Assignment #3 2
Q4: Style Transfer 2
Q5: Generative Adversarial Networks 7

Assignment #3

Q4: Style Transfer

Style Transfer

In this notebook we will implement the style transfer technique from "Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015).

The general idea is to take two images, and produce a new image that reflects the content of one but the artistic "style" of the other. We will do this by first formulating a loss function that matches the content and style of each respective image in the feature space of a deep network, and then performing gradient descent on the pixels of the image itself.

The deep network we use as a feature extractor is SqueezeNet, a small model that has been trained on ImageNet. You could use any network, but we chose SqueezeNet here for its small size and efficiency.

Here's an example of the images you'll be able to produce by the end of this notebook:

下载图片

```
tf.reset_default_graph() # remove all existing variables in the graph
sess = get_session() # start a new Session

# Load pretrained SqueezeNet model
SAVE_PATH = 'cs231n/datasets/squeezenet.ckpt'
#if not os.path.exists(SAVE_PATH):
#    raise ValueError("You need to download SqueezeNet!")
model = SqueezeNet(save_path=SAVE_PATH, sess=sess)

# Load data for testing
content_img_test = preprocess_image(load_image('styles/tubingen.jpg', si
style_img_test = preprocess_image(load_image('styles/starry_night.jpg',
answers = np.load('style-transfer-checks-tf.npz')
```

```
INFO:tensorflow:Restoring parameters from cs231n/datasets/squeezenet.ck
pt
```

Computing Loss

We're going to compute the three components of our loss function now. The loss function is a weighted sum of three terms: content loss + style loss + total variation loss. You'll fill in the functions that compute these weighted terms below.

Content loss

We can generate an image that reflects the content of one image and the style of another by incorporating both in our loss function. We want to penalize deviations from the content of the content image and deviations from the style of the style image. We can then use this hybrid loss function to perform gradient descent **not on the parameters** of the model, but instead **on the pixel values** of our original image.

Let's first write the content loss function. Content loss measures how much the feature map of the generated image differs from the feature map of the source image. We only care about the content representation of one layer of the network (say, layer ℓ), that has feature maps $A^\ell \in \mathbb{R}^{1\times C_\ell \times H_\ell \times W_\ell}$. C_ℓ is the number of filters/channels in layer ℓ, H_ℓ and W_ℓ are the height and width. We will work with reshaped versions of these feature maps that combine all spatial positions into one dimension. Let $F^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the current image and $P^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the content source image where $M_\ell = H_\ell \times W_\ell$ is the number of elements in each feature map. Each row of F^ℓ or P^ℓ represents the vectorized activations of a particular filter, convolved over all positions of the image. Finally, let w_c be the weight of the content loss term in the loss function.

Then the content loss is given by:

$L_c = w_c \times \sum_{i,j}(F_{ij}^\ell - P_{ij}^\ell)^2$

```
# tf.shape outputs a tensor containing the size of each axis.
shapes = tf.shape(content_current)

F_l = tf.reshape(content_current, [shapes[1], shapes[2]*shapes[3]])
P_l = tf.reshape(content_original,[shapes[1], shapes[2]*shapes[3]])

loss = content_weight * (tf.reduce_sum((F_l - P_l)**2))
```

Style loss

Now we can tackle the style loss. For a given layer ℓ, the style loss is defined as follows:

First, compute the Gram matrix G which represents the correlations between the responses of each filter, where F is as above. The Gram matrix is an approximation to the covariance matrix -- we want the activation statistics of our generated image to match the activation statistics of our style image, and matching the (approximate) covariance is one way to do that. There are a variety of ways you could do this, but the Gram matrix is nice because it's easy to compute and in practice shows good results.

Given a feature map F^ℓ of shape $(1, C_\ell, M_\ell)$, the Gram matrix has shape $(1, C_\ell, C_\ell)$ and its elements are given by:

$$G_{ij}^\ell = \sum_k F_{ik}^\ell F_{jk}^\ell$$

Assuming G^ℓ is the Gram matrix from the feature map of the current image, A^ℓ is the Gram Matrix from the feature map of the source style image, and w_ℓ a scalar weight term, then the style loss for the layer ℓ is simply the weighted Euclidean distance between the two Gram matrices:

$$L_s^\ell = w_\ell \sum_{i,j} \left(G_{ij}^\ell - A_{ij}^\ell\right)^2$$

In practice we usually compute the style loss at a set of layers $\mathcal{L}$ rather than just a single layer ℓ; then the total style loss is the sum of style losses at each layer:

$$L_s = \sum_{\ell \in \mathcal{L}} L_s^\ell$$

Begin by implementing the Gram matrix computation below:

```
    shapes = tf.shape(features)

    # Reshape feature map from [1, H, W, C] to [H*W, C].
    F_l = tf.reshape(features, shape=[shapes[1]*shapes[2],shapes[3]])

    # Gram calculation is just a matrix multiply of F_l and F_l transpose to get [C, C] output shape.
    gram = tf.matmul(tf.transpose(F_l),F_l)

    if normalize == True:
        gram /= tf.cast(shapes[1]*shapes[2]*shapes[3],tf.float32)
```

```
def style_loss(feats, style_layers, style_targets, style_weights):
    """
    Computes the style loss at a set of layers.

    Inputs:
    - feats: list of the features at every layer of the current image, as produced by
      the extract_features function.
    - style_layers: List of layer indices into feats giving the layers to include in the
      style loss.
    - style_targets: List of the same length as style_layers, where style_targets[i] is
      a Tensor giving the Gram matrix the source style image computed at
      layer style_layers[i].
    - style_weights: List of the same length as style_layers, where style_weights[i]
      is a scalar giving the weight for the style loss at layer style_layers[i].

    Returns:
    - style_loss: A Tensor contataining the scalar style loss.
    """
    # Hint: you can do this with one for loop over the style layers, and should
    # not be very much code (~5 lines). You will need to use your gram_matrix function.

    # Initialise style loss to 0.0 (this makes it a float)
    style_loss = tf.constant(0.0)

    # Compute style loss for each desired feature layer and then sum.
    for i in range(len(style_layers)):
        current_im_gram = gram_matrix(feats[style_layers[i]])
        style_loss += style_weights[i] * tf.reduce_sum((current_im_gram - style_targets[i])**2)
```

Total-variation regularization

It turns out that it's helpful to also encourage smoothness in the image. We can do this by adding another term to our loss that penalizes wiggles or "total variation" in the pixel values.

You can compute the "total variation" as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (horizontally or vertically). Here we sum the total-variation regualarization for each of the 3 input channels (RGB), and weight the total summed loss by the total variation weight, w_t:

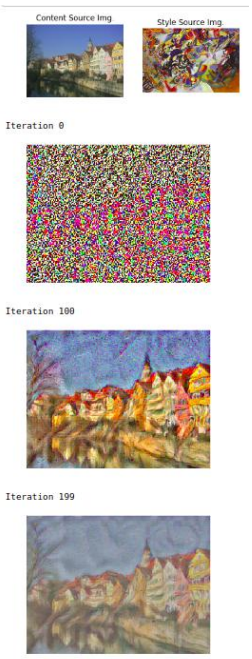
$$L_{tv} = w_t \times \sum_{c=1}^{3} \sum_{i=1}^{H-1} \sum_{j=1}^{W-1} \left((x_{i,j+1,c} - x_{i,j,c})^2 + (x_{i+1,j,c} - x_{i,j,c})^2 \right)$$

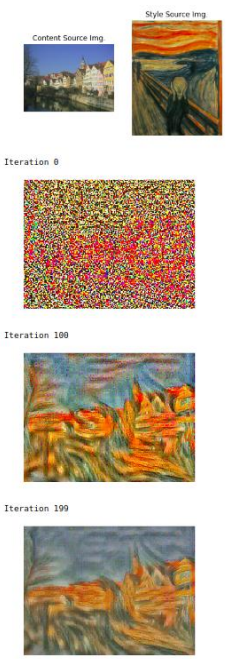
In the next cell, fill in the definition for the TV loss term. To receive full credit, your implementation should not have any loops.

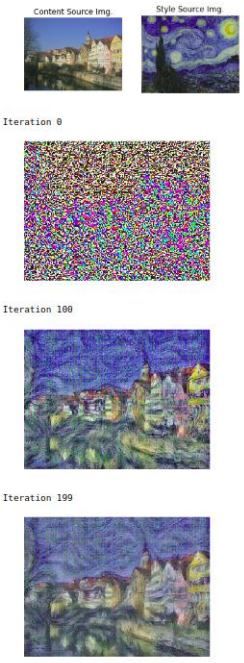
```
w_variance = tf.reduce_sum((img[:,:,1:,:] - img[:,:,:-1,:])**2)
h_variance = tf.reduce_sum((img[:,1:,:,:] - img[:,:-1,:,:])**2)

loss = tv_weight * (w_variance + h_variance)
```

```
def style_transfer(content_image, style_image, image_size, style_size, content_layer, content_weight,
                   style_layers, style_weights, tv_weight, init_random = False):
    """Run style transfer!

    Inputs:
    - content_image: filename of content image
    - style_image: filename of style image
    - image_size: size of smallest image dimension (used for content loss and generated image)
    - style_size: size of smallest style image dimension
    - content_layer: layer to use for content loss
    - content_weight: weighting on content loss
    - style_layers: list of layers to use for style loss
    - style_weights: list of weights to use for each layer in style_layers
    - tv_weight: weight of total variation regularization term
    - init_random: initialize the starting image to uniform random noise
    """
```

```python
# Extract features from the content image
content_img = preprocess_image(load_image(content_image, size=image_size))
feats = model.extract_features(model.image)
content_target = sess.run(feats[content_layer],
                          {model.image: content_img[None]})

# Extract features from the style image
style_img = preprocess_image(load_image(style_image, size=style_size))
style_feat_vars = [feats[idx] for idx in style_layers]
style_target_vars = []
# Compute list of TensorFlow Gram matrices
for style_feat_var in style_feat_vars:
    style_target_vars.append(gram_matrix(style_feat_var))
# Compute list of NumPy Gram matrices by evaluating the TensorFlow graph on the style image
style_targets = sess.run(style_target_vars, {model.image: style_img[None]})

# Initialize generated image to content image

if init_random:
    img_var = tf.Variable(tf.random_uniform(content_img[None].shape, 0, 1), name="image")
else:
    img_var = tf.Variable(content_img[None], name="image")

# Extract features on generated image
feats = model.extract_features(img_var)
# Compute loss
c_loss = content_loss(content_weight, feats[content_layer], content_target)
s_loss = style_loss(feats, style_layers, style_targets, style_weights)
t_loss = tv_loss(img_var, tv_weight)
loss = c_loss + s_loss + t_loss

# Set up optimization hyperparameters
initial_lr = 3.0
decayed_lr = 0.1
decay_lr_at = 180
max_iter = 200

# Create and initialize the Adam optimizer
lr_var = tf.Variable(initial_lr, name="lr")
# Create train_op that updates the generated image when run
with tf.variable_scope("optimizer") as opt_scope:
    train_op = tf.train.AdamOptimizer(lr_var).minimize(loss, var_list=[img_var])
# Initialize the generated image and optimization variables
opt_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope=opt_scope.name)
sess.run(tf.variables_initializer([lr_var, img_var] + opt_vars))
# Create an op that will clamp the image values when run
clamp_image_op = tf.assign(img_var, tf.clip_by_value(img_var, -1.5, 1.5))
```







```
f, axarr = plt.subplots(1,2)
axarr[0].axis('off')
axarr[1].axis('off')
axarr[0].set_title('Content Source Img.')
axarr[1].set_title('Style Source Img.')
axarr[0].imshow(deprocess_image(content_img))
axarr[1].imshow(deprocess_image(style_img))
plt.show()
plt.figure()

# Hardcoded handcrafted
for t in range(max_iter):
    # Take an optimization step to update img_var
    sess.run(train_op)
    if t < decay_lr_at:
        sess.run(clamp_image_op)
    if t == decay_lr_at:
        sess.run(tf.assign(lr_var, decayed_lr))
    if t % 100 == 0:
        print('Iteration {}'.format(t))
        img = sess.run(img_var)
        plt.imshow(deprocess_image(img[0], rescale=True))
        plt.axis('off')
        plt.show()
print('Iteration {}'.format(t))
img = sess.run(img_var)
plt.imshow(deprocess_image(img[0], rescale=True))
plt.axis('off')
plt.show()
```

Feature Inversion

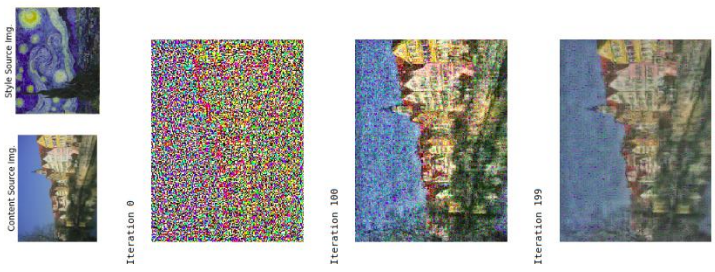
The code you've written can do another cool thing. In an attempt to understand the types of features that convolutional networks learn to recognize, a recent paper [1] attempts to reconstruct an image from its feature representation. We can easily implement this idea using image gradients from the pretrained network, which is exactly what we did above (but with two different feature representations).

Now, if you set the style weights to all be 0 and initialize the starting image to random noise instead of the content source image, you'll reconstruct an image from the feature representation of the content source image. You're starting with total noise, but you should end up with something that looks quite a bit like your original image.

(Similarly, you could do "texture synthesis" from scratch if you set the content weight to 0 and initialize the starting image to random noise, but we won't ask you to do that here.)

[1] Aravindh Mahendran, Andrea Vedaldi, "Understanding Deep Image Representations by Inverting them", CVPR 2015

```
# Feature Inversion -- Starry Night + Tubingen
params_inv = {
    'content_image' : 'styles/tubingen.jpg',
    'style_image' : 'styles/starry_night.jpg',
    'image_size' : 192,
    'style_size' : 192,
    'content_layer' : 3,
    'content_weight' : 6e-2,
    'style_layers' : [1, 4, 6, 7],
    'style_weights' : [0, 0, 0, 0], # we discard any contributions from style to the loss
    'tv_weight' : 2e-2,
    'init_random': True # we want to initialize our image to be random
}

style_transfer(**params_inv)
```



Q5: Generative Adversarial Networks

Generative Adversarial Networks (GANs)

So far in CS231N, all the applications of neural networks that we have explored have been **discriminative models** that take an input and are trained to produce a labeled output. This has ranged from straightforward classification of image categories to sentence generation (which was still phrased as a classification problem, our labels were in vocabulary space and we'd learned a recurrence to capture multi-word labels). In this notebook, we will expand our repetoire, and build **generative models** using neural networks. Specifically, we will learn how to build models which generate novel images that resemble a set of training images.

What is a GAN?

In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D), and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

where $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In Goodfellow et al., they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

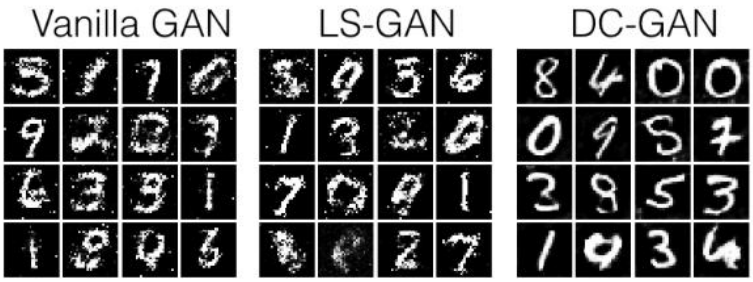
To optimize this minimax game, we will aternate between taking gradient *descent* steps on the objective for G, and gradient *ascent* steps on the objective for D:

1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**.
2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers, and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates:

1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:
$$\underset{G}{\text{maximize}}\ \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:
$$\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$



下载 MNIST 数据集

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('./cs231n/datasets/MNIST_data', one_hot=False)
```

LeakyReLU

In the cell below, you should implement a LeakyReLU. See the class notes (where alpha is small number) or equation (3) in this paper. LeakyReLUs keep ReLU units from dying and are often used in GAN methods (as are maxout units, however those increase model size and therefore are not used in this notebook).

activation = tf.maximum(x,alpha*x)

```
# TODO: sample and return noise
random_noise = tf.random_uniform(maxval=1,minval=-1,shape=[batch_size, dim])
```

Discriminator

Our first step is to build a discriminator. You should use the layers in `tf.layers` to build the model. All fully connected layers should include bias terms.

Architecture:

- Fully connected layer from size 784 to 256
- LeakyReLU with alpha 0.01
- Fully connected layer from 256 to 256
- LeakyReLU with alpha 0.01
- Fully connected layer from 256 to 1

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

```
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(inputs=x, units=256, activation=leaky_relu)
        fc2 = tf.layers.dense(inputs=fc1, units=256, activation=leaky_relu)
        logits = tf.layers.dense(inputs=fc2, units=1)

        return logits
```

Generator

Now to build a generator. You should use the layers in `tf.layers` to construct the model. All fully connected layers should include bias terms.

Architecture:

- Fully connected layer from tf.shape(z)[1] (the number of noise dimensions) to 1024
- ReLU
- Fully connected layer from 1024 to 1024
- ReLU
- Fully connected layer from 1024 to 784
- TanH (To restrict the output to be [-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(inputs=z, units=1024, activation=tf.nn.relu)
        fc2 = tf.layers.dense(inputs=fc1, units=1024, activation=tf.nn.relu)
        img = tf.layers.dense(inputs=fc2, units=784, activation=tf.nn.tanh)

        return img
```

GAN Loss

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x\sim p_{data}}\left[\log D(x)\right] - \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

```
# Target label vector for generator loss and used in discriminator loss.
true_labels = tf.ones_like(logits_fake)

# DISCRIMINATOR loss has 2 parts: how well it classifies real images and how well it
# classifies fake images.
real_image_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=logits_real, labels=true_labels)
fake_image_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=logits_fake, labels=1-true_labels)

# Combine and average losses over the batch
D_loss = real_image_loss + fake_image_loss
D_loss = tf.reduce_mean(D_loss)

# GENERATOR is trying to make the discriminator output 1 for all its images.
# So we use our target label vector of ones for computing generator loss.
G_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=logits_fake, labels=true_labels)

# Average generator loss over the batch.
G_loss = tf.reduce_mean(G_loss)
```

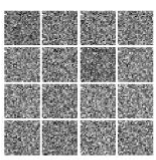
Optimizing our loss

Make an `AdamOptimizer` with a 1e-3 learning rate, beta1=0.5 to mininize G_loss and D_loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge in the Improved Techniques for Training GANs paper. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there's a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your D(x) learns to be too fast (e.g. loss goes near zero), your G(z) is never able to learn. Often D(x) is trained with SGD with Momentum or RMSProp instead of Adam, but here we'll use Adam for both D(x) and G(z).

```
D_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1)
G_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1)
```

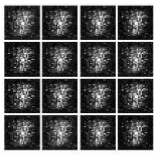
```
tf.reset_default_graph()

# 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')
```

```
    # 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()
```

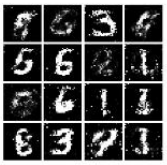


```
Iter: 0, D: 0.9408, G:0.7071
Iter: 50, D: 0.2111, G:1.9
Iter: 100, D: 1.025, G:1.472
Iter: 150, D: 1.149, G:1.521
Iter: 200, D: 1.275, G:1.281
```

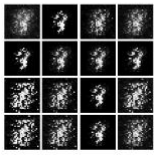
```
Iter: 4000, D: 1.326, G:0.763
Iter: 4050, D: 1.323, G:0.917
Iter: 4100, D: 1.402, G:0.898
Iter: 4150, D: 1.272, G:0.801
Iter: 4200, D: 1.306, G:0.789
```

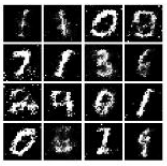


```
Iter: 250, D: 1.772, G:0.8271
Iter: 300, D: 1.827, G:1.049
Iter: 350, D: 1.047, G:1.162
Iter: 400, D: 1.436, G:2.082
Iter: 450, D: 1.739, G:1.702
```



```
Iter: 4250, D: 1.284, G:0.872
Final images
```





Least Squares GAN

We'll now look at Least Squares GAN, a newer, more stable alternative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

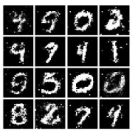
$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z))-1)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[(D(x)-1)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)))^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`score_real` and `score_fake`).

```
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
"""

true_labels = tf.ones_like(score_fake)
fake_image_loss = tf.reduce_mean((score_real - true_labels)**2)
real_image_loss = tf.reduce_mean(score_fake**2)
D_loss = 0.5*(fake_image_loss + real_image_loss)

G_loss = 0.5*tf.reduce_mean((score_fake - true_labels)**2)
```



```
Iter: 4250, D: 0.2128, G:0.1872
Final images
```

改进模型：

1.Deep Convolutional GANs

Deep Convolutional GANs

In the first part of the notebook, we implemented an almost direct copy of the original GAN network from Ian Goodfellow. However, this network architecture allows no real spatial reasoning. It is unable to reason about things like "sharp edges" in general because it lacks any convolutional layers. Thus, in this section, we will implement some of the ideas from DCGAN, where we use convolutional networks as our discriminators and generators.

Discriminator

We will use a discriminator inspired by the TensorFlow MNIST classification tutorial, which is able to get above 99% accuracy on the MNIST dataset fairly quickly. *Be sure to check the dimensions of x and reshape when needed*, fully connected blocks expect [N,D] Tensors while conv2d blocks expect [N,H,W,C] Tensors.

Architecture:

- 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

Generator

For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for tf.nn.conv2d_transpose. We are always "training" in GAN mode.

Architecture:

- 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

2. WGAN-GP

WGAN-GP (Small Extra Credit)

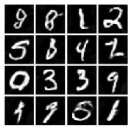
Please only attempt after you have completed everything above.

We'll now look at Improved Wasserstein GAN as a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement Algorithm 1 in the paper.

You'll also need to use a discriminator and corresponding generator without max-pooling. So we cannot use the one we currently have from DCGAN. Pair the DCGAN Generator (from InfoGAN) with the discriminator from InfoGAN Appendix C.1 MNIST (We don't use Q, simply implement the network up to D). You're also welcome to define a new generator and discriminator in this notebook, in case you want to use the fully-connected pair of D(x) and G(z) you used at the top of this notebook.

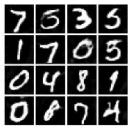
Architecture:

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



```
Iter: 1000, D: 1.219, G:1.209
Iter: 1050, D: 1.214, G:1.203
Iter: 1100, D: 1.217, G:0.965
Iter: 1150, D: 1.177, G:0.869
Iter: 1200, D: 1.425, G:0.692
```

```
Iter: 1250, D: 1.198, G:1.009
Final images
```



```
Iter: 1250, D: -0.1453, G:-21.1
Iter: 1300, D: -0.2719, G:-0.6963
Iter: 1350, D: -1.588, G:-1.248
Iter: 1400, D: 0.2692, G:-6.189
Iter: 1450, D: -0.447, G:-10.51
```

```
Iter: 1500, D: 0.001418, G:-11.09
Iter: 1550, D: -0.3884, G:-1.438
Iter: 1600, D: -0.8741, G:7.658
Iter: 1650, D: 0.313, G:-1.547
Iter: 1700, D: -1.022, G:-3.035
```

```
Iter: 1750, D: -0.884, G:5.057
Iter: 1800, D: -0.6864, G:11.55
Iter: 1850, D: -0.5427, G:-4.58
```