生成对抗网络 (GANs)

编写：张礼俊/SlyneD

校对：毛丽

总校对与审核：寒小阳

CS231N到目前位置，所有对神经网络的应用都是**判别式模型**，给定一个输入，训练产生一个label输出。从直接对一个图片的分类到句子生成(也是一个分类问题，我们的label是在词空间中，我们会去逐个学习来产生多词label)。在这个作业中，我们会拓展开来，用神经网络来构建一个**生成式模型**。特别的，我们会学习如何构建模型来生成与训练集类似的图片。

GAN是什么?

在2014年，Goodfellow et al.发表了训练生成模型的一个方法：生成对抗网络（GANs）。在一个GAN中，我们构建两个不同的神经网络。 第一个网络是传统的分类网络叫**判别器**。 我们会用判别器来判断图片是真实的(属于训练集)还是假的(不在训练集中)。另一个网络，叫做**生成器**，会把随机噪音作为输入，然后用一个神经网络通过它生成图片。生成器的目标就是为了骗过判别器，让判别器以为生成的图片是真的。

我们可以把这个想成是一个最小最大博弈(minimax game), 生成器 (G)反复的想要糊弄判别器，而判别器 (D)则要努力的正确区分真实还是假的。 $\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1-D(G(z)))\right]$ 其中 $x\sim p_{\text{data}}$ 是来自于输入数据的样本， $z\sim p(z)$ 是随机噪音样本，$G(z)$ 是用生成网络G生成的图片，D 是判别器的输出，指的是一个输入是真实图片的概率。 在Goodfellow et al., 他们分析了最小最大博弈并且展示了它和最小化训练数据分布和G的生成样本分布之间的Jensen-Shannon散度的关系。

为了优化这个最小最大博弈，我们会在对于G的目标上采用梯度下降和在 D的目标上采用梯度上升之间转换。

1. 更新生成器(G)来最小化**判别器做出正确选择**的概率
2. 更新判别器(D)来最大化**判别器做出正确选择**的概率

虽然这些更新理论上是有很有用的，但是在实际中他们表现并不好。取而代之的是，当我们更新生成器的时候，我们会用不同的目标函数：最大化**判别器做出错误选择**的概率。这一个小小的改变减轻了由于判别器置信度非常高的时候导致的生成器梯度消失的问题。这也是在大部分GAN的论文中的标准更新方法，并且在 Goodfellow et al.原始论文中也是这么用的。

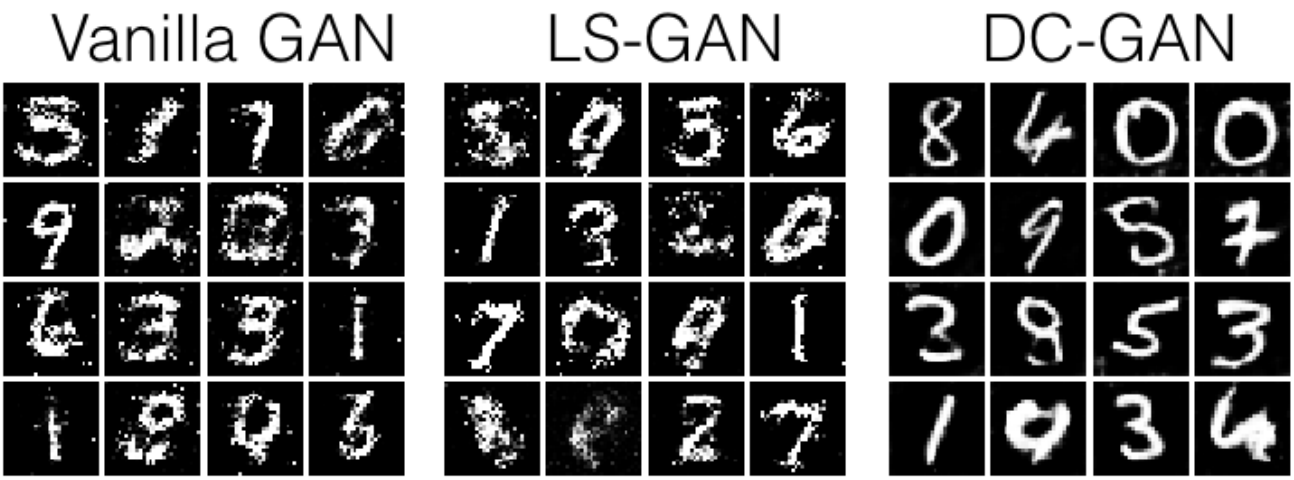
在这个作业中，我们会交替进行下面的更新:

1. 更新生成器(G)来最大化判别器在生成数据上做出错误选择的概率
   $\underset{G}{\text{maximize}}\ \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$
2. 更新判别器(D)来最大化判别器在真实以及生成数据上做出正确选择的概率
   $\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1-D(G(z)))\right]$

一些其它的工作

自2014年以来，GAN被应用到广泛的研究领域中，有大量的workshops,以及上百篇论文。 相比于生成模型的其它方法，他们通常生成最高质量的样本，但是训练起来也是最难和繁琐的(详见this github repo 包含了17中方法，对于训练模型很有帮助)。提升GAN的训练的稳定性和鲁棒性一直是一个开放的研究问题，每天都有新的论文。最近的GANs的教程，详见here.也有一些最近更让人兴奋的工作，把目标函数变成了Wasserstein距离，在不同的模型架构中生成了更加稳定的结果。WGAN, WGAN-GP.

GANs并不是唯一的训练生成模型的方法！对于其它的生成模型可以参考深度学习书book的deep generative model chapter 。 另一个流行的训练神经网络作为生成模型的方法是变分自动编码器(Variational Autoencoders) (co-discovered here and here)。 变分自动编码器用变分推断来训练深度生成模型。这些模型训练起来更稳定和容易，但是现在还没有GANs生成的图片那么漂亮。



你可以预期会生成的图片，如上所示，你的可能稍微有点不一样

构造

```
from __future__ import print_function, division
import tensorflow as tf
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# A bunch of utility functions

def show_images(images):
    images = np.reshape(images, [images.shape[0], -1])  # images reshape to (batch_size, D)
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
```

```
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

def preprocess_img(x):
    return 2 * x - 1.0

def deprocess_img(x):
    return (x + 1.0) / 2.0

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def count_params():
    """Count the number of parameters in the current TensorFlow graph """
    param_count = np.sum([np.prod(x.get_shape().as_list()) for x in tf.global_variables()])
    return param_count


def get_session():
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    session = tf.Session(config=config)
    return session

answers = np.load('gan-checks-tf.npz')
```

数据集

GANs对于超参数是要求很高的，也需要训练很多轮。为了能让这个作业在没有GPU的情况下也可以完成，我们会在MNIST数据集上做，其中60000张作为训练集，10000作为测试集。每张图片的当中都是一个白色的数字在黑色的背景上(0-9)。这也是训练卷积网络最早实用的数据集之一，相对也比较简单，一个标准的CNN模型就可以轻松超过99%的准确率。

为了简化我们的代码，我们会用TensorFlow MNIST的封装，它会下载和加载MNIST数据集，见这个文档。默认的参数会用5000张训练样本作为验证集。数据会被存到一个文件夹叫`MNIST_data`(笔者注:同学们也可以自己下载数据集后放到对应的文件夹下就可以了)

敲黑板: 注意TensorFlow MNIST的封装会把图片作为向量返回，也就是说他们的尺寸是(batch, 784)，如果你想把他们作为图片处理，就需要resize他们变成(batch,28,28)或者(batch,28,28,1)。 它们的类型是np.float32并且位于[0,1]之间。

```
from tensorflow.examples.tutorials.mnist import input_data
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"

mnist = input_data.read_data_sets('./cs231n/datasets/MNIST_data', one_hot=False)

# show a batch
#show_images(mnist.train.next_batch(16)[0])
```

```
Extracting ./cs231n/datasets/MNIST_data/train-images-idx3-ubyte.gz
Extracting ./cs231n/datasets/MNIST_data/train-labels-idx1-ubyte.gz
Extracting ./cs231n/datasets/MNIST_data/t10k-images-idx3-ubyte.gz
Extracting ./cs231n/datasets/MNIST_data/t10k-labels-idx1-ubyte.gz
```

LeakyReLU

实现一个LeakyReLU，参见课堂笔记或者这篇论文的等式(3)this paper。 LeakyReLUs使得ReLU的元素不"失活"，在GANS的方法中经常实用(maxout也可以，但是这会增加整个模型的大小，因此这个作业里没有用)。

```
def leaky_relu(x, alpha=0.01):
    """Compute the leaky ReLU activation function.

    Inputs:
    - x: TensorFlow Tensor with arbitrary shape
    - alpha: leak parameter for leaky ReLU

    Returns:
    TensorFlow Tensor with the same shape as x
    """
    # TODO: implement leaky ReLU
    condition = tf.less(x, 0)
    res = tf.where(condition, alpha * x, x)

    return res
```

测试你的leaky ReLU 实现。你的误差应当小于1e-10。

```
def test_leaky_relu(x, y_true):
    tf.reset_default_graph()
    with get_session() as sess:
        y_tf = leaky_relu(tf.constant(x))
        y = sess.run(y_tf)
        print('Maximum error: %g'%rel_error(y_true, y))

test_leaky_relu(answers['lrelu_x'], answers['lrelu_y'])
```

```
Maximum error: 0
```

随机噪声

生成一个在-1到1之间大小为[batch_size, dim]的均匀分布的噪音张量(Tensor)。

```
def sample_noise(batch_size, dim):
    """Generate random uniform noise from -1 to 1.

    Inputs:
    - batch_size: integer giving the batch size of noise to generate
    - dim: integer giving the dimension of the the noise to generate

    Returns:
    TensorFlow Tensor containing uniform noise in [-1, 1] with shape [batch_size, dim]
    """
    # TODO: sample and return noise
    return tf.random_uniform([batch_size,dim], minval=-1, maxval=1)
```

确保噪音的大小和类型是正确的:

```
def test_sample_noise():
    batch_size = 3
    dim = 4
    tf.reset_default_graph()
    with get_session() as sess:
        z = sample_noise(batch_size, dim)
        # Check z has the correct shape
        assert z.get_shape().as_list() == [batch_size, dim]
        # Make sure z is a Tensor and not a numpy array
        assert isinstance(z, tf.Tensor)
        # Check that we get different noise for different evaluations
        z1 = sess.run(z)
        z2 = sess.run(z)
        assert not np.array_equal(z1, z2)
        # Check that we get the correct range
        assert np.all(z1 >= -1.0) and np.all(z1 <= 1.0)
        print("All tests passed!")

test_sample_noise()
```

```
All tests passed!
```

判别器

首先我们来建一个判别器，你可以用tf.layers来构建层。 所有的全连接层都应该包含偏置项(bias)。

架构:

- 全连接层，从784到256
- LeakyReLU， alpha为0.01
- 全连接层，从256到256
- LeakyReLU, alpha为0.01
- 全连接层，从256到1

判别器的输出应该是[batch_size, 1]，并且包含了实数来表示batch_size中的每张图片是真实图片的打分。

```
def discriminator(x):
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    with tf.variable_scope("discriminator"):
        # TODO: implement architecture
        fc1 = tf.layers.dense(x, units=256, use_bias=True, name="first_fc")
        leaky_relu1 = leaky_relu(fc1, alpha=0.01)
        fc2 = tf.layers.dense(leaky_relu1, units=256, use_bias=True, name="second_fc")
        leaky_relu2 = leaky_relu(fc2, alpha=0.01)
        logits = tf.layers.dense(leaky_relu2, units=1, name="logits")
        return logits
```

测试以确保判别器中参数数量是正确的:

```
def test_discriminator(true_count=267009):
    tf.reset_default_graph()
    with get_session() as sess:
        y = discriminator(tf.ones((2, 784)))
        cur_count = count_params()
        if cur_count != true_count:
            print('Incorrect number of parameters in discriminator. {0} instead of {1}. Check
your achitecture.'.format(cur_count,true_count))
        else:
            print('Correct number of parameters in discriminator.')

test_discriminator()
```

```
Correct number of parameters in discriminator.
```

生成器

现在来构建一个生成器，你可以用tf.layers来构建这个模型，所有的全连接层都要包含偏置项。

架构:

- 全连接层从tf.shape(z)[1] (噪音的纬度) 到1024
- ReLU
- 全连接层从1024到1024
- ReLU
- 全连接层从1024到784
- TanH (严格限定输出是[-1,1])

```
def generator(z):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    with tf.variable_scope("generator"):
        # TODO: implement architecture
        fc1 = tf.layers.dense(z, units=1024, use_bias=True)
        relu1 = tf.nn.relu(fc1)
        fc2 = tf.layers.dense(relu1, units=1024, use_bias=True)
        relu2 = tf.nn.relu(fc2)
        fc3 = tf.layers.dense(relu2, units=784, use_bias=True)
        img = tf.nn.tanh(fc3)
        return img
```

测试以确保生成器参数的数量是正确的:

```
def test_generator(true_count=1858320):
    tf.reset_default_graph()
    with get_session() as sess:
        y = generator(tf.ones((1, 4)))
        cur_count = count_params()
        if cur_count != true_count:
            print('Incorrect number of parameters in generator. {0} instead of {1}. Check your
achitecture.'.format(cur_count,true_count))
        else:
            print('Correct number of parameters in generator.')

test_generator()
```

```
Correct number of parameters in generator.
```

GAN 损失函数

上面的公式是生成器和判别器的损失函数。这两个公式是之前的公式前面加了一个负号，因为我们现在是要最小化这些loss。(之前我们说的都是最大化xxx的概率) 这里可以用tf.ones_like和tf.zeros_like来为你的判别器生成label。用sigmoid_cross_entropy_loss来计算损失函数。我们不用期望来计算，而是直接计算minibatch上面的元素均值，所以要注意用的是平均而不是求和。

```
def gan_loss(logits_real, logits_fake):
    """Compute the GAN loss.

    Inputs:
    - logits_real: Tensor, shape [batch_size, 1], output of discriminator
        Log probability that the image is real for each real image
    - logits_fake: Tensor, shape[batch_size, 1], output of discriminator
        Log probability that the image is real for each fake image

    Returns:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar
    """
    # TODO: compute D_loss and G_loss
    D_loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(logits_real),
logits=logits_real, name="discriminator_real_loss")+ \
             tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.zeros_like(logits_fake),
logits=logits_fake, name="discriminator_fake_loss")
    G_loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(logits_fake),
logits=logits_fake, name="generator_loss")
    D_loss = tf.reduce_mean(D_loss)
    G_loss = tf.reduce_mean(G_loss)
    return D_loss, G_loss
```

测试GAN的损失函数。确定生成器和判别器的损失都是正确的。你的误差应小于1e-5。

```
def test_gan_loss(logits_real, logits_fake, d_loss_true, g_loss_true):
    tf.reset_default_graph()
    with get_session() as sess:
        d_loss, g_loss = sess.run(gan_loss(tf.constant(logits_real), tf.constant(logits_fake)))
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_gan_loss(answers['logits_real'], answers['logits_fake'],
              answers['d_loss_true'], answers['g_loss_true'])
```

```
Maximum error in d_loss: 0
Maximum error in g_loss: 0
```

优化损失函数

用AdamOptimizer，1e-3的学习率, beta1=0.5来分别最小化生成器和判别器的损失函数。减小beta值的这个技巧在帮助GANs收敛上很有效，参考Improved Techniques for Training GANs。事实上，在我们当前的高参下，如果你把beta1设置为Tensorflow的默认值0.9，有一定的概率会让你的判别器loss变成0，生成器就完全不能学了。这也是GANs失败的一个常见的模式;如果判别器学习的太快(比如，损失函数值接近0)，你的生成器就再也学不了了。

通常判别器是用SGD加上Momentum或者RMSProp而不是Adam,但是这里我们的判别器和生成器都用了Adam。

```
# TODO: create an AdamOptimizer for D_solver and G_solver
def get_solvers(learning_rate=1e-3, beta1=0.5):
    """Create solvers for GAN training.

    Inputs:
    - learning_rate: learning rate to use for both solvers
    - beta1: beta1 parameter for both solvers (first moment decay)

    Returns:
    - D_solver: instance of tf.train.AdamOptimizer with correct learning_rate and beta1
    - G_solver: instance of tf.train.AdamOptimizer with correct learning_rate and beta1
    """
    D_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1)
    G_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1)
    return D_solver, G_solver
```

组合所有部分

下面做一些简单的组合.....这部分要仔细的阅读，确保明白我们是怎么组合生成器以及判别器的。

```
tf.reset_default_graph()

# number of images for each batch
batch_size = 128
# our noise dimension
noise_dim = 96

# placeholder for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
# random noise fed into our generator
z = sample_noise(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(preprocess_img(x))
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'generator')

# get our solver
D_solver, G_solver = get_solvers()

# get our loss
D_loss, G_loss = gan_loss(logits_real, logits_fake)
```

```
# setup training steps
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'generator')
```

训练 GAN!

并不难，是吧? 在最初的100s内的迭代，你应该可以看到黑色的背景，当你到1000步的时候会出现模糊的图片，当我们超过3000步的时候会看到清楚的形状，大约一半的图片都是清晰可辨的。在我们的训练中，我们在每次迭代都都会训练一次判别器和生成器。然而，有些论文会实验不同的流程来训练判别器和生成器，有的时候训练一个比另一个多几次，或者甚至先把其中一个都训练到"足够好"，然后再训练另一个。

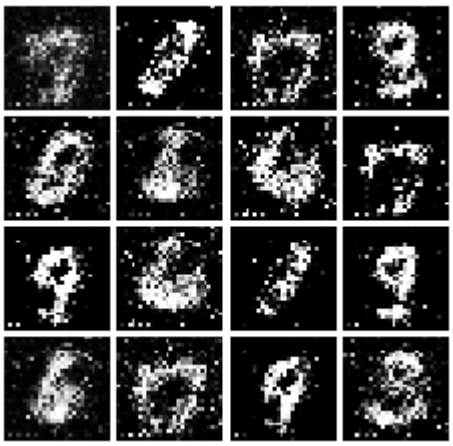
```
# a giant helper function
def run_a_gan(sess, G_train_step, G_loss, D_train_step, D_loss, G_extra_step, D_extra_step,\
              show_every=1000, print_every=50, batch_size=128, num_epoch=10):
    """Train a GAN for a certain number of epochs.

    Inputs:
    - sess: A tf.Session that we want to use to run our data
    - G_train_step: A training step for the Generator
    - G_loss: Generator loss
    - D_train_step: A training step for the Generator
    - D_loss: Discriminator loss
    - G_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for generator
    - D_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for discriminator
    Returns:
        Nothing
    """
    # compute the number of iterations we need
    max_iter = int(mnist.train.num_examples*num_epoch/batch_size)
    for it in range(max_iter):
        # every show often, show a sample result
        if it % show_every == 0:
            samples = sess.run(G_sample)
            fig = show_images(samples[:16])
            plt.show()
            print()
        # run a batch of data through the network
        minibatch,minbatch_y = mnist.train.next_batch(batch_size)
        _, D_loss_curr = sess.run([D_train_step, D_loss], feed_dict={x: minibatch})
        _, G_loss_curr = sess.run([G_train_step, G_loss])

        # print loss every so often.
        # We want to make sure D_loss doesn't go to 0
        if it % print_every == 0:
            print('Iter: {}, D: {:.4}, G:{:.4}'.format(it,D_loss_curr,G_loss_curr))
    print('Final images')
    samples = sess.run(G_sample)
```

```
    fig = show_images(samples[:16])
    plt.show()
```

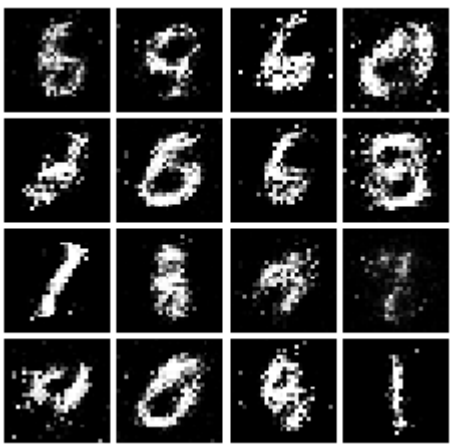
训练吧，GPU上大约是10分钟，GPU上小于1分钟

```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess,G_train_step,G_loss,D_train_step,D_loss,G_extra_step,D_extra_step)
```

```
Iter: 0, D: 1.907, G:0.7063
Iter: 50, D: 0.5351, G:1.408
Iter: 100, D: 0.9569, G:1.96
Iter: 150, D: 1.164, G:1.363
Iter: 200, D: 1.553, G:0.7027
Iter: 250, D: 1.656, G:0.2917
Iter: 300, D: 1.024, G:1.533
Iter: 350, D: 1.037, G:1.561
Iter: 400, D: 1.526, G:1.355
Iter: 450, D: 0.911, G:1.168
Iter: 500, D: 1.149, G:1.341
Iter: 550, D: 0.9293, G:2.111
Iter: 600, D: 1.001, G:1.016
Iter: 650, D: 1.098, G:1.346
Iter: 700, D: 1.238, G:0.5977
Iter: 750, D: 1.125, G:1.427
Iter: 800, D: 1.537, G:0.5509
Iter: 850, D: 1.29, G:1.125
Iter: 900, D: 1.207, G:1.741
Iter: 950, D: 0.9664, G:1.357
```



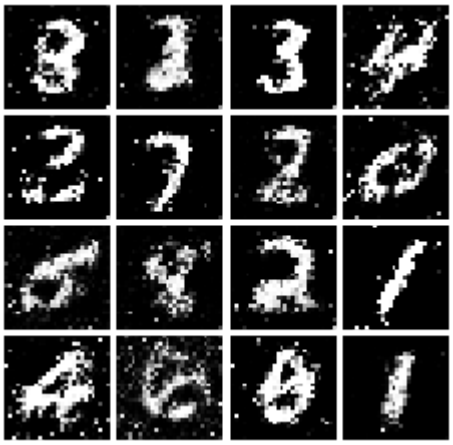
```
Iter: 1000, D: 1.248, G:1.411
Iter: 1050, D: 0.8648, G:1.189
Iter: 1100, D: 1.145, G:1.142
Iter: 1150, D: 1.312, G:0.7995
Iter: 1200, D: 1.196, G:1.221
Iter: 1250, D: 1.182, G:1.199
Iter: 1300, D: 1.204, G:1.085
Iter: 1350, D: 1.114, G:1.332
Iter: 1400, D: 1.14, G:0.9212
Iter: 1450, D: 1.31, G:0.8265
Iter: 1500, D: 1.104, G:1.052
Iter: 1550, D: 1.325, G:0.9979
Iter: 1600, D: 1.168, G:0.938
Iter: 1650, D: 1.369, G:0.7423
Iter: 1700, D: 1.361, G:0.8276
Iter: 1750, D: 1.279, G:1.054
Iter: 1800, D: 1.21, G:0.8292
Iter: 1850, D: 1.248, G:0.8468
Iter: 1900, D: 1.32, G:0.9191
Iter: 1950, D: 1.321, G:0.8962
```



```
Iter: 2000, D: 1.342, G:1.026
Iter: 2050, D: 1.235, G:1.374
Iter: 2100, D: 1.294, G:0.8541
Iter: 2150, D: 1.334, G:0.8669
Iter: 2200, D: 1.362, G:0.7651
Iter: 2250, D: 1.282, G:0.9314
Iter: 2300, D: 1.234, G:0.7797
Iter: 2350, D: 1.328, G:0.7483
Iter: 2400, D: 1.339, G:0.8163
Iter: 2450, D: 1.298, G:0.9128
Iter: 2500, D: 1.491, G:0.8884
Iter: 2550, D: 1.274, G:0.9663
Iter: 2600, D: 1.303, G:0.8186
Iter: 2650, D: 1.287, G:0.7887
Iter: 2700, D: 1.333, G:0.8452
Iter: 2750, D: 1.445, G:1.002
Iter: 2800, D: 1.262, G:0.8284
Iter: 2850, D: 1.279, G:0.8043
Iter: 2900, D: 1.333, G:0.9543
Iter: 2950, D: 1.324, G:0.8988
```

```
Iter: 3000, D: 1.319, G:0.8393
Iter: 3050, D: 1.315, G:0.8503
Iter: 3100, D: 1.295, G:0.8025
Iter: 3150, D: 1.349, G:0.855
Iter: 3200, D: 1.347, G:1.021
Iter: 3250, D: 1.281, G:0.9025
Iter: 3300, D: 1.374, G:0.8575
Iter: 3350, D: 1.266, G:0.8088
Iter: 3400, D: 1.237, G:0.9011
Iter: 3450, D: 1.254, G:0.9397
Iter: 3500, D: 1.28, G:0.8415

Iter: 3550, D: 1.292, G:0.8904
```

```
Iter: 3600, D: 1.373, G:0.8439
Iter: 3650, D: 1.416, G:0.8279
Iter: 3700, D: 1.253, G:0.92
Iter: 3750, D: 1.363, G:0.9052
Iter: 3800, D: 1.248, G:0.8335
Iter: 3850, D: 1.318, G:0.9213
Iter: 3900, D: 1.339, G:0.8299
Iter: 3950, D: 1.236, G:0.8503
```



```
Iter: 4000, D: 1.351, G:0.9842
Iter: 4050, D: 1.343, G:0.8414
Iter: 4100, D: 1.29, G:0.8869
Iter: 4150, D: 1.321, G:0.76
Iter: 4200, D: 1.336, G:0.7591
Iter: 4250, D: 1.249, G:0.813
Final images
```

Least Squares GAN

接下来呢，我们改变一下我们的损失函数，变成这式，论文详见Least Squares GAN，这个新论文更加稳定。 生成器损失：$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))-1\right)^2\right]$ 判别器损失：$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[\left(D(x)-1\right)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))\right)^2\right]$ 同样的我们就不计算期望了，直接把minibatch里面的元素求平均

```
def lsgan_loss(score_real, score_fake):
    """Compute the Least Squares GAN loss.

    Inputs:
    - score_real: Tensor, shape [batch_size, 1], output of discriminator
        score for each real image
    - score_fake: Tensor, shape[batch_size, 1], output of discriminator
        score for each fake image

    Returns:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar
    """
    # TODO: compute D_loss and G_loss
    D_loss = 0.5 * tf.reduce_mean(tf.square(score_real-1)) + 0.5 *
tf.reduce_mean(tf.square(score_fake))
    G_loss = 0.5 * tf.reduce_mean(tf.square(score_fake-1))

    return D_loss, G_loss
```

测试你的LSGAN 损失。你的误差应该小于1e-7。

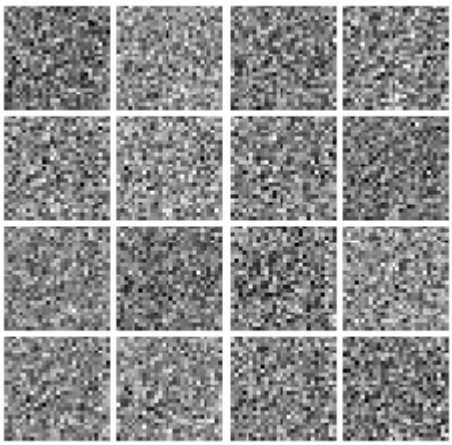
```
def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
    with get_session() as sess:
        d_loss, g_loss = sess.run(
            lsgan_loss(tf.constant(score_real), tf.constant(score_fake)))
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_lsgan_loss(answers['logits_real'], answers['logits_fake'],
                answers['d_loss_lsgan_true'], answers['g_loss_lsgan_true'])
```

```
Maximum error in d_loss: 0
Maximum error in g_loss: 0
```

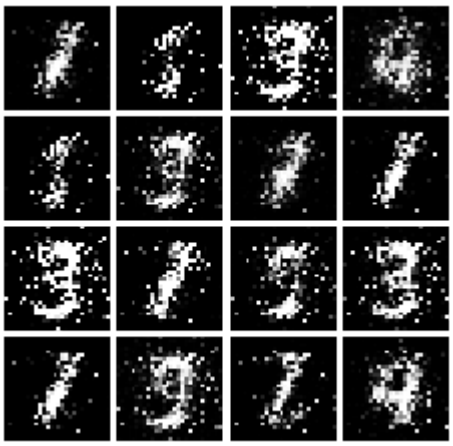
实现一个新的训练步骤，这样我们可以最小化LSGAN损失函数：

```
D_loss, G_loss = lsgan_loss(logits_real, logits_fake)
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
```

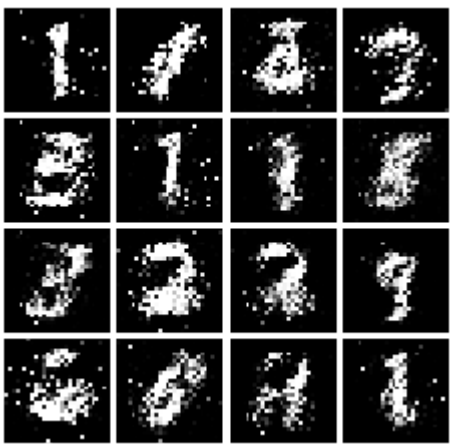
```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess, G_train_step, G_loss, D_train_step, D_loss, G_extra_step, D_extra_step)
```



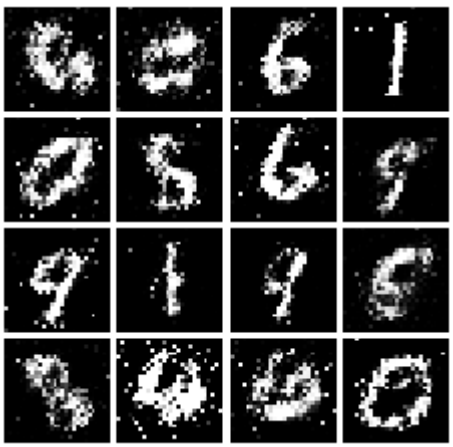
```
Iter: 0, D: 2.134, G:0.4007
Iter: 50, D: 0.03952, G:0.644
Iter: 100, D: 0.6377, G:0.1408
Iter: 150, D: 0.2105, G:0.3686
Iter: 200, D: 0.2091, G:0.3572
Iter: 250, D: 0.1055, G:0.5144
Iter: 300, D: 0.1433, G:0.3639
Iter: 350, D: 0.04518, G:0.5931
Iter: 400, D: 0.205, G:0.4512
Iter: 450, D: 0.1519, G:0.4475
Iter: 500, D: 0.1033, G:0.4973
Iter: 550, D: 0.1185, G:0.1725
Iter: 600, D: 0.1643, G:0.2601
Iter: 650, D: 0.1841, G:0.05871
Iter: 700, D: 0.1618, G:0.3077
Iter: 750, D: 0.1764, G:0.2961
Iter: 800, D: 0.199, G:0.2603
Iter: 850, D: 0.1684, G:0.2607
Iter: 900, D: 0.2062, G:0.1552
Iter: 950, D: 0.1325, G:0.3433
```



```
Iter: 1000, D: 0.1541, G:0.1126
Iter: 1050, D: 0.1326, G:0.3329
Iter: 1100, D: 0.1953, G:0.2548
Iter: 1150, D: 0.1355, G:0.2694
Iter: 1200, D: 0.1417, G:0.2822
Iter: 1250, D: 0.1625, G:0.2273
Iter: 1300, D: 0.2261, G:0.1918
Iter: 1350, D: 0.175, G:0.1922
Iter: 1400, D: 0.1739, G:0.2635
Iter: 1450, D: 0.1747, G:0.2179
Iter: 1500, D: 0.1567, G:0.4182
Iter: 1550, D: 0.1756, G:0.2844
Iter: 1600, D: 0.1753, G:0.2492
Iter: 1650, D: 0.1679, G:0.3978
Iter: 1700, D: 0.1404, G:0.3078
Iter: 1750, D: 0.1693, G:0.2454
Iter: 1800, D: 0.2236, G:0.258
Iter: 1850, D: 0.1722, G:0.2356
Iter: 1900, D: 0.2071, G:0.2309
Iter: 1950, D: 0.1988, G:0.2213
```



```
Iter: 2000, D: 0.1955, G:0.1973
Iter: 2050, D: 0.2171, G:0.1807
Iter: 2100, D: 0.2133, G:0.1837
Iter: 2150, D: 0.2126, G:0.1783
Iter: 2200, D: 0.2443, G:0.1973
Iter: 2250, D: 0.2244, G:0.2044
Iter: 2300, D: 0.243, G:0.1545
Iter: 2350, D: 0.1955, G:0.2028
Iter: 2400, D: 0.232, G:0.1908
Iter: 2450, D: 0.2073, G:0.205
Iter: 2500, D: 0.2058, G:0.2306
Iter: 2550, D: 0.2123, G:0.1918
Iter: 2600, D: 0.2238, G:0.1801
Iter: 2650, D: 0.2394, G:0.1486
Iter: 2700, D: 0.2015, G:0.1842
Iter: 2750, D: 0.2252, G:0.1781
Iter: 2800, D: 0.2399, G:0.2047
Iter: 2850, D: 0.2444, G:0.16
Iter: 2900, D: 0.2265, G:0.1662
Iter: 2950, D: 0.215, G:0.1731
```



```
Iter: 3000, D: 0.2088, G:0.1714
Iter: 3050, D: 0.2323, G:0.1676
Iter: 3100, D: 0.2274, G:0.1796
Iter: 3150, D: 0.234, G:0.1747
Iter: 3200, D: 0.2338, G:0.1781
Iter: 3250, D: 0.2267, G:0.1647
Iter: 3300, D: 0.2357, G:0.1579
Iter: 3350, D: 0.235, G:0.1427
Iter: 3400, D: 0.2203, G:0.1629
Iter: 3450, D: 0.2234, G:0.1648
Iter: 3500, D: 0.213, G:0.1644
Iter: 3550, D: 0.2167, G:0.1759
```

```
Iter: 3600, D: 0.2354, G:0.1655
Iter: 3650, D: 0.2335, G:0.1541
Iter: 3700, D: 0.2181, G:0.2017
Iter: 3750, D: 0.2378, G:0.1513
Iter: 3800, D: 0.2337, G:0.1639
Iter: 3850, D: 0.234, G:0.1666
Iter: 3900, D: 0.2271, G:0.1706
Iter: 3950, D: 0.2494, G:0.1586
```

```
Iter: 4000, D: 0.217, G:0.1854
Iter: 4050, D: 0.2101, G:0.1819
Iter: 4100, D: 0.2314, G:0.1938
Iter: 4150, D: 0.2052, G:0.1997
Iter: 4200, D: 0.2357, G:0.1679
Iter: 4250, D: 0.2131, G:0.1725
Final images
```

INLINE QUESTION 1:

描述一下在训练过程中的样本的视觉上的质量改变。你注意到样本的分布了吗？在不同的训练过程中结果是如何改变的？

Deep Convolutional GANs

在第一部分我们主要直接拷贝了Ian Goodfellow的GAN网络，然而这个网络不能够生成真实的空间上的特征。因为缺少卷积层，通常是没有办法生成出"锋利的边缘"。因此，这部分，我们实现DCGAN的部分想法，我们会用卷积网络来作为判别器和生成器。

判别器

我们会用一个类似于TensorFlow MNIST分类的判别器（教程），可以快速得到99%的准确率。全连接层需要的[N,D]这样的纬度张量，然而conv2d需要的是[N,H,W,C]这样的张量，所以同学们在构建的时候需要确保纬度的正确。

架构：

- 32 Filters, 5x5, Stride 1, Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- 64 Filters, 5x5, Stride 1, Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Flatten
- Fully Connected size 4 x 4 x 64, Leaky ReLU(alpha=0.01)
- Fully Connected size 1

```
def discriminator(x):
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    x = tf.reshape(x, shape=(tf.shape(x)[0],28,28,1))
    with tf.variable_scope("discriminator"):
        conv1 = tf.layers.conv2d(x, filters=32, kernel_size=(5,5), strides=(1,1),
activation=leaky_relu)
        max_pool1 = tf.layers.max_pooling2d(conv1, pool_size=(2,2), strides=(2,2))
        conv2 = tf.layers.conv2d(max_pool1, filters=64, kernel_size=(5,5), strides=(1,1),
activation=leaky_relu)
        max_pool2 = tf.layers.max_pooling2d(conv2, pool_size=(2,2), strides=(2,2))
        flat = tf.contrib.layers.flatten(max_pool2)
        fc1 = tf.layers.dense(flat, units=4*4*64, activation=leaky_relu)
        logits = tf.layers.dense(fc1, units=1)
        return logits
test_discriminator(1102721)
```

```
Correct number of parameters in discriminator.
```

Generator

生成器的架构直接是从InfoGAN paper上拷贝过来的，详见附录C.1。可以参考文档tf.nn.conv2d_transpose. 注意在GAN下面，我们用的都是"training"模式。

笔者注: 这里会涉及到batch normalization，所以需要设置training这个参数。另外，卷积的偏置项不要忘记了。

架构：

- Fully connected of size 1024, ReLU
- BatchNorm
- Fully connected of size 7 x 7 x 128, ReLU
- BatchNorm
- Resize into Image Tensor
- 64 conv2d^T (transpose) filters of 4x4, stride 2, ReLU
- BatchNorm
- 1 conv2d^T (transpose) filter of 4x4, stride 2, TanH

```
def generator(z):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    batch_size = tf.shape(z)[0]
    with tf.variable_scope("generator"):
        # TODO: implement architecture
        fc1 = tf.layers.dense(z, units=1024, activation=tf.nn.relu, use_bias=True)
        bn1 = tf.layers.batch_normalization(fc1, training=True)
        fc2 = tf.layers.dense(bn1, units=7*7*128, activation=tf.nn.relu, use_bias=True)
        bn2 = tf.layers.batch_normalization(fc2, training=True)
        resize = tf.reshape(bn2, shape=(-1,7,7,128))
        filter_conv1 = tf.get_variable('deconv1',[4,4,64,128]) # [height, width,
output_channels, in_channels]
        conv_tr1 = tf.nn.conv2d_transpose(resize,filter=filter_conv1, output_shape=
[batch_size,14,14,64],strides=[1,2,2,1])
        bias1 = tf.get_variable("deconv1_bias", [64])
        conv_tr1 = conv_tr1 + bias1
        relu_conv_tr1 = tf.nn.relu(conv_tr1)
        bn3 = tf.layers.batch_normalization(relu_conv_tr1, training=True)
        filter_conv2 = tf.get_variable('deconv2',[4,4,1,64]) #[height, width, output_channels,
in_channels]


        conv_tr2 = tf.nn.conv2d_transpose(bn3, filter=filter_conv2,output_shape=
```

```
[batch_size,28,28,1], strides=[1,2,2,1])
        bias2 = tf.get_variable("deconv2_bias",[1])
        conv_tr2 = conv_tr2 + bias2
        img = tf.nn.tanh(conv_tr2)
        img = tf.contrib.layers.flatten(img)
        return img
test_generator(6595521)
```

```
Correct number of parameters in generator.
```

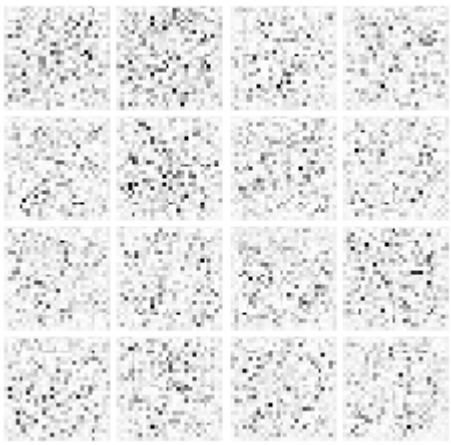
由于我们改动了方程，需要重新实现网络。

```
tf.reset_default_graph()

batch_size = 128
# our noise dimension
noise_dim = 96

# placeholders for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
z = sample_noise(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(preprocess_img(x))
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'generator')

D_solver,G_solver = get_solvers()
D_loss, G_loss = gan_loss(logits_real, logits_fake)
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'discriminator') # 因为加了batch norm，
这步不可省略
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'generator') # 因为加了batch norm，这步
不可省略
```

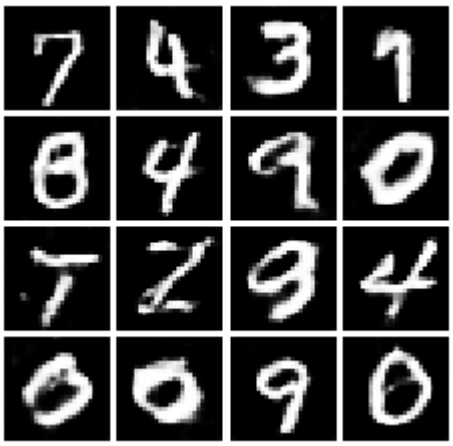
训练和评估 DCGAN

在GPU上这一步跑5个epoches只要三分钟左右，在一个双核的GPU上会需要50分钟(如果你用CPU，你可以改成3个epochs)

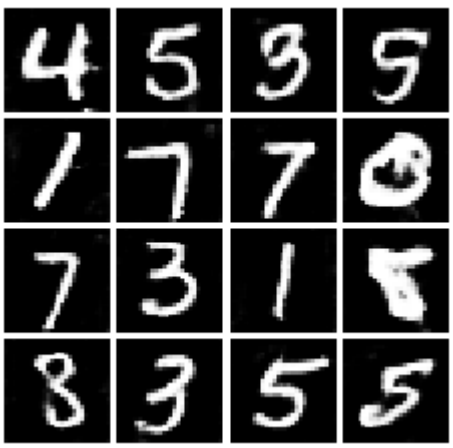
```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())

 run_a_gan(sess,G_train_step,G_loss,D_train_step,D_loss,G_extra_step,D_extra_step,num_epoch=5)
```



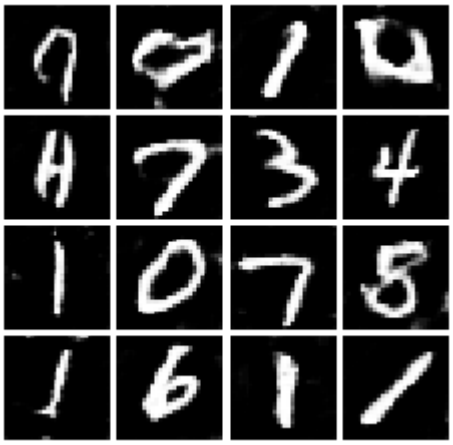
```
Iter: 0, D: 1.417, G:0.7949
Iter: 50, D: 0.777, G:2.711
Iter: 100, D: 0.7819, G:2.196
Iter: 150, D: 1.089, G:3.214
Iter: 200, D: 2.472, G:0.6526
Iter: 250, D: 0.9326, G:0.9724
Iter: 300, D: 1.049, G:1.275
Iter: 350, D: 0.9999, G:0.9933
Iter: 400, D: 1.023, G:1.007
Iter: 450, D: 1.201, G:0.6148
Iter: 500, D: 1.023, G:1.215
Iter: 550, D: 1.061, G:0.9566
Iter: 600, D: 1.53, G:0.9461
Iter: 650, D: 1.095, G:1.223
Iter: 700, D: 1.11, G:0.6179
Iter: 750, D: 1.374, G:0.9979
Iter: 800, D: 1.039, G:1.436
Iter: 850, D: 1.132, G:0.549
Iter: 900, D: 1.12, G:0.9184
Iter: 950, D: 1.119, G:1.167
```



```
Iter: 1000, D: 1.123, G:1.379
Iter: 1050, D: 1.142, G:1.595
Iter: 1100, D: 1.077, G:1.43
Iter: 1150, D: 1.051, G:0.9881
Iter: 1200, D: 0.9595, G:1.1
Iter: 1250, D: 1.037, G:1.24
Iter: 1300, D: 0.9239, G:1.315
Iter: 1350, D: 1.143, G:1.263
Iter: 1400, D: 1.074, G:0.884
Iter: 1450, D: 1.002, G:1.099
Iter: 1500, D: 1.065, G:1.657
Iter: 1550, D: 1.056, G:1.313
Iter: 1600, D: 1.073, G:0.967
Iter: 1650, D: 1.067, G:1.19
Iter: 1700, D: 1.027, G:1.66
Iter: 1750, D: 1.086, G:0.8463
Iter: 1800, D: 1.199, G:0.9816
Iter: 1850, D: 0.9872, G:1.112
Iter: 1900, D: 0.9564, G:1.001
Iter: 1950, D: 1.077, G:1.949
```



```
Iter: 2000, D: 1.051, G:1.131
Iter: 2050, D: 1.012, G:1.218
Iter: 2100, D: 1.056, G:1.224
Final images
```



INLINE QUESTION 2:

你看到DCGAN和原先的GAN上面结果的不同是神马？

Extra Credit

** Be sure you don't destroy your results above, but feel free to copy+paste code to get results below **

- For a small amount of extra credit, you can implement additional new GAN loss functions below, provided they converge. See AFI, BiGAN, Softmax GAN, Conditional GAN, InfoGAN, etc. They should converge to get credit.
- Likewise for an improved architecture or using a convolutional GAN (or even implement a VAE)
- For a bigger chunk of extra credit, load the CIFAR10 data (see last assignment) and train a compelling generative model on CIFAR-10
- Demonstrate the value of GANs in building semi-supervised models. In a semi-supervised example, only some fraction of the input data has labels; we can supervise this in MNIST by only training on a few dozen or hundred labeled examples. This was first described in Improved Techniques for Training GANs.
- Something new/cool.

Describe what you did here

加分部分，读者自行实现吧~

WGAN-GP (Small Extra Credit)

在实现了上面的模型之后，我们可以参考新的论文Improved Wasserstein GAN实现一个更鲁棒的损失函数。我们在这一部分要修改损失函数，重新训练。我们此处要实现论文的Algorithm 1。 你要用一个没有max_pooling的生成器和判别器。所以之前的DCGAN里面的生成器和判别器就不适用了。需要使用DCGAN里面的生成器和论文InfoGAN中的判别器。 结构：

- 64 Filters of 4x4, stride 2, LeakyReLU
- 128 Filters of 4x4, stride 2, LeakyReLU
- BatchNorm
- Flatten
- Fully connected 1024, LeakyReLU
- Fully connected size 1

这部分类似于上面，只是我们这次要实验WGAN，其实也只需要改变损失函数，然后重新训练模型。你也可以自己定义判别器和生成器的结构。 另外后面我们在实现损失函数的时候要仔细一些，参考论文里面描述的应该就没有什么问题了~

```
def discriminator(x):
    x = tf.reshape(x, [tf.shape(x)[0],28,28,1])
    with tf.variable_scope('discriminator'):
        # TODO: implement architecture
        conv1 = tf.layers.conv2d(x, filters=64, kernel_size=(4,4), strides=(2,2),
activation=leaky_relu)
        conv2 = tf.layers.conv2d(conv1, filters=128, kernel_size=(4,4), strides=(2,2),
activation=leaky_relu)
        bn2 = tf.layers.batch_normalization(conv2, training=True)
        flat = tf.contrib.layers.flatten(bn2)
        fc1 = tf.layers.dense(flat, units=1024, use_bias=True, activation=leaky_relu)
        logits = tf.layers.dense(fc1, units=1, use_bias=True)
        return logits
test_discriminator(3411649)
```

```
Correct number of parameters in discriminator.
```

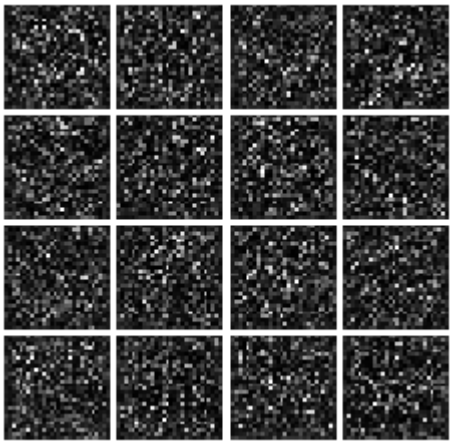
```
tf.reset_default_graph()

batch_size = 128
# our noise dimension
noise_dim = 96

# placeholders for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
z = sample_noise(batch_size, noise_dim)
# generated images

G_sample = generator(z)
```

```
with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(preprocess_img(x))
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'generator')

D_solver, G_solver = get_solvers()
```

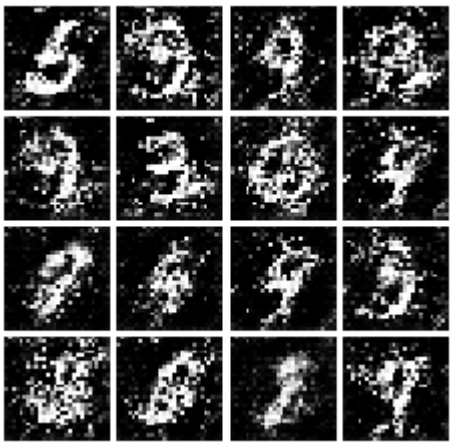
```
def wgangp_loss(logits_real, logits_fake, batch_size, x, G_sample):
    """Compute the WGAN-GP loss.

    Inputs:
    - logits_real: Tensor, shape [batch_size, 1], output of discriminator
        Log probability that the image is real for each real image
    - logits_fake: Tensor, shape[batch_size, 1], output of discriminator
        Log probability that the image is real for each fake image
    - batch_size: The number of examples in this batch
    - x: the input (real) images for this batch
    - G_sample: the generated (fake) images for this batch

    Returns:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar
    """
    # TODO: compute D_loss and G_loss
    D_loss = None
    G_loss = None

    # lambda from the paper
    lam = 10

    # random sample of batch_size (tf.random_uniform)
    eps = tf.random_uniform([1], minval=0, maxval=1)
    x_hat = eps * x + (1-eps) * G_sample

    # Gradients of Gradients is kind of tricky!
    with tf.variable_scope('',reuse=True) as scope:
        grad_D_x_hat = tf.gradients(discriminator(x_hat),x_hat)

    grad_norm = tf.norm(grad_D_x_hat)
    grad_pen = tf.square(grad_norm - 1)

    D_loss =  tf.reduce_mean(logits_fake) - tf.reduce_mean(logits_real) + lam *
tf.reduce_mean(grad_pen)
    G_loss = -tf.reduce_mean(logits_fake)
    return D_loss, G_loss
```

```
D_loss, G_loss = wgangp_loss(logits_real, logits_fake, 128, x, G_sample)
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'generator')
```

```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())

 run_a_gan(sess,G_train_step,G_loss,D_train_step,D_loss,G_extra_step,D_extra_step,batch_size=128
,num_epoch=5)
```



```
Iter: 0, D: 6.056e+03, G:-0.08695
Iter: 50, D: 476.2, G:-1.548
Iter: 100, D: 365.6, G:-0.9713
Iter: 150, D: 34.9, G:-0.7107
Iter: 200, D: 19.47, G:-0.647
Iter: 250, D: 8.324, G:-0.557
Iter: 300, D: 3.603, G:-0.5202
Iter: 350, D: 1.024, G:-0.5002
Iter: 400, D: 0.2818, G:-0.5114
Iter: 450, D: 1.186, G:-0.5261
Iter: 500, D: 0.3026, G:-0.564
Iter: 550, D: -0.0892, G:-0.6268
Iter: 600, D: 0.02073, G:-0.7765
Iter: 650, D: 0.111, G:-0.6887
Iter: 700, D: 0.09421, G:-0.5915
Iter: 750, D: 0.1328, G:-0.638
Iter: 800, D: 0.1107, G:-0.6698
Iter: 850, D: 0.1018, G:-0.6721
Iter: 900, D: 0.1173, G:-0.6976
Iter: 950, D: 0.08645, G:-0.6712
```



```
Iter: 1000, D: 0.1008, G:-0.7126
Iter: 1050, D: 0.08207, G:-0.6909
Iter: 1100, D: 0.05329, G:-0.7102
Iter: 1150, D: 0.07085, G:-0.7307
Iter: 1200, D: 0.05278, G:-0.7284
Iter: 1250, D: 0.05567, G:-0.7188
Iter: 1300, D: 0.0436, G:-0.7196
Iter: 1350, D: 0.05221, G:-0.7221
Iter: 1400, D: 0.04924, G:-0.708
Iter: 1450, D: 0.04604, G:-0.6728
Iter: 1500, D: 0.03481, G:-0.5684
Iter: 1550, D: 0.04124, G:-0.6478
Iter: 1600, D: 0.02364, G:-0.6392
Iter: 1650, D: 0.0337, G:-0.6589
Iter: 1700, D: 0.02415, G:-0.5975
Iter: 1750, D: 0.01834, G:-0.62
Iter: 1800, D: 0.01747, G:-0.5796
Iter: 1850, D: 0.02324, G:-0.6073
Iter: 1900, D: 0.04899, G:-0.7457
Iter: 1950, D: 0.008687, G:-0.6038
```

```
Iter: 2000, D: -0.00565, G:-0.6026
Iter: 2050, D: 0.01727, G:-0.5794
Iter: 2100, D: -0.00662, G:-0.6377
Final images
```