Realization of GAN/DCGAN on MNIST Dataset with TensorFlow(CPU)

TensorFlow相关知识

tf.compat.v1.兼容模块自2020年起不再维护，请使用TensorFlow 2.0或者import torch as tf，哈哈哈哈哈！！！

TensorFlow两个函数

tf.nn.conv2d()函数

```
def conv2d(input, filter, strides, padding, data_format='NHWC', dilations=None, name=None)
```

- input: [batch_size, in_height, in_width, n_channels]，表示图片的批数，大小和通道
- filter: [filter_height, filter_width, in_channels, out_channels]，表示kernel的大小，in_channels应当和input的n_channels一致，out_channels可以随意指定，体现了卷积核的数量，有关channel的理解具体参考【CNN】理解卷积神经网络中的通道 channel
- strides: [1, height_stride, width_stride, 1]，大部分情况，height_stride = width_stride
- padding: 'SAME'或者'VALID'，表示在边缘处的处理方法，即是否需要填充，有关padding的理解具体参考TensorFlow中CNN的两种padding方式“SAME”和“VALID”
- 返回一个tensor，类型不变，shape仍然是[batch_size, height, width, channels]这种形式

tf.nn.max_pool()函数

```
def tf.nn.max_pool(input, ksize, strides, padding, data_format=None, name=None)
```

- input: [batch_size, in_height, in_width, n_channels]，表示特征图的批数，大小和通道
- ksize: [1, kernel_height, kernel_width, 1]，表示池化窗口的大小

- strides: 和卷积类似，窗口在每一个维度上滑动的步长，一般即[1, height_stride, width_stride, 1]
- padding: 和卷积类似，可以取'VALID'或者'SAME'
- 返回一个tensor，类型不变，shape仍然是[batch_size, height, width, channels]这种形式

TensorFlow的作用域

命名域，有两种作用，一是类似C++命名空间的作用；二是用于TensorBorad绘图时的模块名称

```
def tf.name_scope(name)
```

变量域，常与tf.get_variable()一块使用，当reuse=False时，tf.get_variable()创建新共享变量；当reuse=True时，tf.get_variable()重用共享变量，具体参考共享变量｜TensorFlow官方文档中文版

```
def tf.variable_scope(
    name_or_scope,
    default_name=None,
    values=None,
    initializer=None,
    regularizer=None,
    caching_device=None,
    partitioner=None,
    custom_getter=None,
    reuse=None,
    dtype=None,
    use_resource=None,
    constraint=None,
    auxiliary_name_scope=True
)
```

GAN/DCGAN搭建流程

以下各部分拼起来就是完整的代码，按照已设的超参，在CPU上大概要跑一节课的时间，参考资料生成对抗网络（GAN）之MNIST数据生成

MNIST Input

以下导入方法即将弃用，推荐导入方法具体参考Warning: Please use alternatives such as official/mnist/dataset.py from tensorflow/models

```
import tensorflow.compat.v1 as tf
import numpy as np
import pickle
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # 只显示Errors
mnist = input_data.read_data_sets('mnist_data/', one_hot=True)
```

Hyperparameters

```
# 以下列出了所有超参
alpha = 0.01  # leaky ReLU函数的系数
rate = 0.2  # 训练时随机拿掉的神经元所占的比例，防止过拟合
learning_rate = 0.001  # the learning rate
beta1 = 0.4  # 一阶矩估计的指数衰减率，这个参数对DCGAN影响贼大
smooth = 0.1  # label smoothing，目标更加soft，可以防止过拟合
batch_size = 64  # the batch size
epochs = 300  # the number of epochs, maybe 300 for GAN, >=2 for DCGAN
n_sample = 25  # the sample size
```

Generator

Ian Goodfellow论文中的GAN的生成器，使用了两个全连接层

```python
def generator(z, reuse=False):
    # reuse: 仅用于后面的抽取样本进行观察，不观察就不需要reuse的
    # 生成器内部的超参
    alpha = 0.01  # leaky ReLU函数的系数
    rate = 0.2  # 训练时随机拿掉的神经元所占的比例，防止过拟合
    with tf.variable_scope('generator', reuse=reuse):
        with tf.variable_scope('fc_layer', reuse=reuse):
            # 全连接层，含128个神经元
            h_fc1 = tf.nn.leaky_relu(tf.layers.dense(z, 128), alpha=alpha)
            h_fc1_drop = tf.nn.dropout(h_fc1, rate=rate)
        with tf.variable_scope('output', reuse=reuse):
            # 输出层（由全连接层实现），这里经过测试，tanh激活函数要比sigmoid或者relu激活函数的效果好
            y_fake = tf.nn.tanh(tf.layers.dense(h_fc1_drop, 784))
    return y_fake
```

Alec Radford论文中的DCGAN的生成器，使用了三个卷积层和一个全连接层（简易版本）

```python
def generator(z, reuse=False):
    # reuse：仅用于后面的抽取样本进行观察，不观察就不需要reuse的
    # 生成器内部的超参
    alpha = 0.01  # leaky ReLU函数的系数
    rate = 0.2  # 训练时随机拿掉的神经元所占的比例，防止过拟合
    with tf.variable_scope('generator', reuse=reuse):
        with tf.variable_scope('fc_layer', reuse=reuse):
            # 全连接层，100 x 1 to 4*4*512 x 1 to to 4 x 4 x 512
            h_fc1 = tf.reshape(tf.layers.dense(z, 4*4*512), [-1, 4, 4, 512])
            # batch normalization，1.有助于快速收敛，2.防止因层数过多而导致的梯度消失，3.稳定每一层的数据
            h_fc1_bn = tf.layers.batch_normalization(h_fc1, training=not reuse)
            h_fc1_lr = tf.nn.leaky_relu(h_fc1_bn, alpha=alpha)
            h_fc1_drop = tf.nn.dropout(h_fc1_lr, rate=rate)
        with tf.variable_scope('cnn_layer1', reuse=reuse):
            # 第一层卷积，4 x 4 x 512 to 7 x 7 x 256
            h_conv1 = tf.layers.conv2d_transpose(h_fc1_drop, 256, 4, strides=1, padding='VALID')
            h_conv1_bn = tf.layers.batch_normalization(h_conv1, training=not reuse)
            h_conv1_lr = tf.nn.leaky_relu(h_conv1_bn, alpha=alpha)
            h_conv1_drop = tf.nn.dropout(h_conv1_lr, rate=rate)
        with tf.variable_scope('cnn_layer2', reuse=reuse):
            # 第二层卷积，7 x 7 x 256 to 14 x 14 x 128
            h_conv2 = tf.layers.conv2d_transpose(h_conv1_drop, 128, 3, strides=2, padding='SAME')
            h_conv2_bn = tf.layers.batch_normalization(h_conv2, training=not reuse)
            h_conv2_lr = tf.nn.leaky_relu(h_conv2_bn, alpha=alpha)
            h_conv2_drop = tf.nn.dropout(h_conv2_lr, rate=rate)
        with tf.variable_scope('output', reuse=reuse):
            # 输出层（由卷积层实现），14 x 14 x 128 to 28 x 28 x 1 to 784 x 1
            h_conv3 = tf.layers.conv2d_transpose(h_conv2_drop, 1, 3, strides=2, padding='SAME')
            # 卷积层后如果有batch normalization，则不需要bias，而这里需要
            b_conv3 = tf.get_variable('biases', [1], initializer=tf.zeros_initializer())
            y_fake = tf.nn.tanh(tf.reshape(h_conv3 + b_conv3, [-1, 784]))
    return y_fake
```

Discriminator

Ian Goodfellow论文中的GAN的判别器，使用了两个全连接层

```
def discriminator(x, reuse=False):
    # reuse: 因为真实图像与生成图像是共享判别器的参数的
    # 判别器内部的超参
    alpha = 0.01  # leaky ReLU函数的系数
    rate = 0.2  # 训练时随机拿掉的神经元所占的比例，防止过拟合
    with tf.variable_scope('discriminator', reuse=reuse):
        with tf.variable_scope('fc_layer', reuse=reuse):
            # 全连接层，含128个神经元
            h_fc1 = tf.nn.leaky_relu(tf.layers.dense(x, 128), alpha=alpha)
            h_fc1_drop = tf.nn.dropout(h_fc1, rate=rate)
        with tf.variable_scope('output', reuse=reuse):
            # 输出层（由全连接层实现），y_logits用于tf.nn.sigmoid_cross_entropy_with_logits()以构建损失函数
            y_logits = tf.layers.dense(h_fc1_drop, 1)
            y_prob = tf.nn.sigmoid(y_logits)
    return y_logits, y_prob
```

Alec Radford论文中的DCGAN的判别器，使用了三个卷积层和一个全连接层（简易版本）

```python
def discriminator(x, reuse=False):
    # reuse: 因为真实图像与生成图像是共享判别器的参数的
    # 判别器内部的超参
    alpha = 0.01  # leaky ReLU函数的系数
    rate = 0.2  # 训练时随机拿掉的神经元所占的比例，防止过拟合
    with tf.variable_scope('discriminator', reuse=reuse):
        with tf.variable_scope('cnn_layer1', reuse=reuse):
            # 第一层卷积，784 x 1 to 28 x 28 x 1 to 14 x 14 x 128
            h_conv1 = tf.layers.conv2d(tf.reshape(x, [-1, 28, 28, 1]), 128, 3, strides=2, padding='SAME')
            # 这里training=True是因为判别器只用在了训练过程中，不同于生成器
            h_conv1_bn = tf.layers.batch_normalization(h_conv1, training=True)
            h_conv1_lr = tf.nn.leaky_relu(h_conv1_bn, alpha=alpha)
            h_conv1_drop = tf.nn.dropout(h_conv1_lr, rate=rate)
        with tf.variable_scope('cnn_layer2', reuse=reuse):
            # 第二层卷积，14 x 14 x 128 to 7 x 7 x 256
            h_conv2 = tf.layers.conv2d(h_conv1_drop, 256, 3, strides=2, padding='SAME')
            h_conv2_bn = tf.layers.batch_normalization(h_conv2, training=True)
            h_conv2_lr = tf.nn.leaky_relu(h_conv2_bn, alpha=alpha)
            h_conv2_drop = tf.nn.dropout(h_conv2_lr, rate=rate)
        with tf.variable_scope('cnn_layer3', reuse=reuse):
            # 第三层卷积，7 x 7 x 256 to 4 x 4 x 512
            h_conv3 = tf.layers.conv2d(h_conv2_drop, 512, 4, strides=1, padding='VALID')
            h_conv3_bn = tf.layers.batch_normalization(h_conv3, training=True)
            h_conv3_lr = tf.nn.leaky_relu(h_conv3_bn, alpha=alpha)
            h_conv3_drop = tf.nn.dropout(h_conv3_lr, rate=rate)
        with tf.variable_scope('output', reuse=reuse):
            # 输出层（由全连接层实现），4 x 4 x 512 to 4*4*512 x 1
            h_fc1 = tf.reshape(h_conv3_drop, [-1, 4*4*512])
            # y_logits用于tf.nn.sigmoid_cross_entropy_with_logits()以构建损失函数
            y_logits = tf.layers.dense(h_fc1, 1)
            y_prob = tf.nn.sigmoid(y_logits)
    return y_logits, y_prob
```

Loss & Optimizer

Ian Goodfellow论文中的loss如下，本实现并没有使用该loss，而是使用交叉熵（Cross entropy），当然改进为WGAN-GP类似的loss最好，有关各种损失函数的理解具体参考【深度学习】一文读懂机器学习常用损失函数（Loss Function）

```
d_loss = -tf.reduce_mean(tf.log(d_real) + tf.log(1. - d_fake))
g_loss = -tf.reduce_mean(tf.log(d_fake))
```

```
def loss_and_optimizer(x_logits_real, x_logits_fake, z_vars, x_vars):
    # 损失函数和优化器的超参
    learning_rate = 0.001  # the learning rate
    beta1 = 0.4  # 一阶矩估计的指数衰减率，这个参数对DCGAN影响贼大
    smooth = 0.1  # label smoothing，目标更加soft，可以防止过拟合
    with tf.name_scope('loss'):
        # tf.nn.sigmoid_cross_entropy_with_logits()函数内部会对预测输入执行Sigmoid函数
        # 对于给定的真实图像，判别器要为其打上标签1
        # 对于给定的生成图像，判别器要为其打上标签0
        x_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
            logits=x_logits_real, labels=tf.ones_like(x_logits_real)) * (1 - smooth))
        x_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
            logits=x_logits_fake, labels=tf.zeros_like(x_logits_fake)))
        # discriminator的loss
        x_loss = tf.add(x_loss_real, x_loss_fake)
        # generator的loss
        # 对于生成器传给判别器的生成图像，生成器希望判别器打上标签1
        z_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
            logits=x_logits_fake, labels=tf.ones_like(x_logits_fake)) * (1 - smooth))
    with tf.name_scope('optimizer'):
        z_train_opt = tf.train.AdamOptimizer(learning_rate, beta1).minimize(z_loss, var_list=z_vars)
        x_train_opt = tf.train.AdamOptimizer(learning_rate, beta1).minimize(x_loss, var_list=x_vars)
    return z_loss, x_loss, z_train_opt, x_train_opt
```

Building Model

```
with tf.name_scope('input'):
    x_real = tf.placeholder(tf.float32, [None, 784])
    z_noise = tf.placeholder(tf.float32, [None, 100])  # 输入噪声为100维度的向量组
with tf.name_scope('generator'):
    x_fake = generator(z_noise)
with tf.name_scope('discriminator'):
    # 分别输入真实图像和生成图像，并投入判别器以判断真伪
    d_logits_real, d_real = discriminator(x_real)
    d_logits_fake, d_fake = discriminator(x_fake, reuse=True)
```

Training Model

训练trick：训练m次生成器，训练n次判别器，然后交替。更多GAN的训练技巧具体参考How to Train a GAN? Tips and tricks to make GANs work

```
with tf.name_scope('train'):
    # 训练模型时的超参
    batch_size = 64  # the batch size
    epochs = 300  # the number of epochs, maybe 300 for GAN, >=2 for DCGAN
    n_sample = 25  # the sample size

    # 生成器网络和判别器网络是两个网络，所以各自的optimizer优化的变量不同
    train_vars = tf.trainable_variables()
    g_vars = [var for var in train_vars if var.name.startswith('generator')]
    d_vars = [var for var in train_vars if var.name.startswith("discriminator")]
    saver = tf.train.Saver(var_list=g_vars)

    # 一些用于保存的处理
    losses = []
    samples = []
```

```python
# loss & optimizer
g_loss, d_loss, g_train_opt, d_train_opt = loss_and_optimizer(d_logits_real, d_logits_fake, g_vars, d_vars)

with tf.Session() as sess:  # 这么写，当不需要该session时，会自动释放资源
    sess.run(tf.global_variables_initializer())
    for e in range(1, epochs + 1):
        for it in range(1, mnist.train.num_examples//batch_size+1):
            batch = mnist.train.next_batch(batch_size)
            # 对图像像素进行scale，这是因为tanh输出的结果介于(-1, 1)，与生成图像的像素相对应
            batch_imgs = batch[0].reshape((batch_size, 784))  # batch[0]是tuple类型的
            batch_imgs = batch_imgs * 2 - 1
            # 随机生成generator的输入噪声
            noise_imgs = np.random.uniform(-1, 1, size=(batch_size, 100))
            # run optimizers
            _ = sess.run(g_train_opt, feed_dict={z_noise: noise_imgs})
            _ = sess.run(d_train_opt, feed_dict={x_real: batch_imgs, z_noise: noise_imgs})
            # # 打印并保存loss，生成图像（这里一次生成25张）并保存，对应DCGAN的情况
            # if it % 150 == 0:
            #     g_loss_curr = g_loss.eval({z_noise: noise_imgs})
            #     d_loss_curr = d_loss.eval({x_real: batch_imgs, z_noise: noise_imgs})
            #     losses.append([g_loss_curr, d_loss_curr])
            #     print("Epoch {}/{}...".format(e, epochs),
            #           "Generator Loss: {:.4f}...".format(g_loss_curr),
            #           "Discriminator Loss: {:.4f}".format(d_loss_curr))
            #     sample_noise = np.random.uniform(-1, 1, size=(n_sample, 100))
            #     gen_samples = sess.run(generator(z_noise, reuse=True), feed_dict={z_noise: sample_noise})
            #     samples.append(gen_samples)
        # 打印并保存loss，生成图像（这里一次生成25张）并保存，对应GAN的情况
        g_loss_curr = g_loss.eval({z_noise: noise_imgs})
        d_loss_curr = d_loss.eval({x_real: batch_imgs, z_noise: noise_imgs})
        losses.append([g_loss_curr, d_loss_curr])
        print("Epoch {}/{}...".format(e, epochs),
              "Generator Loss: {:.4f}...".format(g_loss_curr),
              "Discriminator Loss: {:.4f}".format(d_loss_curr))
```

```
            sample_noise = np.random.uniform(-1, 1, size=(n_sample, 100))
            gen_samples = sess.run(generator(z_noise, reuse=True), feed_dict={z_noise: sample_noise})
            samples.append(gen_samples)
            saver.save(sess, 'gan_saves/gan.ckpt')  # 存储checkpoints

    # 将loss和sample用pickle序列化后保存到文件（这里序列化为二进制形式）
    with open('gan_saves/losses.pkl', 'wb') as f:
        pickle.dump(losses, f)
    with open('gan_saves/samples.pkl', 'wb') as f:
        pickle.dump(samples, f)
```

Testing Model

```
# 绘制训练过程中的loss曲线
with open('gan_saves/losses.pkl', 'rb') as f:
    losses = pickle.load(f)
plt.figure(figsize=(20, 7))
plt.title('The Variation of Loss during Training')
plt.xlabel('Number of Epochs')
# plt.xlabel('Number of Epochs x 5')
plt.ylabel('Loss')
plt.plot(np.array(losses).T[0], label='Generator Loss')
plt.plot(np.array(losses).T[1], label='Discriminator Loss')
plt.legend()
plt.show()
```

Ian Goodfellow论文中的GAN的训练过程中的loss曲线

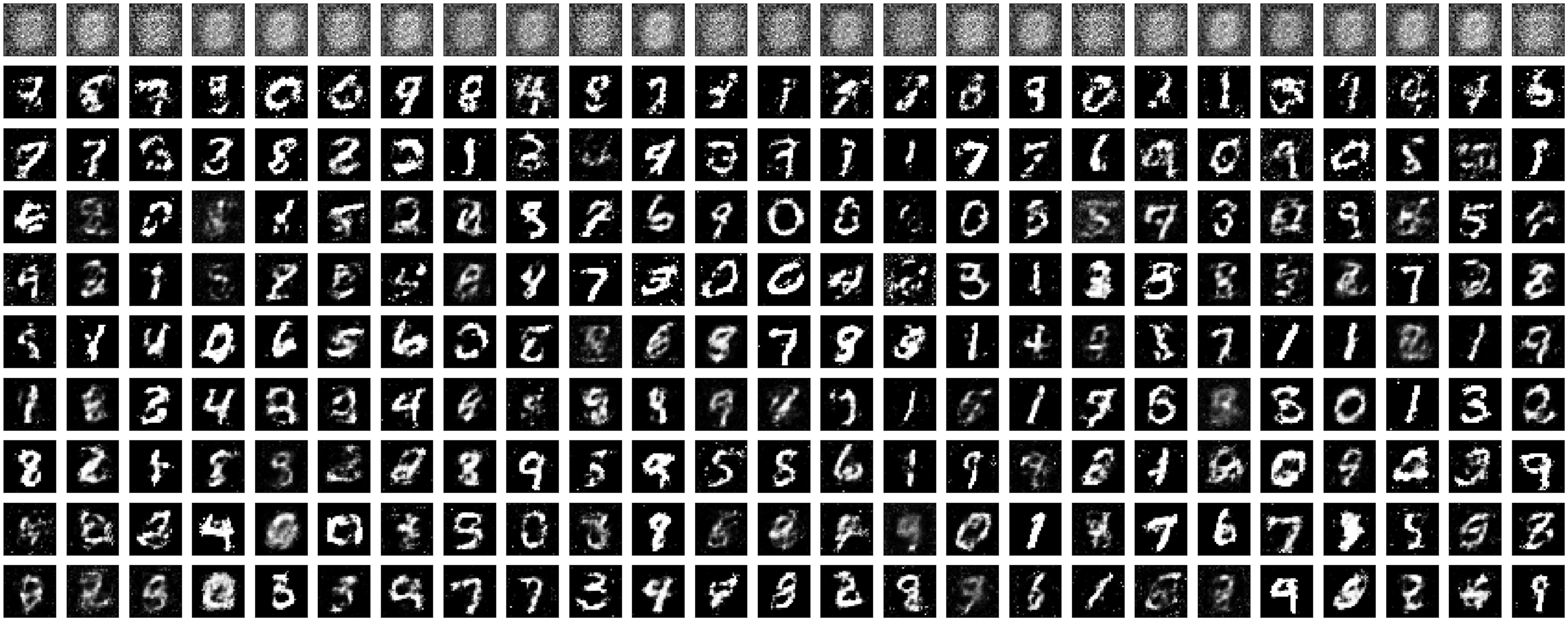
Alec Radford论文中的DCGAN（简易版本）的训练过程中的loss曲线

```python
# 花式显示生成图像
with open('gan_saves/samples.pkl', 'rb') as f:
    samples = pickle.load(f)
# 第一种，显示last epoch的生成图像
_, axes = plt.subplots(figsize=(7, 7), nrows=5, ncols=5, sharex='all', sharey='all')
for ax, img in zip(axes.flatten(), samples[-1]):
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.imshow(img.reshape((28, 28)), cmap='gray')
plt.show()

# 第二种，生成新的图片
saver = tf.train.Saver(var_list=g_vars)  # 加载生成器变量
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('gan_saves/'))
    sample_noise = np.random.uniform(-1, 1, size=(25, 100))
    gen_samples = sess.run(generator(z_noise, reuse=True), feed_dict={z_noise: sample_noise})
_, axes = plt.subplots(figsize=(7, 7), nrows=5, ncols=5, sharex='all', sharey='all')
for ax, img in zip(axes.flatten(), [gen_samples][0]):
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.imshow(img.reshape((28, 28)), cmap='gray')
plt.show()

# 第三种，采样显示整个训练过程的生成图像
epoch_idx = [i for i in range(0, len(samples), int(len(samples) / 10))]  # 指定要查看的sample，注意不要越界
show_imgs = [samples[i] for i in epoch_idx]
_, axes = plt.subplots(figsize=(30, 12), nrows=10, ncols=25, sharex='all', sharey='all')
for ax_row, sample in zip(axes, show_imgs):
    for ax, img in zip(ax_row, sample[::int(len(sample) / 25)]):
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        ax.imshow(img.reshape((28, 28)), cmap='gray')
plt.show()
```

Ian Goodfellow论文中的GAN的训练过程中的生成图像



Alec Radford论文中的DCGAN（简易版本）的训练过程中的生成图像