这个文件就是DCGAN模型定义的函数。调用了utils.py文件和ops.py文件。

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

from \_future\_ import division 这句话当python的版本为2.x时生效，可以让两个整数数字相除的结果返回一个浮点数(在python2中默认是整数,python3默认为浮点数)。glob可以以简单的正则表达式筛选的方式返回某个文件夹下符合要求的文件名列表

import os

import time

import math

from glob import glob

import tensorflow as tf

import numpy as np

from six.moves import xrange

from ops import \*

from utils import \*

定义conv\_out\_size\_same(size, stride)函数。大小和步幅

def conv\_out\_size\_same(size, stride):

return int(math.ceil(float(size) / float(stride)))

def gen\_random(mode, size):

if mode=='normal01': return np.random.normal(0,1,size=size)

if mode=='uniform\_signed': return np.random.uniform(-1,1,size=size)

if mode=='uniform\_unsigned': return np.random.uniform(0,1,size=size)

然后是定义了DCGAN类，剩余代码都是在写DCGAN类，所以下面几步都是在这个类里面定义进行的

class DCGAN(object):

定义类的初始化函数 init。主要是对一些默认的参数进行初始化。包括session、crop、批处理大小batch\_size、样本数量sample\_num、输入与输出的高和宽、各种维度、生成器与判别器的批处理、数据集名字、灰度值、构建模型函数，需要注意的是，要判断数据集的名字是否是mnist，是的话则直接用load\_mnist()函数加载数据，否则需要从本地data文件夹中读取数据，并将图像读取为灰度图

def \_\_init\_\_(self, sess, input\_height=108, input\_width=108, crop=True,

batch\_size=64, sample\_num = 64, output\_height=64, output\_width=64,

y\_dim=None, z\_dim=100, gf\_dim=64, df\_dim=64,

gfc\_dim=1024, dfc\_dim=1024, c\_dim=3, dataset\_name='default',

max\_to\_keep=1,

input\_fname\_pattern='\*.jpg', checkpoint\_dir='ckpts', sample\_dir='samples', out\_dir='./out', data\_dir='./data'):

"""

Args:

sess: TensorFlow session

batch\_size: The size of batch. Should be specified before training.

y\_dim: (optional) Dimension of dim for y. [None]

z\_dim: (optional) Dimension of dim for Z. [100]

gf\_dim: (optional) Dimension of gen filters in first conv layer. [64]

df\_dim: (optional) Dimension of discrim filters in first conv layer. [64]

gfc\_dim: (optional) Dimension of gen units for for fully connected layer. [1024]

dfc\_dim: (optional) Dimension of discrim units for fully connected layer. [1024]

c\_dim: (optional) Dimension of image color. For grayscale input, set to 1. [3]

"""

self.sess = sess

self.crop = crop

self.batch\_size = batch\_size

self.sample\_num = sample\_num

self.input\_height = input\_height

self.input\_width = input\_width

self.output\_height = output\_height

self.output\_width = output\_width

self.y\_dim = y\_dim

self.z\_dim = z\_dim

self.gf\_dim = gf\_dim

self.df\_dim = df\_dim

self.gfc\_dim = gfc\_dim

self.dfc\_dim = dfc\_dim

# batch normalization : deals with poor initialization helps gradient flow

self.d\_bn1 = batch\_norm(name='d\_bn1')

self.d\_bn2 = batch\_norm(name='d\_bn2')

if not self.y\_dim:

self.d\_bn3 = batch\_norm(name='d\_bn3')

self.g\_bn0 = batch\_norm(name='g\_bn0')

self.g\_bn1 = batch\_norm(name='g\_bn1')

self.g\_bn2 = batch\_norm(name='g\_bn2')

if not self.y\_dim:

self.g\_bn3 = batch\_norm(name='g\_bn3')

self.dataset\_name = dataset\_name

self.input\_fname\_pattern = input\_fname\_pattern

self.checkpoint\_dir = checkpoint\_dir

self.data\_dir = data\_dir

self.out\_dir = out\_dir

self.max\_to\_keep = max\_to\_keep

data\_path = os.path.join(self.data\_dir, self.dataset\_name, self.input\_fname\_pattern)

self.data = glob(data\_path)

if len(self.data) == 0:

raise Exception("[!] No data found in '" + data\_path + "'")

np.random.shuffle(self.data)

imreadImg = imread(self.data[0])

if len(imreadImg.shape) >= 3: #check if image is a non-grayscale image by checking channel number

self.c\_dim = imread(self.data[0]).shape[-1]

else:

self.c\_dim = 1

if len(self.data) < self.batch\_size:

raise Exception("[!] Entire dataset size is less than the configured batch\_size")

self.grayscale = (self.c\_dim == 1)

self.build\_model()

定义构建模型函数build\_model(self)。

首先判断y\_dim，然后用tf.placeholder占位符定义并初始化y。

判断crop是否为真，是的话是进行测试，图像维度是输出图像的维度；否则是输入图像的维度。

利用tf.placeholder定义inputs，是真实数据的向量。

定义并初始化生成器用到的噪音z，z\_sum。

再次判断y\_dim，如果为真，用噪音z和标签y初始化生成器G、用输入inputs初始化判别器D和D\_logits、样本、用G和y初始化D\_和D\_logits；如果为假，跟上面一样初始化各种变量，只不过都没有标签y。

将5中的D、D\_、G分别放在d\_sum、d\_\_sum、G\_sum。

定义sigmoid交叉熵损失函数sigmoid\_cross\_entropy\_with\_logits(x, y)。都是调用tf.nn.sigmoid\_cross\_entropy\_with\_logits函数，只不过一个是训练，y是标签，一个是测试，y是目标。

定义各种损失值。真实数据的判别损失值d\_loss\_real、虚假数据的判别损失值d\_loss\_fake、生成器损失值g\_loss、判别器损失值d\_loss。

定义训练的所有变量t\_vars。

定义生成和判别的参数集。

最后是保存。

def build\_model(self):

if self.y\_dim:

self.y = tf.placeholder(tf.float32, [self.batch\_size, self.y\_dim], name='y')

else:

self.y = None

if self.crop:

image\_dims = [self.output\_height, self.output\_width, self.c\_dim]

else:

image\_dims = [self.input\_height, self.input\_width, self.c\_dim]

self.inputs = tf.compat.v1.placeholder(

tf.float32, [self.batch\_size] + image\_dims, name='real\_images')

inputs = self.inputs

self.z = tf.compat.v1.placeholder(

tf.float32, [None, self.z\_dim], name='z')

self.z\_sum = histogram\_summary("z", self.z)

self.g\_loss 是生成器损失; self.d\_loss\_real 是真实图片的鉴别器损失; self.d\_loss\_fake 是虚假图片(由生成器生成的fake images)的损失; self.d\_loss 是总的鉴别器损失。

这里的 histogram\_summary 和 scalar\_summary 是为了在后续在tensorboard中对各个损失函数进行可视化

self.G = self.generator(self.z, self.y)

self.generator 用于构造生成器; self.discriminator 用于构造鉴别器; self.sampler 用于随机采样(用于生成样本)。这里需要注意的是, self.y 只有当dataset是mnist的时候才不为None,不是mnist的情况下,只需要 self.z 即可生成samples

self.D, self.D\_logits = self.discriminator(inputs, self.y, reuse=False)

self.sampler = self.sampler(self.z, self.y)

self.D\_, self.D\_logits\_ = self.discriminator(self.G, self.y, reuse=True)

self.d\_sum = histogram\_summary("d", self.D)

self.d\_\_sum = histogram\_summary("d\_", self.D\_)

self.G\_sum = image\_summary("G", self.G)

sigmoid\_cross\_entropy\_with\_logits 函数被重新定义了，是为了兼容不同版本的tensorflow。该函数首先使用sigmoid activation，然后计算cross-entropy los

def sigmoid\_cross\_entropy\_with\_logits(x, y):

try:

return tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=x, labels=y)

except:

return tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=x, targets=y)

self.d\_loss\_real = tf.reduce\_mean(

sigmoid\_cross\_entropy\_with\_logits(self.D\_logits, tf.ones\_like(self.D)))

self.d\_loss\_fake = tf.reduce\_mean(

sigmoid\_cross\_entropy\_with\_logits(self.D\_logits\_, tf.zeros\_like(self.D\_)))

self.g\_loss = tf.reduce\_mean(

sigmoid\_cross\_entropy\_with\_logits(self.D\_logits\_, tf.ones\_like(self.D\_)))

self.d\_loss\_real\_sum = scalar\_summary("d\_loss\_real", self.d\_loss\_real)

self.d\_loss\_fake\_sum = scalar\_summary("d\_loss\_fake", self.d\_loss\_fake)

self.d\_loss = self.d\_loss\_real + self.d\_loss\_fake

self.g\_loss\_sum = scalar\_summary("g\_loss", self.g\_loss)

self.d\_loss\_sum = scalar\_summary("d\_loss", self.d\_loss)

tf.trainable\_variables() 可以获取model的全部可训练参数,由于我们在定义生成器和鉴别器变量的时候使用了不同的name,因此我们可以通过variable的name来获取得到self.d\_vars(鉴别器相关变量),self.g\_vars(生成器相关变量)。 self.saver = tf.train.Saver() 用于保存训练好的模型参数到checkpoint

t\_vars = tf.compat.v1.trainable\_variables()

self.d\_vars = [var for var in t\_vars if 'd\_' in var.name]

self.g\_vars = [var for var in t\_vars if 'g\_' in var.name]

self.saver = tf.compat.v1.train.Saver(max\_to\_keep=self.max\_to\_keep)

定义训练函数train(self, config)。

定义判别器优化器d\_optim和生成器优化器g\_optim。

变量初始化。

分别将关于生成器和判别器有关的变量各合并到一个变量中，并写入事件文件中。

噪音z初始化。

根据数据集是否为mnist的判断，进行输入数据和标签的获取。这里使用到了utils.py文件中的get\_image函数。

定义计数器counter和起始时间start\_time。

加载检查点，并判断加载是否成功。

开始for epoch in xrange(config.epoch)循环训练。先判断数据集是否是mnist，获取批处理的大小。

开始for idx in xrange(0, batch\_idxs)循环训练，判断数据集是否是mnist，来定义初始化批处理图像和标签。

定义初始化噪音z。

判断数据集是否是mnist，来更新判别器网络和生成器网络，这里就不管mnist数据集是怎么处理的，其他数据集是，运行生成器优化器两次，以确保判别器损失值不会变为0，然后是判别器真实数据损失值和虚假数据损失值、生成器损失值。

输出本次批处理中训练参数的情况，首先是第几个epoch，第几个batch，训练时间，判别器损失值，生成器损失值。

每100次batch训练后，根据数据集是否是mnist的不同，获取样本、判别器损失值、生成器损失值，调用utils.py文件的save\_images函数，保存训练后的样本，并以epoch、batch的次数命名文件。然后打印判别器损失值和生成器损失值。

每500次batch训练后，保存一次检查点。

step5：定义判别器函数discriminator(self, image, y=None, reuse=False)。

利用with tf.variable\_scope(“discriminator”) as scope，在一个作用域 scope 内共享一些变量。

对scope利用reuse\_variables()进行重利用。

如果为假，则直接设置5层，前4层为使用lrelu激活函数的卷积层，最后一层是使用线性层，最后返回h4和sigmoid处理后的h4。

如果为真，则首先将Y\_dim变为yb，然后利用ops.py文件中的conv\_cond\_concat函数，连接image与yb得到x，然后设置4层网络，前3层是使用lrelu激励函数的卷积层，最后一层是线性层，最后返回h3和sigmoid处理后的h3。

def train(self, config):

d\_optim = tf.compat.v1.train.AdamOptimizer(config.learning\_rate, beta1=config.beta1) \

.minimize(self.d\_loss, var\_list=self.d\_vars)

g\_optim = tf.compat.v1.train.AdamOptimizer(config.learning\_rate, beta1=config.beta1) \

.minimize(self.g\_loss, var\_list=self.g\_vars)

try:

tf.compat.v1.global\_variables\_initializer().run()

except:

tf.initialize\_all\_variables().run()

if config.G\_img\_sum:

self.g\_sum = merge\_summary([self.z\_sum, self.d\_\_sum, self.G\_sum, self.d\_loss\_fake\_sum, self.g\_loss\_sum])

else:

self.g\_sum = merge\_summary([self.z\_sum, self.d\_\_sum, self.d\_loss\_fake\_sum, self.g\_loss\_sum])

self.d\_sum = merge\_summary(

[self.z\_sum, self.d\_sum, self.d\_loss\_real\_sum, self.d\_loss\_sum])

self.writer = SummaryWriter(os.path.join(self.out\_dir, "logs"), self.sess.graph)

sample\_z = gen\_random(config.z\_dist, size=(self.sample\_num , self.z\_dim))

sample\_files = self.data[0:self.sample\_num]

sample = [

get\_image(sample\_file,

input\_height=self.input\_height,

input\_width=self.input\_width,

resize\_height=self.output\_height,

resize\_width=self.output\_width,

crop=self.crop,

grayscale=self.grayscale) for sample\_file in sample\_files]

if (self.grayscale):

sample\_inputs = np.array(sample).astype(np.float32)[:, :, :, None]

else:

sample\_inputs = np.array(sample).astype(np.float32)

counter = 1

start\_time = time.time()

could\_load, checkpoint\_counter = self.load(self.checkpoint\_dir)

if could\_load:

counter = checkpoint\_counter

print(" [\*] Load SUCCESS")

else:

print(" [!] Load failed...")

for epoch in xrange(config.epoch):

self.data = glob(os.path.join(

config.data\_dir, config.dataset, self.input\_fname\_pattern))

np.random.shuffle(self.data)

batch\_idxs = min(len(self.data), config.train\_size) // config.batch\_size

for idx in xrange(0, int(batch\_idxs)):

batch\_files = self.data[idx\*config.batch\_size:(idx+1)\*config.batch\_size]

batch = [

get\_image(batch\_file,

input\_height=self.input\_height,

input\_width=self.input\_width,

resize\_height=self.output\_height,

resize\_width=self.output\_width,

crop=self.crop,

grayscale=self.grayscale) for batch\_file in batch\_files]

if self.grayscale:

batch\_images = np.array(batch).astype(np.float32)[:, :, :, None]

else:

batch\_images = np.array(batch).astype(np.float32)

batch\_z = gen\_random(config.z\_dist, size=[config.batch\_size, self.z\_dim]) \

.astype(np.float32)

# Update D network

\_, summary\_str = self.sess.run([d\_optim, self.d\_sum],

feed\_dict={ self.inputs: batch\_images, self.z: batch\_z })

self.writer.add\_summary(summary\_str, counter)

# Update G network

\_, summary\_str = self.sess.run([g\_optim, self.g\_sum],

feed\_dict={ self.z: batch\_z })

self.writer.add\_summary(summary\_str, counter)

# Run g\_optim twice to make sure that d\_loss does not go to zero (different from paper)

\_, summary\_str = self.sess.run([g\_optim, self.g\_sum],

feed\_dict={ self.z: batch\_z })

self.writer.add\_summary(summary\_str, counter)

errD\_fake = self.d\_loss\_fake.eval({ self.z: batch\_z })

errD\_real = self.d\_loss\_real.eval({ self.inputs: batch\_images })

errG = self.g\_loss.eval({self.z: batch\_z})

print("[%8d Epoch:[%2d/%2d] [%4d/%4d] time: %4.4f, d\_loss: %.8f, g\_loss: %.8f" \

% (counter, epoch, config.epoch, idx, batch\_idxs,

time.time() - start\_time, errD\_fake+errD\_real, errG))

if np.mod(counter, config.sample\_freq) == 0:

try:

samples, d\_loss, g\_loss = self.sess.run(

[self.sampler, self.d\_loss, self.g\_loss],

feed\_dict={

self.z: sample\_z,

self.inputs: sample\_inputs,

},

)

save\_images(samples, image\_manifold\_size(samples.shape[0]),

'./{}/train\_{:08d}.png'.format(config.sample\_dir, counter))

print("[Sample] d\_loss: %.8f, g\_loss: %.8f" % (d\_loss, g\_loss))

except:

print("one pic error!...")

if np.mod(counter, config.ckpt\_freq) == 0:

self.save(config.checkpoint\_dir, counter)

counter += 1

def discriminator(self, image, y=None, reuse=False):

with tf.compat.v1.variable\_scope("discriminator") as scope:

if reuse:

scope.reuse\_variables()

if not self.y\_dim:

h0 = lrelu(conv2d(image, self.df\_dim, name='d\_h0\_conv'))

h1 = lrelu(self.d\_bn1(conv2d(h0, self.df\_dim\*2, name='d\_h1\_conv')))

h2 = lrelu(self.d\_bn2(conv2d(h1, self.df\_dim\*4, name='d\_h2\_conv')))

h3 = lrelu(self.d\_bn3(conv2d(h2, self.df\_dim\*8, name='d\_h3\_conv')))

h4 = linear(tf.reshape(h3, [self.batch\_size, -1]), 1, 'd\_h4\_lin')

return tf.nn.sigmoid(h4), h4

else:

yb = tf.reshape(y, [self.batch\_size, 1, 1, self.y\_dim])

x = conv\_cond\_concat(image, yb)

h0 = lrelu(conv2d(x, self.c\_dim + self.y\_dim, name='d\_h0\_conv'))

h0 = conv\_cond\_concat(h0, yb)

h1 = lrelu(self.d\_bn1(conv2d(h0, self.df\_dim + self.y\_dim, name='d\_h1\_conv')))

h1 = tf.reshape(h1, [self.batch\_size, -1])

h1 = concat([h1, y], 1)

h2 = lrelu(self.d\_bn2(linear(h1, self.dfc\_dim, 'd\_h2\_lin')))

h2 = concat([h2, y], 1)

h3 = linear(h2, 1, 'd\_h3\_lin')

return tf.nn.sigmoid(h3), h3

定义生成器函数generator(self, z, y=None)。

利用with tf.variable\_scope(“generator”) as scope，在一个作用域 scope 内共享一些变量。

根据y\_dim是否为真，进行判别网络的设置。

如果为假：首先获取输出的宽和高，然后根据这一值得到更多不同大小的高和宽的对。然后获取h0层的噪音z，权值w，偏置值b，然后利用relu激励函数。h1层，首先对h0层解卷积得到本层的权值和偏置值，然后利用relu激励函数。h2、h3等同于h1。h4层，解卷积h3，然后直接返回使用tanh激励函数后的h4。

如果为真：首先也是获取输出的高和宽，根据这一值得到更多不同大小的高和宽的对。然后获取yb和噪音z。h0层，使用relu激励函数，并与1连接。h1层，对线性全连接后使用relu激励函数，并与yb连接。h2层，对解卷积后使用relu激励函数，并与yb连接。最后返回解卷积、sigmoid处理后的h2

def generator(self, z, y=None):

with tf.compat.v1.variable\_scope("generator") as scope:

if not self.y\_dim:

s\_h, s\_w = self.output\_height, self.output\_width

s\_h2, s\_w2 = conv\_out\_size\_same(s\_h, 2), conv\_out\_size\_same(s\_w, 2)

s\_h4, s\_w4 = conv\_out\_size\_same(s\_h2, 2), conv\_out\_size\_same(s\_w2, 2)

s\_h8, s\_w8 = conv\_out\_size\_same(s\_h4, 2), conv\_out\_size\_same(s\_w4, 2)

s\_h16, s\_w16 = conv\_out\_size\_same(s\_h8, 2), conv\_out\_size\_same(s\_w8, 2)

self.z\_, self.h0\_w, self.h0\_b = linear(

z, self.gf\_dim\*8\*s\_h16\*s\_w16, 'g\_h0\_lin', with\_w=True)

self.h0 = tf.reshape(

self.z\_, [-1, s\_h16, s\_w16, self.gf\_dim \* 8])

h0 = tf.nn.relu(self.g\_bn0(self.h0))

self.h1, self.h1\_w, self.h1\_b = deconv2d(

h0, [self.batch\_size, s\_h8, s\_w8, self.gf\_dim\*4], name='g\_h1', with\_w=True)

h1 = tf.nn.relu(self.g\_bn1(self.h1))

h2, self.h2\_w, self.h2\_b = deconv2d(

h1, [self.batch\_size, s\_h4, s\_w4, self.gf\_dim\*2], name='g\_h2', with\_w=True)

h2 = tf.nn.relu(self.g\_bn2(h2))

h3, self.h3\_w, self.h3\_b = deconv2d(

h2, [self.batch\_size, s\_h2, s\_w2, self.gf\_dim\*1], name='g\_h3', with\_w=True)

h3 = tf.nn.relu(self.g\_bn3(h3))

h4, self.h4\_w, self.h4\_b = deconv2d(

h3, [self.batch\_size, s\_h, s\_w, self.c\_dim], name='g\_h4', with\_w=True)

return tf.nn.tanh(h4)

else:

s\_h, s\_w = self.output\_height, self.output\_width

s\_h2, s\_h4 = int(s\_h/2), int(s\_h/4)

s\_w2, s\_w4 = int(s\_w/2), int(s\_w/4)

yb = tf.reshape(y, [self.batch\_size, 1, 1, self.y\_dim])

z = concat([z, y], 1)

h0 = tf.nn.relu(

self.g\_bn0(linear(z, self.gfc\_dim, 'g\_h0\_lin')))

h0 = concat([h0, y], 1)

h1 = tf.nn.relu(self.g\_bn1(

linear(h0, self.gf\_dim\*2\*s\_h4\*s\_w4, 'g\_h1\_lin')))

h1 = tf.reshape(h1, [self.batch\_size, s\_h4, s\_w4, self.gf\_dim \* 2])

h1 = conv\_cond\_concat(h1, yb)

h2 = tf.nn.relu(self.g\_bn2(deconv2d(h1,

[self.batch\_size, s\_h2, s\_w2, self.gf\_dim \* 2], name='g\_h2')))

h2 = conv\_cond\_concat(h2, yb)

return tf.nn.sigmoid(

deconv2d(h2, [self.batch\_size, s\_h, s\_w, self.c\_dim], name='g\_h3'))

定义sampler(self, z, y=None)函数。

利用tf.variable\_scope(“generator”) as scope，在一个作用域 scope 内共享一些变量。

对scope利用reuse\_variables()进行重利用。

根据y\_dim是否为真，进行判别网络的设置。

然后就跟生成器差不多，不在赘述。

def sampler(self, z, y=None):

with tf.compat.v1.variable\_scope("generator") as scope:

scope.reuse\_variables()

if not self.y\_dim:

s\_h, s\_w = self.output\_height, self.output\_width

s\_h2, s\_w2 = conv\_out\_size\_same(s\_h, 2), conv\_out\_size\_same(s\_w, 2)

s\_h4, s\_w4 = conv\_out\_size\_same(s\_h2, 2), conv\_out\_size\_same(s\_w2, 2)

s\_h8, s\_w8 = conv\_out\_size\_same(s\_h4, 2), conv\_out\_size\_same(s\_w4, 2)

s\_h16, s\_w16 = conv\_out\_size\_same(s\_h8, 2), conv\_out\_size\_same(s\_w8, 2)

h0 = tf.reshape(

linear(z, self.gf\_dim\*8\*s\_h16\*s\_w16, 'g\_h0\_lin'),

[-1, s\_h16, s\_w16, self.gf\_dim \* 8])

h0 = tf.nn.relu(self.g\_bn0(h0, train=False))

h1 = deconv2d(h0, [self.batch\_size, s\_h8, s\_w8, self.gf\_dim\*4], name='g\_h1')

h1 = tf.nn.relu(self.g\_bn1(h1, train=False))

h2 = deconv2d(h1, [self.batch\_size, s\_h4, s\_w4, self.gf\_dim\*2], name='g\_h2')

h2 = tf.nn.relu(self.g\_bn2(h2, train=False))

h3 = deconv2d(h2, [self.batch\_size, s\_h2, s\_w2, self.gf\_dim\*1], name='g\_h3')

h3 = tf.nn.relu(self.g\_bn3(h3, train=False))

h4 = deconv2d(h3, [self.batch\_size, s\_h, s\_w, self.c\_dim], name='g\_h4')

return tf.nn.tanh(h4)

else:

s\_h, s\_w = self.output\_height, self.output\_width

s\_h2, s\_h4 = int(s\_h/2), int(s\_h/4)

s\_w2, s\_w4 = int(s\_w/2), int(s\_w/4)

yb = tf.reshape(y, [self.batch\_size, 1, 1, self.y\_dim])

z = concat([z, y], 1)

h0 = tf.nn.relu(self.g\_bn0(linear(z, self.gfc\_dim, 'g\_h0\_lin'), train=False))

h0 = concat([h0, y], 1)

h1 = tf.nn.relu(self.g\_bn1(

linear(h0, self.gf\_dim\*2\*s\_h4\*s\_w4, 'g\_h1\_lin'), train=False))

h1 = tf.reshape(h1, [self.batch\_size, s\_h4, s\_w4, self.gf\_dim \* 2])

h1 = conv\_cond\_concat(h1, yb)

h2 = tf.nn.relu(self.g\_bn2(

deconv2d(h1, [self.batch\_size, s\_h2, s\_w2, self.gf\_dim \* 2], name='g\_h2'), train=False))

h2 = conv\_cond\_concat(h2, yb)

return tf.nn.sigmoid(deconv2d(h2, [self.batch\_size, s\_h, s\_w, self.c\_dim], name='g\_h3'))

@property

定义model\_dir(self)函数。返回数据集名字，batch大小，输出的高和宽

def model\_dir(self):

return "{}\_{}\_{}\_{}".format(

self.dataset\_name, self.batch\_size,

self.output\_height, self.output\_width)

定义save(self, checkpoint\_dir, step)函数。保存训练好的模型。创建检查点文件夹，如果路径不存在，则创建；然后将其保存在这个文件夹下

def save(self, checkpoint\_dir, step, filename='model', ckpt=True, frozen=False):

# model\_name = "DCGAN.model"

# checkpoint\_dir = os.path.join(checkpoint\_dir, self.model\_dir)

filename += '.b' + str(self.batch\_size)

if not os.path.exists(checkpoint\_dir):

os.makedirs(checkpoint\_dir)

if ckpt:

self.saver.save(self.sess,

os.path.join(checkpoint\_dir, filename),

global\_step=step)

if frozen:

tf.train.write\_graph(

tf.graph\_util.convert\_variables\_to\_constants(self.sess, self.sess.graph\_def, ["generator\_1/Tanh"]),

checkpoint\_dir,

'{}-{:06d}\_frz.pb'.format(filename, step),

as\_text=False)

定义load(self, checkpoint\_dir)函数。读取检查点，获取路径，重新存储检查点，并且计数。打印成功读取的提示；如果没有路径，则打印失败的提示

def load(self, checkpoint\_dir):

#import re

print(" [\*] Reading checkpoints...", checkpoint\_dir)

# checkpoint\_dir = os.path.join(checkpoint\_dir, self.model\_dir)

# print(" ->", checkpoint\_dir)

ckpt = tf.train.get\_checkpoint\_state(checkpoint\_dir)

if ckpt and ckpt.model\_checkpoint\_path:

ckpt\_name = os.path.basename(ckpt.model\_checkpoint\_path)

self.saver.restore(self.sess, os.path.join(checkpoint\_dir, ckpt\_name))

#counter = int(next(re.finditer("(\d+)(?!.\*\d)",ckpt\_name)).group(0))

counter = int(ckpt\_name.split('-')[-1])

print(" [\*] Success to read {}".format(ckpt\_name))

return True, counter

else:

print(" [\*] Failed to find a checkpoint")

return False, 0

以上，就是model.py所有内容，主要是定义了DCGAN的类，完成了生成判别网络的实现