VE445 INTRO. TO MACHINE LEARNING

REPORT OF Lab 3

GAN

Name and ID

Sijun Zhang 516370910155

Professor: Bo Yuan

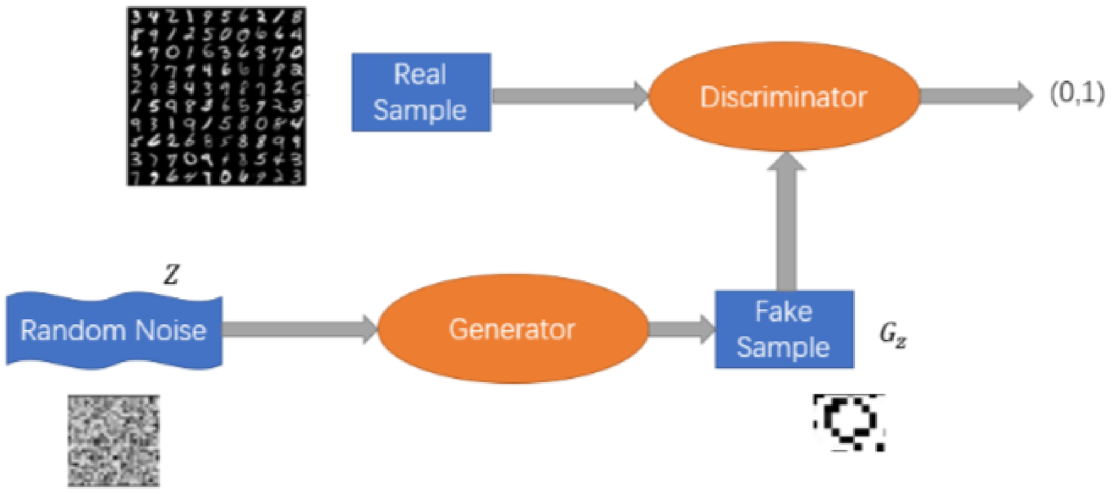
May 1, 2019

1 Introduction

In the last lab, we have implemented the generative model, Generative Adversarial Networks. Put up by Ian J. Goodfellow in 2014, Generative Adversarial Networks have been wildly used in image generation and semantic segmentation. In lab 2, we have implemented one of the most efficient and popular network model and learned the simple idea behind big data and complex network structures.

1.1 Generative Adversarial Network

Compared with other networks that contain single topological structure, GAN has two individual network structures: discriminant model and generative network. The function of discriminant network D, or discriminator, is to identify whether a given sample is a true sample or a fake one produced by generative model G. While the function for generator G is to generate samples by random noise. In the training procedure, the generator should deceive the discriminator as much as possible by generating fake samples. The abstract process of GAN training is shown below:



The output of the discriminator is a value between (0,1), which represents for the probability whether the given sample is real. The random noise is an random vector served as arbitrary input for generator. The loss function should be calculated individually for both the discriminator and the generator.

$$\min L_D = \min_D - \sum_{i=1}^{m_r} \log D(\mathbf{x}_i) - \sum_{j=1}^{m_f} \log D(G(\mathbf{z}))$$

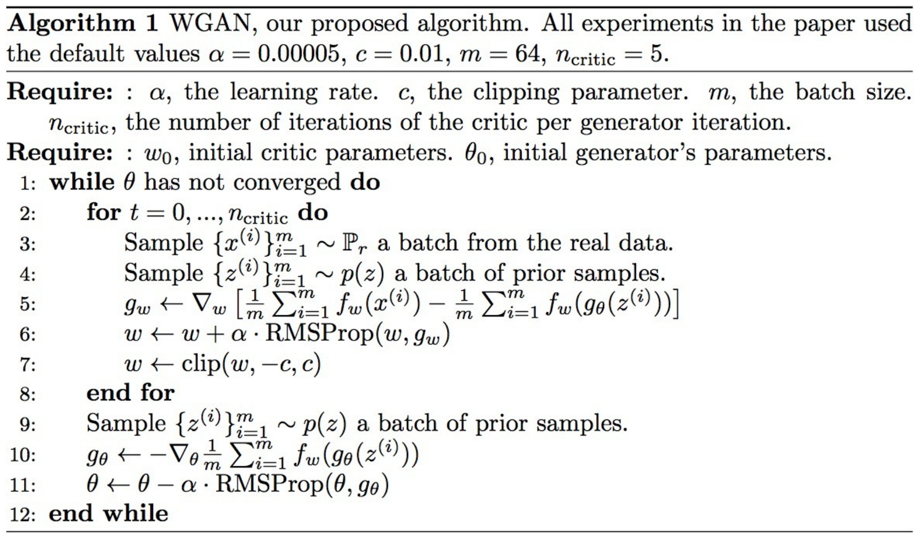
$$\min L_G = \min_G - \sum_{i=1}^{m_f} \log D(G(\mathbf{z}))$$

1.2 Wasserstein GAN

Put up by Ian J. Goodfellow in 2014, GAN is always associated with the Problems such as the training difficulty, the loss of Generator and Discriminator can not guide the training process and the paucity of variation in generated samples. One of the famous enhancement trials is DCGAN, which is dependent on exhausting the possible structure of Discriminator and Generator to find the best one, but it still negelected the fundamental problems in GAN. Wassertein GAN is created in 2017, and successfully the following problems:

- Successfully handled the stability of Generator and Discriminator.
- Avoided collapse mode and ensured the variety of generated samples.
- Found a universal loss to indicate training process completeness.
- Added the robustness to the network structure.

And the algorithm is shown in the following chart.



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

```
Require: : $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch size.
    $n_{\text{critic}}$, the number of iterations of the critic per generator iteration.
Require: : $w_0$, initial critic parameters. $\theta_0$, initial generator's parameters.
 1: while $\theta$ has not converged do
 2:     for $t = 0, ..., n_{\text{critic}}$ do
 3:         Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$ a batch from the real data.
 4:         Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.
 5:         $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^m f_w(g_\theta(z^{(i)}))\right]$
 6:         $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
 7:         $w \leftarrow \text{clip}(w, -c, c)$
 8:     end for
 9:     Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.
10:     $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^m f_w(g_\theta(z^{(i)}))$
11:     $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: end while
```

The WGAN also introduced the W-distance to stimulate the global loss.

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

$$\max_{w\in\mathcal{W}} \mathbb{E}_{x\sim\mathbb{P}_r}[f_w(x)] - \mathbb{E}_{z\sim p(z)}[f_w(g_\theta(z)]$$

Compared to Jensen-Shannon divergence, W-distance is well-posed and smooth.

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r \| \mathbb{P}_m) + KL(\mathbb{P}_g \| \mathbb{P}_m)$$

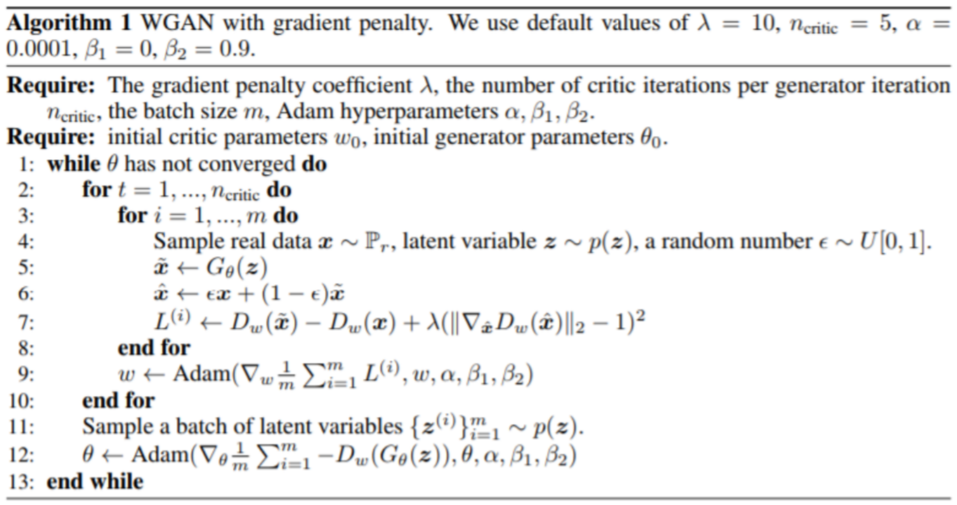
where $\mathbb{P}_m$ is the mixture $(\mathbb{P}_r + \mathbb{P}_g)/2$. This divergence is symmetrical and always defined because we can choose $\mu = \mathbb{P}_m$.

1.3 Wasserstein GAN with gradient penalty

We find WGAN has a step to clip the weight into the interval of (-C, C), which is not so meaningful, beacuse:

- The weight clipping take a global effect, but it only has an indirect effect on the norm of gradient, which has the risk in gradient vanishing and explosion.
- Gradient Penalty only influence the centre region where samples cluster and the middle region, but it restricts the norm of gradient and easy to adjust to the reasonable size.

so the WGAN-GP has the following algorithm:



Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

```
Require: The gradient penalty coefficient $\lambda$, the number of critic iterations per generator iteration $n_{\text{critic}}$, the batch size $m$, Adam hyperparameters $\alpha, \beta_1, \beta_2$.
Require: initial critic parameters $w_0$, initial generator parameters $\theta_0$.
1:  while $\theta$ has not converged do
2:      for $t = 1, ..., n_{\text{critic}}$ do
3:          for $i = 1, ..., m$ do
4:              Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
5:              $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
6:              $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
7:              $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
8:          end for
9:          $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
10:     end for
11:     Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
12:     $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
13: end while
```

2 Result

2.1 Generated Samples

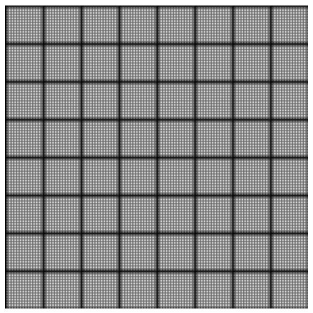


figure 1. Epoch 1

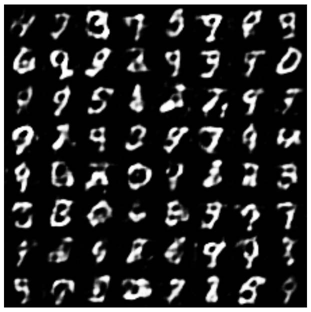


figure 2. Epoch 6

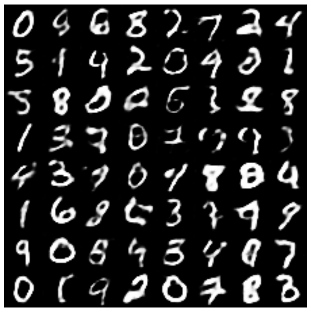


figure 3. Epoch 11

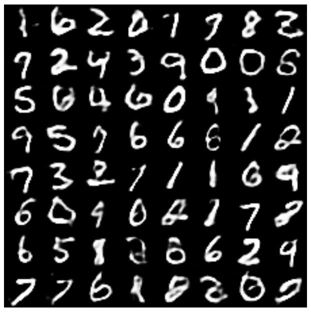


figure 4. Epoch 16

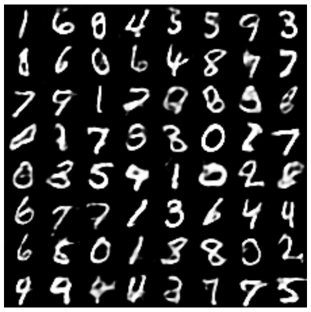


figure 5. Epoch 21

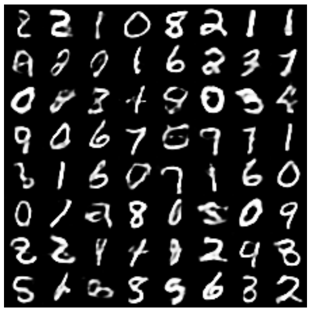


figure 6. Epoch 26

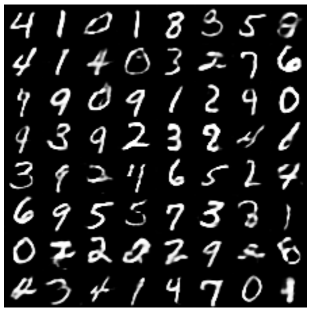


figure 7. Epoch 31

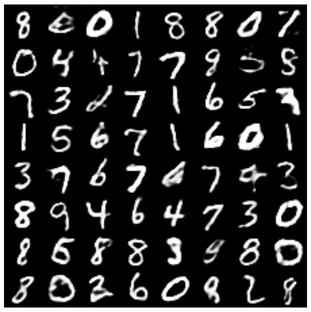


figure 8. Epoch 36

Each Epoch including 1000 iters, which has the batch size of 64

We can find the image is becoming clear and sharp. The contrast ratio is strengthened after the tenth epoch. The noise point is dramatically decreased along with the increasement of characters' luminance.

2.2 Accuracy and Loss

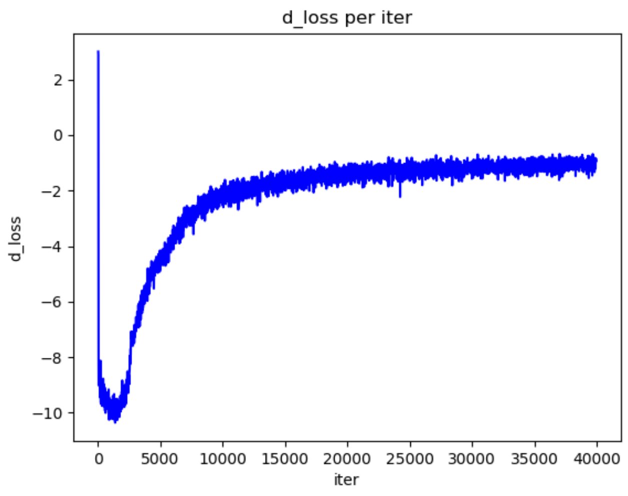


figure 9. The Discriminator loss of real and generated data

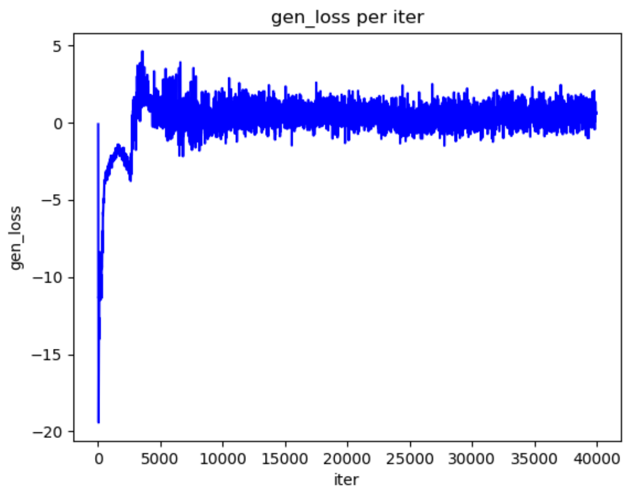


figure 10. The Generator loss of generated data in discrimination

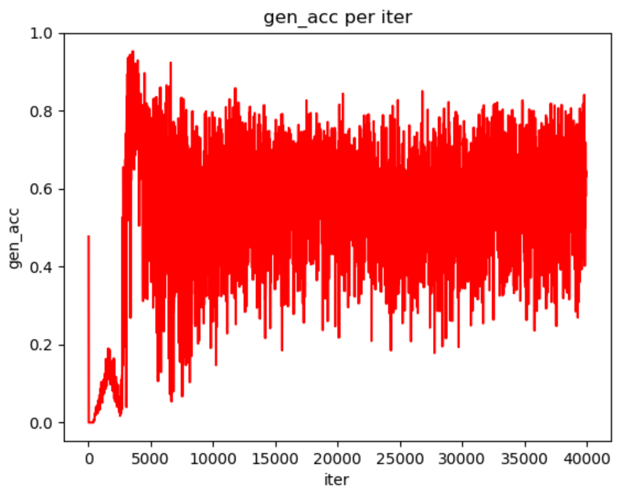


figure 11. The identification accuracy of generated data in discrimination obtained by sigmoid of g_loss

As the WGAN-GP eliminate the sigmoid function in the output layer of discriminator, the generator loss is much more unstable than the normal GAN in the training process, although it still share an expectancy of 0 and correspondingly, the sigmoid value as the accuracy has the expectancy of 0.5. The approaching process is of great variance but has the approach to the expectancy. We can also learn that the g_loss and d_loss are only the local loss of the whole problem.

3 Appendix

3.1 mnist_model.py

```
#coding:utf-8
import os
import numpy as np
import scipy.misc
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data #as mnist_data
import matplotlib.pyplot as plt
from tensorflow.python.framework import ops
ops.reset_default_graph()

class MnistModel:

    def __init__(self):
        self.DEPTH = 28
        self.OUTPUT_SIZE = 28
        self.BATCH_SIZE = 64
        self.LAMBDA = 10
        self.EPOCH = 41
        self.LEARNING_RATE = 0.01
        self.BETA1 = 0.5
        self.BETA2 = 0.9

    def lrelu(self,name,x, leak=0.2):
        return tf.maximum(x, leak * x, name=name)

    def Discriminator(self,name,inputs,reuse):
        with tf.variable_scope(name, reuse=reuse):
            o = tf.reshape(inputs, [-1, 28, 28, 1])
            with tf.variable_scope('d_conv_1'):
                w1 = tf.get_variable('w1', [5, 5, o.get_shape()[-1],self.DEPTH], dtype=tf.float32, i
                var1 = tf.nn.conv2d(o,w1,[1,2, 2,1],padding='SAME')
                b1 = tf.get_variable('b1', [self.DEPTH], 'float32',initializer=
                tf.constant_initializer(0.01))
                o1 = tf.nn.bias_add(var1, b1)

            o2 = self.lrelu('d_lrelu_1', o1)
            with tf.variable_scope('d_conv_2'):
                w2 = tf.get_variable('w2', [5, 5, o2.get_shape()[-1],2*self.DEPTH], dtype=tf.float3
                var2 = tf.nn.conv2d(o2,w2,[1,2, 2,1],padding='SAME')
                b2 = tf.get_variable('b2', [2*self.DEPTH], 'float32',initializer=
                tf.constant_initializer(0.01))
                o3 = tf.nn.bias_add(var2, b2)

            o4 = self.lrelu('d_lrelu_2', o3)

            with tf.variable_scope('d_conv_3'):
                w3 = tf.get_variable('w3', [5, 5, o4.get_shape()[-1],4*self.DEPTH], dtype=tf.float3
                var3 = tf.nn.conv2d(o4,w3,[1,2, 2,1],padding='SAME')
                b3 = tf.get_variable('b3', [4*self.DEPTH], 'float32',initializer=
                tf.constant_initializer(0.01))
                o5 = tf.nn.bias_add(var3, b3)

            o6 = self.lrelu('d_lrelu_3', o5)
            chanel = o6.get_shape().as_list()
            o7 = tf.reshape(o6, [self.BATCH_SIZE, chanel[1]*chanel[2]*chanel[3]])
            o8 = self.dense('d_fc', o7, 1)
            return o8

    def Generator(self,name, reuse=False):
        with tf.variable_scope(name, reuse=reuse):
            noise = tf.random_normal([self.BATCH_SIZE, 128])
            noise = tf.reshape(noise, [self.BATCH_SIZE, 128], 'noise')
        output = self.dense('g_fc_1', noise, 2*2*8*self.DEPTH)
        output = tf.reshape(output, [self.BATCH_SIZE, 2, 2, 8*self.DEPTH], 'g_conv')

        output = self.deconv2d('g_deconv_1', output, ksize=5, outshape=[self.BATCH_SIZE,
        4, 4, 4*self.DEPTH])
        output = tf.nn.relu(output)
        output = tf.reshape(output, [self.BATCH_SIZE, 4, 4, 4*self.DEPTH])

        output = self.deconv2d('g_deconv_2', output, ksize=5, outshape=[self.BATCH_SIZE,
        7, 7, 2* self.DEPTH])
        output = tf.nn.relu(output)

        output = self.deconv2d('g_deconv_3', output, ksize=5, outshape=[self.BATCH_SIZE,
        14, 14, self.DEPTH])
        output = tf.nn.relu(output)

        output = self.deconv2d('g_deconv_4', output, ksize=5, outshape=[self.BATCH_SIZE,
        self.OUTPUT_SIZE, self.OUTPUT_SIZE, 1])
        output = tf.nn.sigmoid(output)
        return tf.reshape(output,[-1,784])


def deconv2d(self,name, tensor, ksize, outshape, stddev=0.01, stride=2, padding='SAME'):
    with tf.variable_scope(name):
        w = tf.get_variable('w', [ksize, ksize, outshape[-1], tensor.get_shape()[-1]],
        dtype=tf.float32,
                            initializer=tf.random_normal_initializer(stddev=stddev))
        var = tf.nn.conv2d_transpose(tensor, w, outshape, strides=[1, stride, stride, 1],
        padding=padding)
        b = tf.get_variable('b', [outshape[-1]], 'float32',
        initializer=tf.constant_initializer(0.01))
        return tf.nn.bias_add(var, b)

def save_images(self, images, size, path):
    img = (images + 1.0) / 2.0
    h, w = img.shape[1], img.shape[2]
    merge_img = np.zeros((h * size[0], w * size[1], 3))
    for idx, image in enumerate(images):
        i = idx % size[1]
        j = idx // size[1]
        merge_img[j * h:j * h + h, i * w:i * w + w, :] = image
    return scipy.misc.imsave(path, merge_img)

def dense(self, name,value, o_shape):
    with tf.variable_scope(name, reuse=None) as scope:
        shape = value.get_shape().as_list()
        w = tf.get_variable('w', [shape[1], o_shape], dtype=tf.float32, initializer=tf.random_no
        b = tf.get_variable('b', [o_shape], dtype=tf.float32,
        initializer=tf.constant_initializer(0.0))
        return tf.matmul(value, w) + b

def acc_cnt(self, inp, size):
    cnt = 0
    for i in inp:
        if (i > 0.5):
            cnt += 1
    acc = float(cnt) / float(size)
    return acc

def train(self):
    with tf.variable_scope(tf.get_variable_scope()):
        real_data = tf.placeholder(tf.float32, shape=[self.BATCH_SIZE,784])

        with tf.variable_scope(tf.get_variable_scope()):
            fake_data = self.Generator('gen',reuse=False)
            disc_real = self.Discriminator('dis_r',real_data,reuse=False)
            disc_fake = self.Discriminator('dis_r',fake_data,reuse=True)

        t_vars = tf.trainable_variables()
        d_vars = [var for var in t_vars if 'd_' in var.name]
        g_vars = [var for var in t_vars if 'g_' in var.name]

        gen_cost = tf.reduce_mean(disc_fake)
        disc_cost = -tf.reduce_mean(disc_fake) + tf.reduce_mean(disc_real)
        real_in_box = tf.reduce_mean(tf.sigmoid(disc_real) )
        fake_in_box = tf.reduce_mean(tf.sigmoid(disc_fake))

        alpha = tf.random_uniform(shape=[self.BATCH_SIZE, 1],minval=0.,maxval=1.)
        differences = fake_data - real_data
        interpolates = real_data + (alpha * differences)
        gradients = tf.gradients(self.Discriminator('dis_r',interpolates,reuse=True),
        [interpolates])[0]
        slopes = tf.sqrt(tf.reduce_sum(tf.square(gradients), reduction_indices=[1]))
        gradient_penalty = tf.reduce_mean((slopes - 1.) ** 2)
        disc_cost += self.LAMBDA * gradient_penalty

        with tf.variable_scope(tf.get_variable_scope(), reuse=None):
            gen_train_op = tf.train.GradientDescentOptimizer(learning_rate=
            self.LEARNING_RATE)
            .minimize(gen_cost,var_list=g_vars)
            disc_train_op = tf.train.GradientDescentOptimizer(learning_rate=
            self.LEARNING_RATE)
    .minimize(disc_cost,var_list=d_vars)

saver = tf.train.Saver()

# sess = tf.Session()
sess = tf.InteractiveSession()
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
if not os.path.exists('img'):
    os.mkdir('img')

init = tf.global_variables_initializer()
sess.run(init)
mnist = input_data.read_data_sets("data", one_hot=True)
cnt = 0
step_jump_col = []
g_loss_col = []
d_loss_col = []
gen_acc_col = []
for EPOCH in range (1,self.EPOCH):
    idxs = 1000
    for iters in range(1, idxs):
        img, _ = mnist.train.next_batch(self.BATCH_SIZE)

        for x in range (0,5):
            _, d_loss = sess.run([disc_train_op, disc_cost],
            feed_dict={real_data: img})
        _, g_loss = sess.run([gen_train_op, gen_cost])

        if (iters % 10 == 0):
            step_jump_col.append(cnt)
            g_loss_col.append(g_loss)
            d_loss_col.append(d_loss)
            real_confidence = sess.run(real_in_box, feed_dict = {real_data:img})
            fake_confidence = sess.run(fake_in_box)
            gen_acc_col.append(fake_confidence)
            print("( EPOCH:" + str(EPOCH) + "  iters: " + str(iters) +
            ") d_loss: " + str(d_loss) + ", g_loss: " +
            str(g_loss) +  ", gen_acc: " + str(fake_confidence))
        cnt +=1

    with tf.variable_scope(tf.get_variable_scope()):
        samples = self.Generator('gen', reuse=True)
        samples = tf.reshape(samples, shape=[self.BATCH_SIZE, 28,28,1])
        samples=sess.run(samples)
        self.save_images(samples, [8,8], os.getcwd()+'/img/'+'sample_
                    %d_epoch.png' % (EPOCH))

                if EPOCH>=(self.EPOCH - 1):
                    checkpoint_path = os.path.join(os.getcwd(), 'generator_mnist')
                    saver.save(sess, checkpoint_path)

            # coord.request_stop()
            # coord.join(threads)

            p1 = './fig/d_loss'
            p2 = './fig/g_loss'
            p3 = './fig/gen_acc'
            plt.plot(step_jump_col, d_loss_col, 'b-')
            plt.title('d_loss per iter')
            plt.xlabel('iter')
            plt.ylabel('d_loss')
            plt.savefig(p1)
            plt.show()

            plt.plot(step_jump_col, g_loss_col, 'b-')
            plt.title('gen_loss per iter')
            plt.xlabel('iter')
            plt.ylabel('gen_loss')
            plt.savefig(p2)
            plt.show()



            plt.plot(step_jump_col, gen_acc_col, 'r-')
            plt.title('gen_acc per iter')
            plt.xlabel('iter')
            plt.ylabel('gen_acc')
            plt.savefig(p3)
            plt.show()

            coord.request_stop()
            coord.join(threads)
            sess.close()
if __name__ == '__main__':
    mnist = MnistModel()
    mnist.train()
```