Face Generation¶

In this project, you'll use generative adversarial networks to generate new images of faces.

Get the Data

You'll be using two datasets in this project:

- MNIST
- CelebA

Since the celebA dataset is complex and you're doing GANs in a project for the first time, we want you to test your neural network on MNIST before CelebA. Running the GANs on MNIST will allow you to see how well your model trains sooner.

If you're using FloydHub, set `data_dir` to "/input" and use the FloydHub data ID "R5KrjnANiKVhLWAkpXhNBe".

In [1]:

```
data_dir = './data'

# FloydHub - Use with data ID "R5KrjnANiKVhLWAkpXhNBe"
# data_dir = '/input'


"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import helper

helper.download_extract('mnist', data_dir)
helper.download_extract('celeba', data_dir)
```

```
Found mnist Data
Found celeba Data
```

Explore the Data

MNIST

As you're aware, the MNIST dataset contains images of handwritten digits. You can view the first number of examples by changing `show_n_images`.

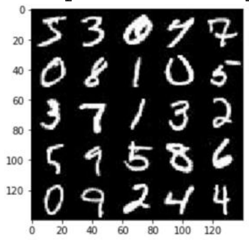
In [2]:

```
show_n_images = 25

"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
%matplotlib inline
import os
from glob import glob
from matplotlib import pyplot
```

```
mnist_images = helper.get_batch(glob(os.path.join(data_dir,
'mnist/*.jpg'))[:show_n_images], 28, 28, 'L')
pyplot.imshow(helper.images_square_grid(mnist_images, 'L'), cmap='gray')
```

Out[2]:

```
<matplotlib.image.AxesImage at 0x1857f3756d8>
```



CelebA

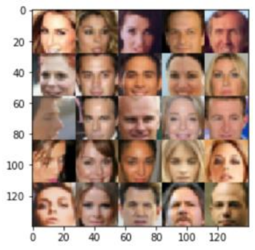
The CelebFaces Attributes Dataset (CelebA) dataset contains over 200,000 celebrity images with annotations. Since you're going to be generating faces, you won't need the annotations. You can view the first number of examples by changing `show_n_images`.

In [3]:

```
show_n_images = 25

"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
mnist_images = helper.get_batch(glob(os.path.join(data_dir,
'img_align_celeba/*.jpg'))[:show_n_images], 28, 28, 'RGB')
pyplot.imshow(helper.images_square_grid(mnist_images, 'RGB'))
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x1850003be48>
```



Preprocess the Data

Since the project's main focus is on building the GANs, we'll preprocess the data for you. The values of the MNIST and CelebA dataset will be in the range of -0.5 to 0.5 of 28x28 dimensional images. The CelebA images will be cropped to remove parts of the image that don't include a face, then resized down to 28x28.

The MNIST images are black and white images with a single color channel while the CelebA images have 3 color channels (RGB color channel).

Build the Neural Network

You'll build the components necessary to build a GANs by implementing the following functions below:

- `model_inputs`
- `discriminator`
- `generator`
- `model_loss`
- `model_opt`
- `train`

Check the Version of TensorFlow and Access to GPU

This will check to make sure you have the correct version of TensorFlow and access to a GPU

In [4]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
from distutils.version import LooseVersion
import warnings
import tensorflow as tf

# Check TensorFlow Version
assert LooseVersion(tf.__version__) >= LooseVersion('1.0'), 'Please use 
TensorFlow version 1.0 or newer.  You are using {}'.format(tf.__version__)
print('TensorFlow Version: {}'.format(tf.__version__))

# Check for a GPU
if not tf.test.gpu_device_name():
    warnings.warn('No GPU found. Please use a GPU to train your neural 
network.')
else:
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
```

```
TensorFlow Version: 1.1.0
c:\users\vadymserpak\anaconda3\envs\python35\lib\site-
packages\ipykernel_launcher.py:14: UserWarning: No GPU found. Please use a GPU
to train your neural network.
```

Input

Implement the `model_inputs` function to create TF Placeholders for the Neural Network. It should create the following placeholders:

- Real input images placeholder with rank 4 using `image_width`, `image_height`, and `image_channels`.
- Z input placeholder with rank 2 using `z_dim`.
- Learning rate placeholder with rank 0.

Return the placeholders in the following the tuple (tensor of real input images, tensor of z data)

In [5]:

```
import problem_unittests as tests

def model_inputs(image_width, image_height, image_channels, z_dim):
    """
    Create the model inputs
    :param image_width: The input image width
    :param image_height: The input image height
    :param image_channels: The number of image channels
    :param z_dim: The dimension of Z
    :return: Tuple of (tensor of real input images, tensor of z data, learning 
rate)
    """
    # TODO: Implement Function
    input_real = tf.placeholder(tf.float32, shape=(None, image_height,
image_width, image_channels))
    input_z = tf.placeholder(tf.float32, shape=(None, z_dim))
    learning_rate = tf.placeholder(tf.float32, shape=())

    return input_real, input_z, learning_rate
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_model_inputs(model_inputs)
```

```
Tests Passed
```

Discriminator

Implement `discriminator` to create a discriminator neural network that discriminates on `images`. This function should be able to reuse the variabes in the neural network. Use `tf.variable_scope` with a scope name of "discriminator" to allow the variables to be reused. The function should return a tuple of (tensor output of the generator, tensor logits of the generator).

In [7]:

```
def discriminator(images, reuse=False, alpha=0.1):
    """
    Create the discriminator network
    :param image: Tensor of input image(s)
    :param reuse: Boolean if the weights should be reused
    :return: Tuple of (tensor output of the discriminator, tensor logits of the
discriminator)
    """
    # TODO: Implement Function
    ker_init = tf.random_normal_initializer(stddev=0.02)

    with tf.variable_scope('discriminator', reuse=reuse):
        # Input is 28x28x3
        x = tf.layers.conv2d(images, 64, 5, strides=2,
kernel_initializer=ker_init, padding='same')
        x = tf.maximum(x * alpha, x)
        # 14x14x64

        x = tf.layers.conv2d(x, 128, 5, strides=2, kernel_initializer=ker_init,
padding='same')
        x = tf.maximum(x * alpha, x)
        x = tf.layers.batch_normalization(x, training=True)
        # 7x7x128

        x = tf.layers.conv2d(x, 256, 5, strides=2, kernel_initializer=ker_init,
padding='same')
        x = tf.maximum(x * alpha, x)
        x = tf.layers.batch_normalization(x, training=True)
        # 4x4x256

        x = tf.reshape(x, (-1, 4*4*256))
        logits = tf.layers.dense(x, 1)
        out = tf.sigmoid(logits)
    return out, logits
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_discriminator(discriminator, tf)
```

```
Tests Passed
```

Generator

Implement `generator` to generate an image using `z`. This function should be able to reuse the variabes in the neural network. Use `tf.variable_scope` with a scope name of "generator" to allow the variables to be reused. The function should return the generated 28 x 28 x `out_channel_dim` images.

In [8]:

```
def generator(z, out_channel_dim, is_train=True, alpha=0.1):
    """
    Create the generator network
    :param z: Input z
    :param out_channel_dim: The number of channels in the output image
    :param is_train: Boolean if generator is being used for training
    :return: The tensor output of the generator
    """
    # TODO: Implement Function

    ker_init = tf.random_normal_initializer(stddev=0.02)

    with tf.variable_scope('generator', reuse=(not is_train)):
        x = tf.layers.dense(z, 7*7*256)
        x = tf.reshape(x, (-1, 7, 7, 256))
        x = tf.maximum(x * alpha, x)
        x = tf.layers.batch_normalization(x, training=is_train)
        # 7x7x256

        x = tf.layers.conv2d_transpose(x, 128, 5, strides=2,
kernel_initializer=ker_init, padding='same')
        x = tf.maximum(x * alpha, x)
        x = tf.layers.batch_normalization(x, training=is_train)
        # 14x14x128

        x = tf.layers.conv2d_transpose(x, 64, 5, strides=2,
kernel_initializer=ker_init, padding='same')
        x = tf.maximum(x * alpha, x)
        x = tf.layers.batch_normalization(x, training=is_train)
        # 28x28x64

        logits = tf.layers.conv2d_transpose(x, out_channel_dim, 3, strides=1,
kernel_initializer=ker_init, padding='same')
        # 28x28x3

        out = tf.tanh(logits)

    return out
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_generator(generator, tf)
```

```
Tests Passed
```

Loss

Implement `model_loss` to build the GANs for training and calculate the loss. The function should return a tuple of (discriminator loss, generator loss). Use the following functions you implemented:

- `discriminator(images, reuse=False)`
- `generator(z, out_channel_dim, is_train=True)`

In [9]:

```
def model_loss(input_real, input_z, out_channel_dim, alpha=0.1):
    """
    Get the loss for the discriminator and generator
    :param input_real: Images from the real dataset
    :param input_z: Z input
    :param out_channel_dim: The number of channels in the output image
    :return: A tuple of (discriminator loss, generator loss)
    """
    # TODO: Implement Function
    g_model = generator(input_z, out_channel_dim, is_train=True, alpha=alpha)
    d_real_out, d_real_logits = discriminator(input_real, alpha=alpha)
    d_fake_out, d_fake_logits = discriminator(g_model, reuse=True, alpha=alpha)

    smooth = 0.1
    real_labels = tf.ones_like(d_real_out) * (1 - smooth)
    fake_labels = tf.zeros_like(d_fake_out)
    g_labels    = tf.ones_like(d_fake_out)

    d_real_loss = 
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_real_logits, 
labels=real_labels))
    d_fake_loss = 
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_fake_logits, 
labels=fake_labels))
    g_loss = 
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_fake_logits, 
labels=g_labels))

    d_loss = d_real_loss + d_fake_loss
    return d_loss, g_loss


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_model_loss(model_loss)
```

```
Tests Passed
```

Optimization

Implement `model_opt` to create the optimization operations for the GANs.
Use `tf.trainable_variables` to get all the trainable variables. Filter the variables with names that are in the discriminator and generator scope names. The function should return a tuple of (discriminator training operation, generator training operation).

In [10]:

```
def model_opt(d_loss, g_loss, learning_rate, beta1):
    """
    Get optimization operations
    :param d_loss: Discriminator loss Tensor
    :param g_loss: Generator loss Tensor
    :param learning_rate: Learning Rate Placeholder
    :param beta1: The exponential decay rate for the 1st moment in the 
optimizer
    :return: A tuple of (discriminator training operation, generator training 
operation)
    """
    # TODO: Implement Function
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    d_updates = [opt for opt in update_ops if 
opt.name.startswith('discriminator')]
    g_updates = [opt for opt in update_ops if opt.name.startswith('generator')]

    with tf.control_dependencies(d_updates):
        d_opt = tf.train.AdamOptimizer(learning_rate=learning_rate, 
beta1=beta1).minimize(d_loss, var_list=d_vars)

    with tf.control_dependencies(g_updates):
        g_opt = tf.train.AdamOptimizer(learning_rate=learning_rate, 
beta1=beta1).minimize(g_loss, var_list=g_vars)

    return d_opt, g_opt


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_model_opt(model_opt, tf)
```

```
Tests Passed
```

Neural Network Training

Show Output

Use this function to show the current output of the generator during training. It will help you determine how well the GANs is training.

In [11]:

```
"""
```

```
DON'T MODIFY ANYTHING IN THIS CELL
"""
import numpy as np

def show_generator_output(sess, n_images, input_z, out_channel_dim,
image_mode):
    """
    Show example output for the generator
    :param sess: TensorFlow session
    :param n_images: Number of Images to display
    :param input_z: Input Z Tensor
    :param out_channel_dim: The number of channels in the output image
    :param image_mode: The mode to use for images ("RGB" or "L")
    """
    cmap = None if image_mode == 'RGB' else 'gray'
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, is_train=False),
        feed_dict={input_z: example_z})

    images_grid = helper.images_square_grid(samples, image_mode)
    pyplot.imshow(images_grid, cmap=cmap)
    pyplot.show()
```

Train

Implement `train` to build and train the GANs. Use the following functions you implemented:

- `model_inputs(image_width, image_height, image_channels, z_dim)`
- `model_loss(input_real, input_z, out_channel_dim)`
- `model_opt(d_loss, g_loss, learning_rate, beta1)`

Use the `show_generator_output` to show `generator` output while you train.
Running `show_generator_output` for every batch will drastically increase training time and increase the size of the notebook. It's recommended to print the `generator`output every 100 batches.

In [13]:

```
import time

def train(epoch_count, batch_size, z_dim, learning_rate, beta1,
          get_batches, data_shape, data_image_mode):
    """
    Train the GAN
    :param epoch_count: Number of epochs
    :param batch_size: Batch Size
    :param z_dim: Z dimension
    :param learning_rate: Learning Rate
    :param beta1: The exponential decay rate for the 1st moment in the
optimizer
    :param get_batches: Function to get batches
    :param data_shape: Shape of the data
    :param data_image_mode: The image mode to use for images ("RGB" or "L")
    """
    # TODO: Build Model
    print_every = 25
    out_channel_dim = len(data_image_mode)

    input_real, input_z, learn_r = model_inputs(data_shape[1], data_shape[2], 
out_channel_dim, z_dim)
    d_loss, g_loss = model_loss(input_real, input_z, out_channel_dim)
    d_opt, g_opt = model_opt(d_loss, g_loss, learn_r, beta1)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for epoch_i in range(epoch_count):
            for batch_count, batch_images in 
enumerate(get_batches(batch_size)):
                batch_z = np.random.uniform(-1, 1, size=(batch_size, z_dim))
                batch_images *= 2.0

                _ = sess.run(g_opt, feed_dict={input_z: batch_z, 
learn_r:learning_rate})
                _ = sess.run(d_opt, feed_dict={input_real: batch_images, 
input_z: batch_z, learn_r:learning_rate})
                _ = sess.run(g_opt, feed_dict={input_z: batch_z, 
learn_r:learning_rate})

                if (batch_count % print_every) == 0:
                    train_loss_d = d_loss.eval({input_z: batch_z, input_real: 
batch_images})
                    train_loss_g = g_loss.eval({input_z: batch_z})
                    if (batch_count % (print_every*10)) == 0:
                        show_generator_output(sess, 25, input_z, 
out_channel_dim, data_image_mode)

                    print("Time {}... Epoch {}/{}... Batch 
{}...".format(time.strftime('%X'), epoch_i+1, epoch_count, batch_count),
                          "Discriminator Loss: {:.4f}...".format(train_loss_d),
                          "Generator Loss: {:.4f}".format(train_loss_g))


        show_generator_output(sess, 25, input_z, out_channel_dim, 
data_image_mode)
```

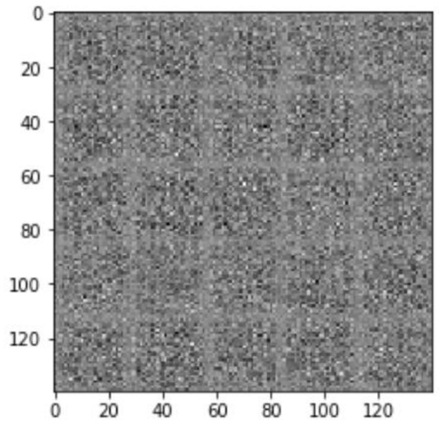
MNIST

Test your GANs architecture on MNIST. After 2 epochs, the GANs should be able to generate images that look like handwritten digits. Make sure the loss of the generator is lower than the loss of the discriminator or close to 0.

In [14]:

```
batch_size = 64
z_dim = 100
learning_rate = 0.0001
beta1 = 0.5
```

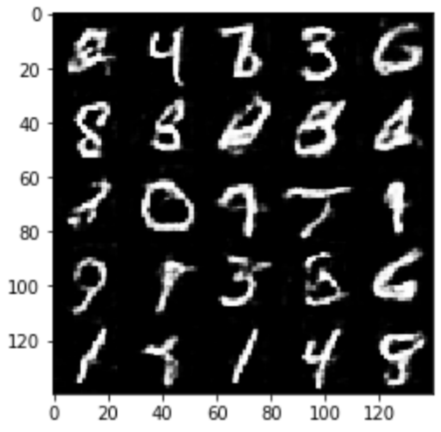
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
epochs = 2

mnist_dataset = helper.Dataset('mnist', glob(os.path.join(data_dir,
'mnist/*.jpg')))
with tf.Graph().as_default():
    train(epochs, batch_size, z_dim, learning_rate, beta1,
mnist_dataset.get_batches,
          mnist_dataset.shape, mnist_dataset.image_mode)
```



```
Time 15:05:55... Epoch 1/2... Batch 0... Discriminator Loss: 4.9102... Generator Loss: 0.0115
```

……..

```
Time 17:57:49... Epoch 2/2... Batch 925... Discriminator Loss: 1.1493... Generator Loss: 0.9874
```



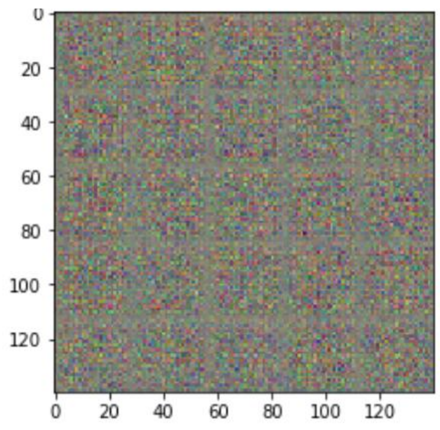
CelebA

Run your GANs on CelebA. It will take around 20 minutes on the average GPU to run one epoch. You can run the whole epoch or stop when it starts to generate realistic faces.

In [15]:

```
batch_size = 64
z_dim = 100
learning_rate = 0.0002
beta1 = 0.5
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
epochs = 1

celeba_dataset = helper.Dataset('celeba', glob(os.path.join(data_dir,
'img_align_celeba/*.jpg')))
with tf.Graph().as_default():
    train(epochs, batch_size, z_dim, learning_rate, beta1,
celeba_dataset.get_batches,
          celeba_dataset.shape, celeba_dataset.image_mode)
```



```
 Time 18:13:49... Epoch 1/1... Batch 0... Discriminator Loss: 7.9274... Generator Loss: 0.0014
 Time 18:15:57... Epoch 1/1... Batch 25... Discriminator Loss: 1.0568... Generator Loss: 0.9701
```

……..

```
Time 23:31:48... Epoch 1/1... Batch 3150... Discriminator Loss: 1.4360... Generator Loss: 0.6966
```