DCGAN 手写数字生成

In [ ]:
```
import tensorflow as tf
import os, gzip
import numpy as np

from tensorflow.contrib.training import HParams
```

读取数据/数据预处理：

In [ ]:
```
def load_mnist(data_dir):

  def extract_data(filename, num_data, head_size, data_size):
    with gzip.open(filename) as bytestream:
      bytestream.read(head_size)
      buf = bytestream.read(data_size * num_data)
      data = np.frombuffer(buf, dtype=np.uint8).astype(np.float)
    return data

  data = extract_data(data_dir + '/train-images-idx3-ubyte.gz', 60000, 16, 28 * 28)
  trX = data.reshape((60000, 28, 28, 1))

  data = extract_data(data_dir + '/train-labels-idx1-ubyte.gz', 60000, 8, 1)
  trY = data.reshape((60000))

  data = extract_data(data_dir + '/t10k-images-idx3-ubyte.gz', 10000, 16, 28 * 28)
  teX = data.reshape((10000, 28, 28, 1))

  data = extract_data(data_dir + '/t10k-labels-idx1-ubyte.gz', 10000, 8, 1)
  teY = data.reshape((10000))

  trY = np.asarray(trY)
  teY = np.asarray(teY)

  X = np.concatenate((trX, teX), axis=0)
  y = np.concatenate((trY, teY), axis=0).astype(np.int)

  seed = 547
  np.random.seed(seed)
  np.random.shuffle(X)
  np.random.seed(seed)
  np.random.shuffle(y)

  y_vec = np.zeros((len(y), 10), dtype=np.float)
  for i, label in enumerate(y):
    y_vec[i, y[i]] = 1.0

  return X / 255., y_vec
```

神经网络基本组件定义

Batch Normlization:

一个加速神经网络训练，防止梯度消失的层。

In [ ]:
```
def _batch_norm(x, is_training, scope):
  return tf.contrib.layers.batch_norm(
    x,
    decay=0.9,
    updates_collections=None,
    epsilon=1e-5,
    scale=True,
    is_training=is_training,
    scope=scope)
```

Leaky ReLU

激活层，引入非线性，防止ReLU失效的一种ReLU的变体。

In [ ]:
```
def _leaky_relu(x):
  return tf.nn.leaky_relu(x, alpha=0.2)
```

卷积层

`_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:

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。

`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。

In [ ]:
```
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
```

反卷积层

与卷积层正好相反，将卷积操作替换成反卷积函数 `tf.nn.conv2d_transpose` ， 表示与卷积相反的操作。

In [ ]:
```python
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
```

线性层:

`_linear` 是一个线性变换操作： `wx+b`

In [ ]:
```python
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
```

构建模型计算图

判别器D:

输入一个28x28x1的手写数字图片，判别器判断图片是否为真实图片

In [ ]:
```python
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
```

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的手写数字图片，即长宽为28个像素的黑白图片。

In [ ]:
```
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
```

构建模型：

`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。

根据生成器输出fake_input和判别器输出D_real_logits/D_fake_logits，计算出它们的目标函数值D_loss和G_loss。

然后，使用Adam优化器构建更新操作节点D_optim和G_optim。

最后将loss和生成的图片通过summary的形式输出。

In [ ]:
```
def build_model(hp):
  #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")

  # 给判别器输入真实图片，获取其输出（概率）
  D_real_logits = _discriminator(input_ph, is_training=True, hp=hp)
  # 让生成器G生成伪造的手写数字图片
  fake_input = _generator(z_ph, is_training=True, hp=hp)
  # 给判别器输入G生成的伪造图片，获取其输出（概率）
  D_fake_logits = _discriminator(fake_input, is_training=True, hp=hp)

  # 计算目标函数
  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)))


  # 获取判别器的所有参数
  D_vars = [var for var in tf.trainable_variables() if var.name.startswith("D")]
  # 获取生成器的所有参数
  G_vars = [var for var in tf.trainable_variables() if var.name.startswith("G")]

  # 构建判别器的优化操作
  D_optim = tf.train.AdamOptimizer(hp.lr, hp.beta1).minimize(D_loss, var_list=D_vars)
  # 构建生成器的优化操作
  G_optim = tf.train.AdamOptimizer(hp.lr*5, hp.beta1).minimize(G_loss, var_list=G_var
s)

  # 构建摘要节点
  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)

  return D_optim, G_optim, D_loss_sum, G_loss_sum, fake_images, G_image_sum, \
    input_ph, z_ph, D_real_loss_sum, D_fake_loss_sum
```

配置超参数

In [ ]:
```
HOME = os.getenv("HOME")
hp = HParams(
  batch_size=256,
  image_height=28,
  image_width=28,
  z_dim=100,
  lr=0.0002,
  beta1=0.5,
  logdir="./log/DCGAN",
  epoch=100,
  #dataset_dir=os.path.join(HOME, "res", "mnist"),
  dataset_dir="./mnist"
)
```

训练模型：

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：

In [ ]:
```
D_optim, G_optim, D_loss_sum, G_loss_sum, fake_images, G_image_sum, \
input_ph, z_ph, D_real_loss_sum, D_fake_loss_sum = build_model(hp)

dataset, _ = load_mnist(hp.dataset_dir)
global_step = 1

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter(hp.logdir, graph=sess.graph)

    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

      z = np.random.uniform(-1, 1, size=(hp.batch_size, hp.z_dim))
      image_sum = sess.run(G_image_sum, feed_dict={z_ph:z})
      writer.add_summary(image_sum, epoch)
```