.startswith("G")]

D_optim = tf.train.AdamOptimizer(hp.lr, hp.beta1).minimize(D_loss,
var_list=D_vars)
G_optim = tf.train.AdamOptimizer(hp.lr*5, hp.beta1).minimize(G_loss,
var_list=G_vars)
```

构建摘要节点，用于显示目标函数值和训练过程中输出的图片。

```
D_loss_sum = tf.summary.scalar("D_loss", D_loss)
D_real_loss_sum = tf.summary.scalar("D_real_loss", D_loss_real)
D_fake_loss_sum = tf.summary.scalar("D_fake_loss", D_loss_fake)
G_loss_sum = tf.summary.scalar("G_loss", G_loss)

fake_images = _generator(z_ph, is_training=False, hp=hp)
G_image_sum = tf.summary.image("G_images", fake_images, max_outputs=10)
```

模型训练

进行hp.epoch次循环，在每个epoch中，遍历整个数据集，每次传给模型一个batch的数据进行训练，并将摘要写到log中：

```
for epoch in range(hp.epoch):
  print("\n========================\nepoch", epoch)
  for idx in range(0, len(dataset), hp.batch_size):
    X = dataset[idx:idx+hp.batch_size]
    if len(X) != hp.batch_size: break
    z = np.random.uniform(-1, 1, size=(hp.batch_size, hp.z_dim))
    feed = {input_ph: X, z_ph:z}
    # train D
    _, summary1, summary2, summary3 = sess.run([D_optim, D_real_loss_sum,
D_fake_loss_sum, D_loss_sum], feed_dict=feed)
    writer.add_summary(summary1, global_step)
    writer.add_summary(summary2, global_step)
    writer.add_summary(summary3, global_step)
    # train G
    _, summary = sess.run([G_optim, G_loss_sum], feed_dict=feed)
    writer.add_summary(summary, global_step)
    global_step += 1
```

参数配置

模型中需要配置若干超参数，然后将超参数传入train函数进行训练：

```
hp = HParams(
    # batch 大小
    batch_size=256,
    # 图片高度
    image_height=28,
    # 图片宽度
    image_width=28,
    # 生成器G输入的随机变量的维度
    z_dim=100,
    # 学习率，影响参数更新速度
    lr=0.0002,
    # 优化器参数
    beta1=0.5,
    # log目录
    logdir="./log/DCGAN",
    # 训练轮数
    epoch=100,
    # 数据集目录
    dataset_dir=os.path.join(HOME, "res", "mnist")
  )
train(hp)
```