# -\*- coding: utf-8 -\*-

# generate new kinds of pokemons

import os

import tensorflow as tf

import numpy as np

import cv2

import random

import scipy.misc

from utils import \*

slim = tf.contrib.slim

HEIGHT, WIDTH, CHANNEL = 128, 128, 3

BATCH\_SIZE = 64

EPOCH = 5000

os.environ['CUDA\_VISIBLE\_DEVICES'] = '15'

version = 'newPokemon'

newPoke\_path = './' + version

def lrelu(x, n, leak=0.2):

return tf.maximum(x, leak \* x, name=n)

def process\_data():

current\_dir = os.getcwd()

# parent = os.path.dirname(current\_dir)

pokemon\_dir = os.path.join(current\_dir, 'resized\_black')

images = []

for each in os.listdir(pokemon\_dir):

images.append(os.path.join(pokemon\_dir,each))

# print images

all\_images = tf.convert\_to\_tensor(images, dtype = tf.string)

images\_queue = tf.train.slice\_input\_producer(

[all\_images])

content = tf.read\_file(images\_queue[0])

image = tf.image.decode\_jpeg(content, channels = CHANNEL)

# sess1 = tf.Session()

# print sess1.run(image)

image = tf.image.random\_flip\_left\_right(image)

image = tf.image.random\_brightness(image, max\_delta = 0.1)

image = tf.image.random\_contrast(image, lower = 0.9, upper = 1.1)

# noise = tf.Variable(tf.truncated\_normal(shape = [HEIGHT,WIDTH,CHANNEL], dtype = tf.float32, stddev = 1e-3, name = 'noise'))

# print image.get\_shape()

size = [HEIGHT, WIDTH]

image = tf.image.resize\_images(image, size)

image.set\_shape([HEIGHT,WIDTH,CHANNEL])

# image = image + noise

# image = tf.transpose(image, perm=[2, 0, 1])

# print image.get\_shape()

image = tf.cast(image, tf.float32)

image = image / 255.0

iamges\_batch = tf.train.shuffle\_batch(

[image], batch\_size = BATCH\_SIZE,

num\_threads = 4, capacity = 200 + 3\* BATCH\_SIZE,

min\_after\_dequeue = 200)

num\_images = len(images)

return iamges\_batch, num\_images

def generator(input, random\_dim, is\_train, reuse=False):

c4, c8, c16, c32, c64 = 512, 256, 128, 64, 32 # channel num

s4 = 4

output\_dim = CHANNEL # RGB image

with tf.variable\_scope('gen') as scope:

if reuse:

scope.reuse\_variables()

w1 = tf.get\_variable('w1', shape=[random\_dim, s4 \* s4 \* c4], dtype=tf.float32,

initializer=tf.truncated\_normal\_initializer(stddev=0.02))

b1 = tf.get\_variable('b1', shape=[c4 \* s4 \* s4], dtype=tf.float32,

initializer=tf.constant\_initializer(0.0))

flat\_conv1 = tf.add(tf.matmul(input, w1), b1, name='flat\_conv1')

# 4\*4\*512

conv1 = tf.reshape(flat\_conv1, shape=[-1, s4, s4, c4], name='conv1')

bn1 = tf.contrib.layers.batch\_norm(conv1, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn1')

act1 = tf.nn.relu(bn1, name='act1')

# 8\*8\*256

conv2 = tf.layers.conv2d\_transpose(act1, c8, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv2')

bn2 = tf.contrib.layers.batch\_norm(conv2, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn2')

act2 = tf.nn.relu(bn2, name='act2')

# 16\*16\*128

conv3 = tf.layers.conv2d\_transpose(act2, c16, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv3')

bn3 = tf.contrib.layers.batch\_norm(conv3, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn3')

act3 = tf.nn.relu(bn3, name='act3')

# 32\*32\*64

conv4 = tf.layers.conv2d\_transpose(act3, c32, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv4')

bn4 = tf.contrib.layers.batch\_norm(conv4, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn4')

act4 = tf.nn.relu(bn4, name='act4')

# 64\*64\*32

conv5 = tf.layers.conv2d\_transpose(act4, c64, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv5')

bn5 = tf.contrib.layers.batch\_norm(conv5, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn5')

act5 = tf.nn.relu(bn5, name='act5')

#128\*128\*3

conv6 = tf.layers.conv2d\_transpose(act5, output\_dim, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv6')

# bn6 = tf.contrib.layers.batch\_norm(conv6, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn6')

act6 = tf.nn.tanh(conv6, name='act6')

return act6

def discriminator(input, is\_train, reuse=False):

c2, c4, c8, c16 = 64, 128, 256, 512 # channel num: 64, 128, 256, 512

with tf.variable\_scope('dis') as scope:

if reuse:

scope.reuse\_variables()

# 64\*64\*64

conv1 = tf.layers.conv2d(input, c2, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv1')

# bn1 = tf.contrib.layers.batch\_norm(conv1, is\_training = is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope = 'bn1')

act1 = lrelu(conv1, n='act1')

# 32\*32\*128

conv2 = tf.layers.conv2d(act1, c4, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv2')

bn2 = tf.contrib.layers.batch\_norm(conv2, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn2')

act2 = lrelu(bn2, n='act2')

# 16\*16\*256

conv3 = tf.layers.conv2d(act2, c8, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv3')

bn3 = tf.contrib.layers.batch\_norm(conv3, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn3')

act3 = lrelu(bn3, n='act3')

# 8\*8\*512

conv4 = tf.layers.conv2d(act3, c16, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

name='conv4')

bn4 = tf.contrib.layers.batch\_norm(conv4, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn4')

act4 = lrelu(bn4, n='act4')

# # 8\*8\*256

# conv5 = tf.layers.conv2d(act4, c32, kernel\_size=[5, 5], strides=[2, 2], padding="SAME",

# kernel\_initializer=tf.truncated\_normal\_initializer(stddev=0.02),

# name='conv5')

# bn5 = tf.contrib.layers.batch\_norm(conv5, is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bn5')

# act5 = lrelu(bn5, n='act5')

# start from act4

dim = int(np.prod(act4.get\_shape()[1:]))

fc1 = tf.reshape(act4, shape=[-1, dim], name='fc1')

# w1 = tf.get\_variable('w1', shape=[fc1.shape[-1], 512], dtype=tf.float32,

# initializer=tf.truncated\_normal\_initializer(stddev=0.02))

# b1 = tf.get\_variable('b1', shape=[512], dtype=tf.float32,

# initializer=tf.constant\_initializer(0.0))

# bnf = tf.contrib.layers.batch\_norm(tf.matmul(fc1,w1), is\_training=is\_train, epsilon=1e-5, decay = 0.9, updates\_collections=None, scope='bnf')

# act\_fc1 = lrelu(tf.nn.bias\_add(bnf, b1),n = 'actf')

w2 = tf.get\_variable('w2', shape=[fc1.shape[-1], 1], dtype=tf.float32,

initializer=tf.truncated\_normal\_initializer(stddev=0.02))

b2 = tf.get\_variable('b2', shape=[1], dtype=tf.float32,

initializer=tf.constant\_initializer(0.0))

# wgan just get rid of the sigmoid

logits = tf.add(tf.matmul(fc1, w2), b2, name='logits')

# dcgan

acted\_out = tf.nn.sigmoid(logits)

return logits #, acted\_out

def train():

random\_dim = 100

print os.environ['CUDA\_VISIBLE\_DEVICES']

with tf.variable\_scope('input'):

real\_image = tf.placeholder(tf.float32, shape = [None, HEIGHT, WIDTH, CHANNEL], name='real\_image')

random\_input = tf.placeholder(tf.float32, shape=[None, random\_dim], name='rand\_input')

is\_train = tf.placeholder(tf.bool, name='is\_train')

# wgan

fake\_image = generator(random\_input, random\_dim, is\_train)

real\_result = discriminator(real\_image, is\_train)

fake\_result = discriminator(fake\_image, is\_train, reuse=True)

d\_loss = tf.reduce\_mean(fake\_result) - tf.reduce\_mean(real\_result) # This optimizes the discriminator.

g\_loss = -tf.reduce\_mean(fake\_result) # This optimizes the generator.

# # dcgan loss

# fake\_image = generator(random\_input, random\_dim, is\_train)

# # sample\_fake = generator(random\_input, random\_dim, is\_train, reuse = True)

# real\_logits, real\_result = discriminator(real\_image, is\_train)

# fake\_logits, fake\_result = discriminator(fake\_image, is\_train, reuse=True)

# d\_loss1 = tf.reduce\_mean(

# tf.nn.sigmoid\_cross\_entropy\_with\_logits(

# logits = real\_logits, labels = tf.ones\_like(real\_logits)))

# d\_loss2 = tf.reduce\_mean(

# tf.nn.sigmoid\_cross\_entropy\_with\_logits(

# logits = fake\_logits, labels = tf.zeros\_like(fake\_logits)))

# d\_loss = d\_loss1 + d\_loss2

# g\_loss = tf.reduce\_mean(

# tf.nn.sigmoid\_cross\_entropy\_with\_logits(

# logits = fake\_logits, labels = tf.ones\_like(fake\_logits)))

t\_vars = tf.trainable\_variables()

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

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

# test

# print(d\_vars)

trainer\_d = tf.train.RMSPropOptimizer(learning\_rate=2e-4).minimize(d\_loss, var\_list=d\_vars)

trainer\_g = tf.train.RMSPropOptimizer(learning\_rate=2e-4).minimize(g\_loss, var\_list=g\_vars)

# clip discriminator weights

d\_clip = [v.assign(tf.clip\_by\_value(v, -0.01, 0.01)) for v in d\_vars]

batch\_size = BATCH\_SIZE

image\_batch, samples\_num = process\_data()

batch\_num = int(samples\_num / batch\_size)

total\_batch = 0

sess = tf.Session()

saver = tf.train.Saver()

sess.run(tf.global\_variables\_initializer())

sess.run(tf.local\_variables\_initializer())

# continue training

ckpt = tf.train.latest\_checkpoint('./model/' + version)

saver.restore(sess, ckpt)

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)

print 'total training sample num:%d' % samples\_num

print 'batch size: %d, batch num per epoch: %d, epoch num: %d' % (batch\_size, batch\_num, EPOCH)

print 'start training...'

for i in range(EPOCH):

for j in range(batch\_num):

d\_iters = 5

g\_iters = 1

train\_noise = np.random.uniform(-1.0, 1.0, size=[batch\_size, random\_dim]).astype(np.float32)

for k in range(d\_iters):

train\_image = sess.run(image\_batch)

#wgan clip weights

sess.run(d\_clip)

# Update the discriminator

\_, dLoss = sess.run([trainer\_d, d\_loss],

feed\_dict={random\_input: train\_noise, real\_image: train\_image, is\_train: True})

# Update the generator

for k in range(g\_iters):

# train\_noise = np.random.uniform(-1.0, 1.0, size=[batch\_size, random\_dim]).astype(np.float32)

\_, gLoss = sess.run([trainer\_g, g\_loss],

feed\_dict={random\_input: train\_noise, is\_train: True})

# print 'train:[%d/%d],d\_loss:%f,g\_loss:%f' % (i, j, dLoss, gLoss)

# save check point every 500 epoch

if i%500 == 0:

if not os.path.exists('./model/' + version):

os.makedirs('./model/' + version)

saver.save(sess, './model/' +version + '/' + str(i))

if i%50 == 0:

# save images

if not os.path.exists(newPoke\_path):

os.makedirs(newPoke\_path)

sample\_noise = np.random.uniform(-1.0, 1.0, size=[batch\_size, random\_dim]).astype(np.float32)

imgtest = sess.run(fake\_image, feed\_dict={random\_input: sample\_noise, is\_train: False})

# imgtest = imgtest \* 255.0

# imgtest.astype(np.uint8)

save\_images(imgtest, [8,8] ,newPoke\_path + '/epoch' + str(i) + '.jpg')

print 'train:[%d],d\_loss:%f,g\_loss:%f' % (i, dLoss, gLoss)

coord.request\_stop()

coord.join(threads)

# def test():

# random\_dim = 100

# with tf.variable\_scope('input'):

# real\_image = tf.placeholder(tf.float32, shape = [None, HEIGHT, WIDTH, CHANNEL], name='real\_image')

# random\_input = tf.placeholder(tf.float32, shape=[None, random\_dim], name='rand\_input')

# is\_train = tf.placeholder(tf.bool, name='is\_train')

# # wgan

# fake\_image = generator(random\_input, random\_dim, is\_train)

# real\_result = discriminator(real\_image, is\_train)

# fake\_result = discriminator(fake\_image, is\_train, reuse=True)

# sess = tf.InteractiveSession()

# sess.run(tf.global\_variables\_initializer())

# variables\_to\_restore = slim.get\_variables\_to\_restore(include=['gen'])

# print(variables\_to\_restore)

# saver = tf.train.Saver(variables\_to\_restore)

# ckpt = tf.train.latest\_checkpoint('./model/' + version)

# saver.restore(sess, ckpt)

if \_\_name\_\_ == "\_\_main\_\_":

train()

# test()