dlnd_face_generation

May 29, 2018

1 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)
```

```
Downloading mnist: 9.92MB [00:01, 5.99MB/s]
Extracting mnist: 100%|| 60.0K/60.0K [00:08<00:00, 6.90KFile/s]
Downloading celeba: 1.44GB [02:20, 10.3MB/s]


Extracting celeba...
```

1.1 Explore the Data

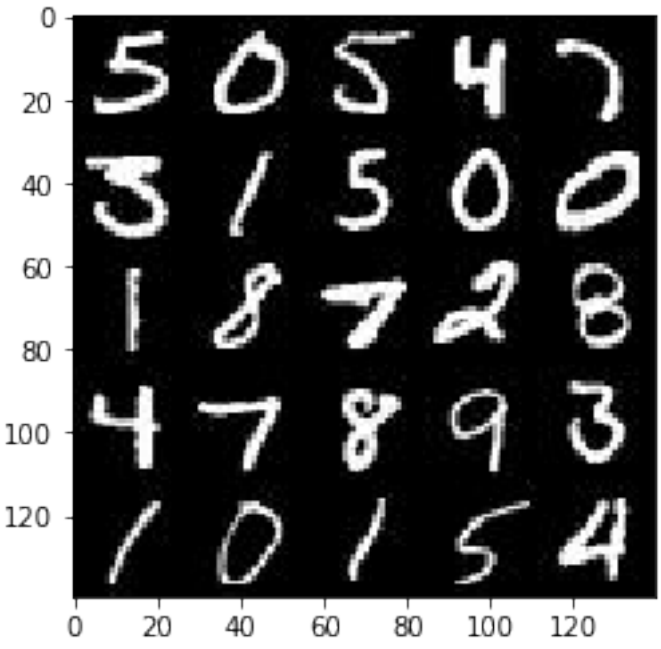
1.1.1 MNIST

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_jpegs  = glob(os.path.join(data_dir, 'mnist/*.jpg'))
mnist_images = helper.get_batch(mnist_jpegs[:show_n_images], 28, 28, 'L')

pyplot.imshow(helper.images_square_grid(mnist_images, 'L'))
# helper.images_square_grid had to be adapted, i.e. convert image to RGBA,
# cmap = 'gray' unnecessary now.
# see also: https://github.com/matplotlib/matplotlib/issues/10616/
```

```
/usr/local/lib/python3.5/dist-packages/matplotlib/font_manager.py:279: UserWarning: Matplotlib i
  'Matplotlib is building the font cache using fc-list. '
```

Out[2]: <matplotlib.image.AxesImage at 0x7f44eec739e8>

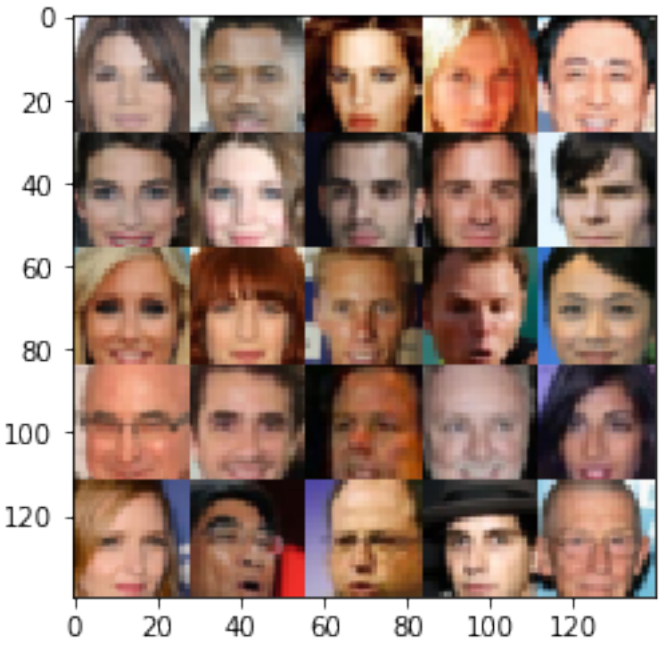


1.1.2 CelebA

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
"""
celeba_jpegs = glob(os.path.join(data_dir, 'img_align_celeba/*.jpg'))
mnist_images = helper.get_batch(celeba_jpegs[:show_n_images], 28, 28, 'RGB')
pyplot.imshow(helper.images_square_grid(mnist_images, 'RGB'))
```

Out[3]:
```
<matplotlib.image.AxesImage at 0x7f44e6aea940>
```



1.2 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](https://en.wikipedia.org/wiki/Channel_(digital_image%29) while the CelebA images have [3 color channels (RGB color channel)](https://en.wikipedia.org/wiki/Channel_(digital_image%29#RGB_Images). ## 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`

1.2.1 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: 1.4.1
Default GPU Device: /device:GPU:0
```

1.2.2 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
    imreal = tf.placeholder(tf.float32, shape = (None, image_width,
                                                 image_height, image_channels))
    z      = tf.placeholder(tf.float32, shape = (None, z_dim))
    lr     = tf.placeholder(tf.float32)
    return imreal, z, lr

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

```
Tests Passed
```

1.2.3 Discriminator

Implement `discriminator` to create a discriminator neural network that discriminates on `images`. This function should be able to reuse the variables 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 discriminator, tensor logits of the discriminator).

```
In [6]: def discriminator(images, reuse=False):
    """
    Create the discriminator network
    : param images : 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)
    """
    def convolution(in_, f, k = (5, 5), s = (2, 2), p = 'SAME',
                    activation = None, use_bias = False):
        return tf.layers.conv2d(in_, filters = f,
                                kernel_size = k, strides = s,
                                padding = p, data_format = 'channels_last',
                                activation = activation, use_bias = use_bias)

    def fullyconnected(in_,num_units = 1, use_bias = False):
        return tf.layers.dense(in_, units = num_units,
                               activation = None, use_bias = use_bias)

    def batchnorm(in_):
        return tf.layers.batch_normalization(in_, training = True)
    #training = True, as discriminator not used for inference

    def activation(in_, alpha=0.02, kind='leakyReLU'):
        if kind == 'leakyReLU':
```

```
                return tf.maximum(alpha*in_, in_)
            elif kind == 'sigmoid':
                return tf.sigmoid(in_)

        # TODO: Implement Function
        with tf.variable_scope("discriminator", reuse = reuse):
            layers         = 3
            layer          = images
            list_layers    = list(range(layers))
            for i in list_layers:
                if i != list_layers[-1]:
                    layer  = convolution(layer, f=2*(i+1)*int(images.shape[1]))
                    if i != 0:
                        layer = batchnorm(layer)
                    layer  = activation(layer)
                else:
                    reshape_dims = layer.shape[1]*layer.shape[2]*layer.shape[3]
                    layer        = tf.reshape(layer, shape = (-1, reshape_dims))
                    logits       = fullyconnected(layer)
                    out          = activation(logits, kind = 'sigmoid')

        return logits, out

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

```
Tests Passed
```

1.2.4 Generator

Implement `generator` to generate an image using `z`. This function should be able to reuse the variables 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 [22]: def generator(z, out_channel_dim, is_train=True):
             """
             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
             """
             def fullyconnected(in_, num_units):
                 return tf.layers.dense(in_, units = num_units, activation = None,
```

```
                                use_bias = False)

def transposed_convolution(in_, f, k = (5, 5), s = (2, 2), p = 'SAME',
                                activation = None, use_bias = False):
    return tf.layers.conv2d_transpose(in_, filters = f,
                                kernel_size = k, strides = s,
                                padding = p, data_format = 'channels_last',
                                activation = activation, use_bias = use_bias)


def batchnorm(in_):
    return tf.layers.batch_normalization(in_, training = is_train)
#training = True, as discriminator not used for inference

def leakyReLU(in_, alpha=0.02):
    return tf.maximum(alpha*in_, in_)

def tanh(in_):
    return tf.tanh(in_)

# TODO: Implement Function
reuse = False if is_train else True
with tf.variable_scope("generator", reuse = reuse):
    ''' --- APPARENTLY TOO COMPLICATED ---
    in_dim = (-1, 3, 3, 256)
    layer = fullyconnected(z, num_units = in_dim[1]*in_dim[2]*in_dim[3])

    # VALID padding and stride = 1 to generate more layers
    # (3,3,256) -> (5, 5, 128) - kernel size k = 3

    layer = tf.reshape(layer, shape = in_dim)
    layer = transposed_convolution(layer, f = 128, s = (1, 1), k = (3, 3), p = 'VAL

    layer = batchnorm(layer)
    layer = leakyReLU(layer)

    # (5, 5, 64) -> (7, 7, 32) - k = 3
    layer = transposed_convolution(layer, f = 64, s = (1, 1), k = (3, 3), p = 'VALI

    layer = batchnorm(layer)
    layer = leakyReLU(layer)

    # (7,7,64) -> (10, 10, 32)  - k = 4
    layer = transposed_convolution(layer, f = 32, s = (1, 1), k = (4, 4), p = 'VALI

    layer = batchnorm(layer)
    layer = leakyReLU(layer)
```

```
        # (10, 10, 32) -> (14, 14, 16) - k = 5
        layer = transposed_convolution(layer, f = 16, s = (1, 1), p = 'VALID')
        #layer = transposed_convolution(layer, f = 16)

        layer = batchnorm(layer)
        layer = leakyReLU(layer)

        # (14, 14, 16) -> (21, 21, 8) - k = 8
        layer = transposed_convolution(layer, f = 8, s = (1, 1), k = (8, 8), p = 'VALID

        layer = batchnorm(layer)
        layer = leakyReLU(layer)


        # (21, 21, 9) -> (28,28,3)  - k = 8
        layer = transposed_convolution(layer, f = out_channel_dim, s = (1, 1), k = (8,
        #layer = transposed_convolution(layer, f = out_channel_dim)

        # layer = batchnorm(layer)
        layer = tanh(layer)/2
        # leakyReLU(layer); to get values in [-0.5, 0.5] as tanh gives [-1, 1]
        '''

        # --- INPUT LAYER ---
        # Dimension of (3,3,64) reshaped into matrix form (batch_size, 3*3*64)
        in_dim = (-1, 3, 3, 64)
        layer = fullyconnected(z, num_units = in_dim[1]*in_dim[2]*in_dim[3])
        layer = tf.reshape(layer, shape = in_dim)

        # (3, 3, 64) -> (7, 7, 32)
        layer = transposed_convolution(layer, f = 32, s = (1, 1), k = (5, 5), p = 'VALI
        layer = batchnorm(layer)
        layer = leakyReLU(layer, alpha = 0.2)

        # (7,7,32) -> (14, 14, 16)
        layer = transposed_convolution(layer, f = 16)
        layer = batchnorm(layer)
        layer = leakyReLU(layer, alpha = 0.2)

        # (14, 14, 16) -> (28,28,1 or 3)
        layer = transposed_convolution(layer, f = out_channel_dim)
        layer = tanh(layer) / 2
    return layer

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

```
Tests Passed
```

1.2.5 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 [39]: def model_loss(input_real, input_z, out_channel_dim):
             """
             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)
             """
             smooth = 1
             # TODO: Implement Function
             generator_out             = generator(input_z, out_channel_dim, is_train = True)
             d_logits_real, d_out_real = discriminator(input_real, reuse = False)
             d_logits_fake, d_out_fake = discriminator(generator_out, reuse = True)

             d_loss_real      = tf.reduce_mean\
             (tf.nn.sigmoid_cross_entropy_with_logits\
              (labels = tf.ones_like(d_out_real)*smooth, logits = d_logits_real, \
               name = 'd_loss_real'))

             d_loss_fake      = tf.reduce_mean\
             (tf.nn.sigmoid_cross_entropy_with_logits\
              (labels = tf.zeros_like(d_out_fake), logits = d_logits_fake, \
               name = 'd_loss_fake'))

             d_loss           = d_loss_real + d_loss_fake
             g_loss           = tf.reduce_mean\
             (tf.nn.sigmoid_cross_entropy_with_logits\
              (labels=tf.ones_like(d_out_fake), logits = d_logits_fake, name = 'generator_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
```

1.2.6 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 [40]: 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_var = tf.trainable_variables ()
             d_var = [d for d in t_var if d.name.startswith("discriminator")]
             g_var = [g for g in t_var if g.name.startswith("generator")]

             update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
             with tf.control_dependencies(update_ops):
                 d_opt = tf.train.AdamOptimizer(learning_rate = learning_rate, beta1 = beta1)\
                 .minimize(d_loss, var_list = d_var)
                 g_opt = tf.train.AdamOptimizer(learning_rate = learning_rate, beta1 = beta1)\
                 .minimize(g_loss, var_list = g_var)
             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
```

1.3 Neural Network Training

1.3.1 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 [41]: """
         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, False),
        feed_dict={input_z: example_z})

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

1.3.2 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 [42]: import scipy.stats as st
         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
             im_real, in_z, lr = model_inputs(data_shape[1], data_shape[2],
```

```
                                        data_shape[3], z_dim)
    d_loss, g_loss = model_loss(im_real, in_z, data_shape[3])
    d_opt, g_opt   = model_opt(d_loss, g_loss, lr, beta1)

    count = 0

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for epoch_i in range(epoch_count):
            for batch_images in get_batches(batch_size):
                # TODO: Train Model
                count += 1

                mu, sigma = 0, 0.25
                minval    = -0.5
                maxval    =  0.5
                rand_numbers = st.truncnorm.rvs((minval- mu)/sigma, (maxval-mu)/sigma,
                                        loc = mu, scale = sigma, size = (batch_size
                train_d_loss, train_g_loss = sess.run([d_loss, g_loss, d_opt, g_opt], \
                                                feed_dict = {im_real: batch_image
                                                             in_z   : rand_number
                                                             lr     : learning_ra
                #Print losses, optimizer, generator
                if count % 200 == 0:
                    print('Summary at {} batches'.format(count))
                    print('discriminator model loss = {}'.format(train_d_loss))
                    print('generator model loss = {}'.format(train_g_loss))

                    show_generator_output(sess, 9, in_z, data_shape[3], data_image_mode
```

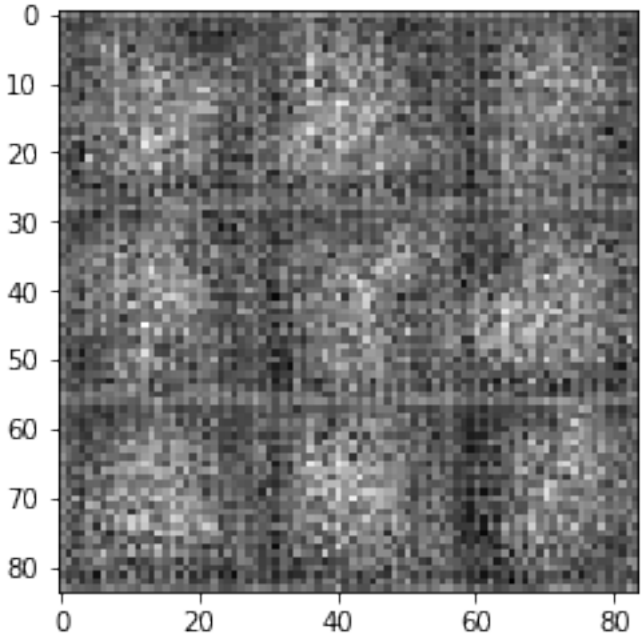
1.3.3 MNIST

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.

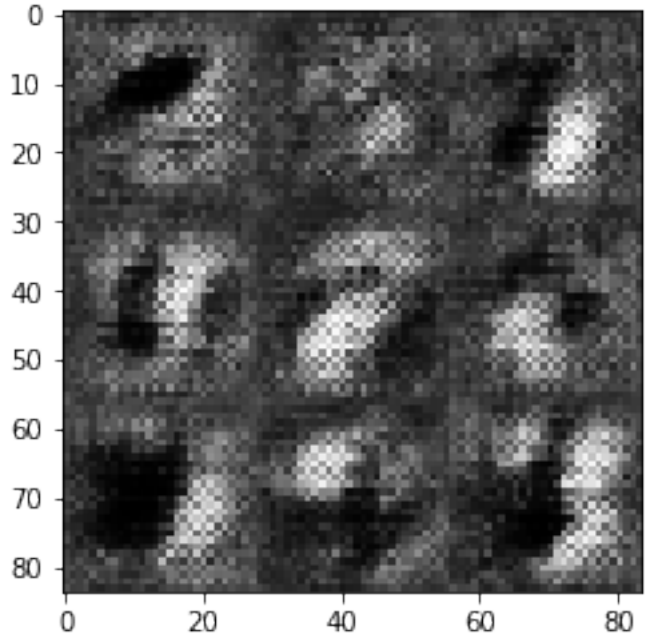
In [43]:
```
%%time
batch_size    = 128
z_dim         = 100
learning_rate = 1e-4
beta1         = 0.9


"""
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)
```

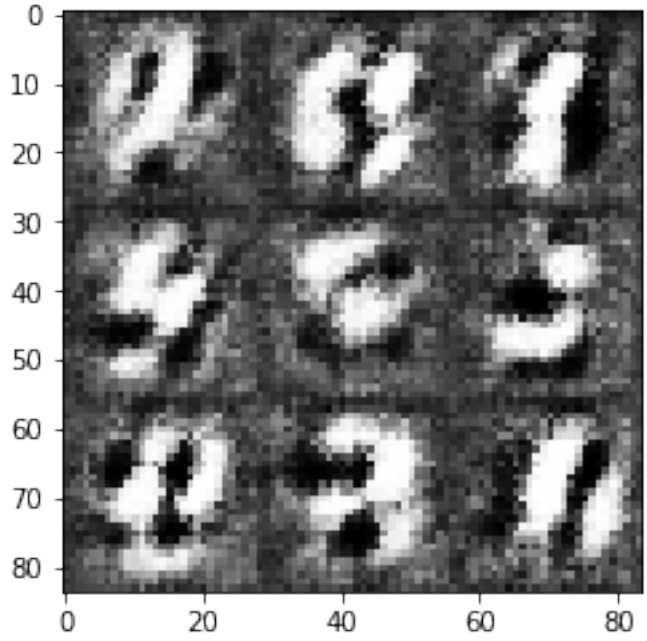
```
Summary at 200 batches
discriminator model loss = 0.11696434020996094
generator model loss = 2.6855640411376953
```



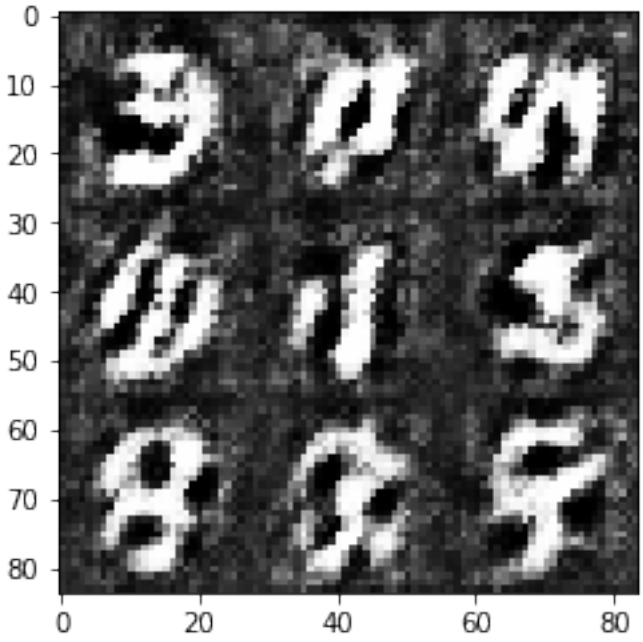

```
Summary at 400 batches
discriminator model loss = 0.04093387722969055
generator model loss = 3.930612564086914
```




```
Summary at 600 batches
discriminator model loss = 0.030817419290542603
generator model loss = 4.29692268371582
```



```
Summary at 800 batches
discriminator model loss = 0.036977220326662064
generator model loss = 4.170480251312256
```




```
CPU times: user 47 s, sys: 9.84 s, total: 56.8 s
Wall time: 54.7 s
```

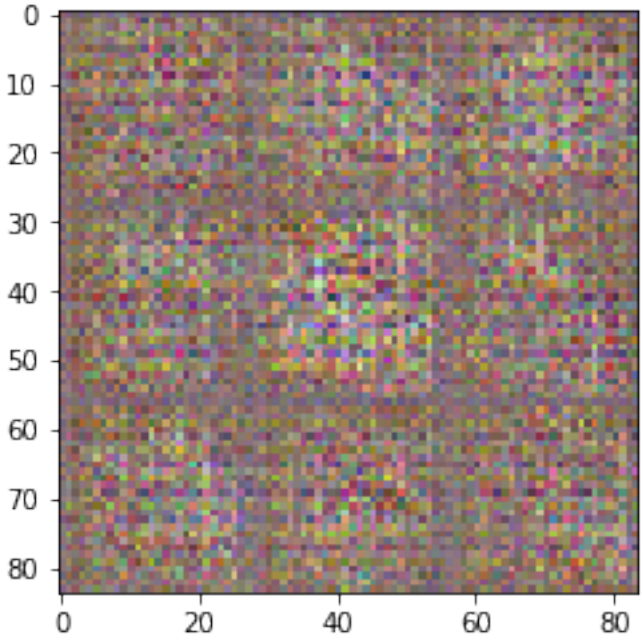
1.3.4 CelebA

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.

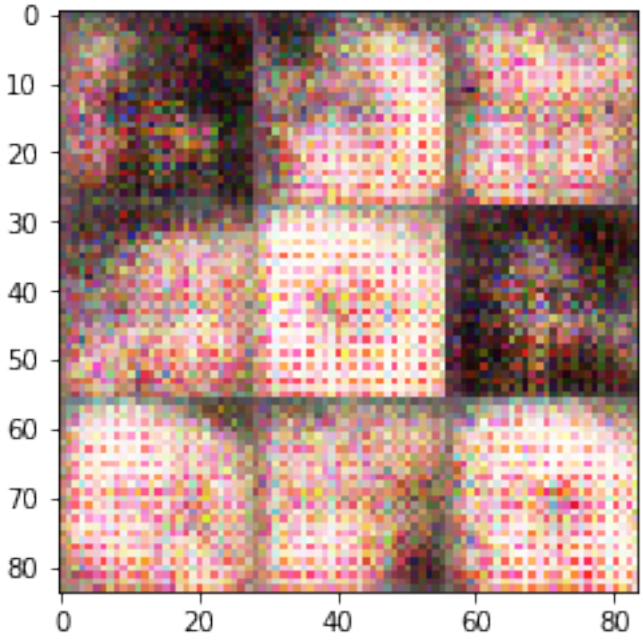
```
In [44]: %%time
         batch_size    = 128
         z_dim         = 100
         learning_rate = 1e-4
         beta1         = 0.9


         """
         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)
```

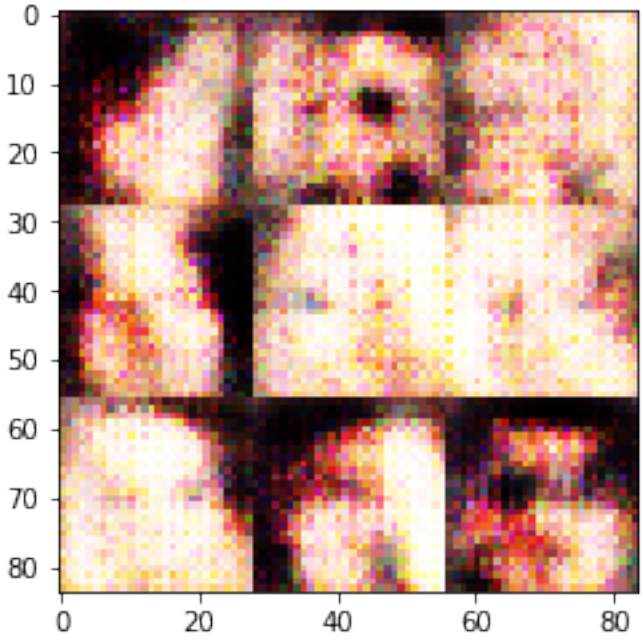
```
Summary at 200 batches
discriminator model loss = 0.03729106858372688
generator model loss = 4.4327392578125
```



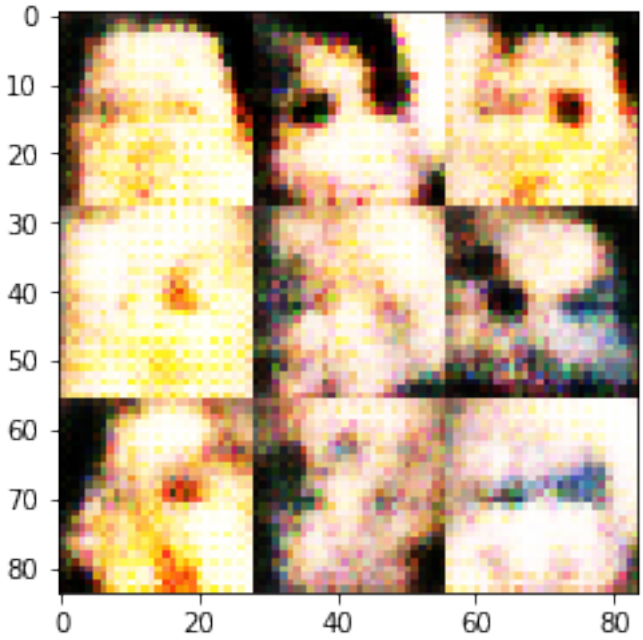

```
Summary at 400 batches
discriminator model loss = 0.0445941798388958
generator model loss = 4.771548748016357
```



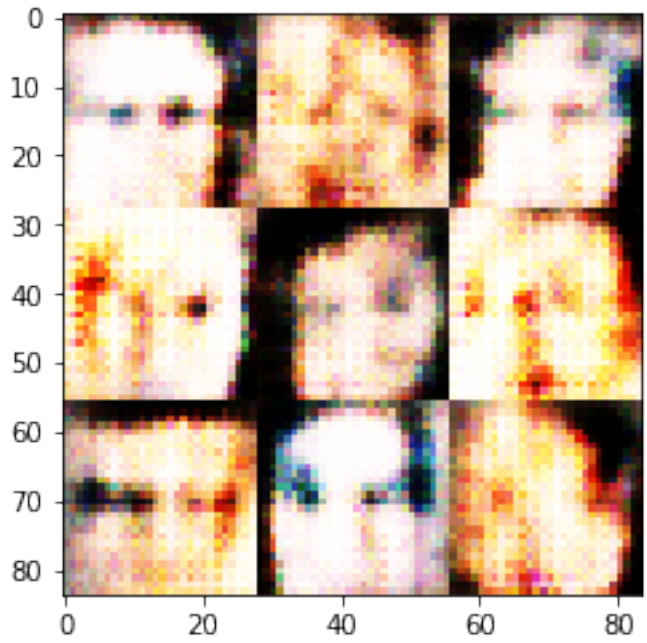

```
Summary at 600 batches
discriminator model loss = 0.03814772889018059
generator model loss = 4.702086448669434
```



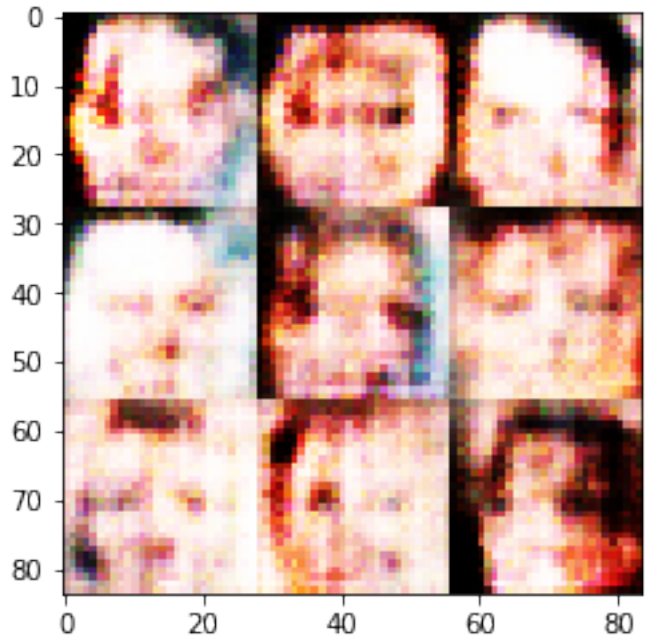
```
Summary at 800 batches
discriminator model loss = 0.06071583181619644
generator model loss = 4.194107532501221
```



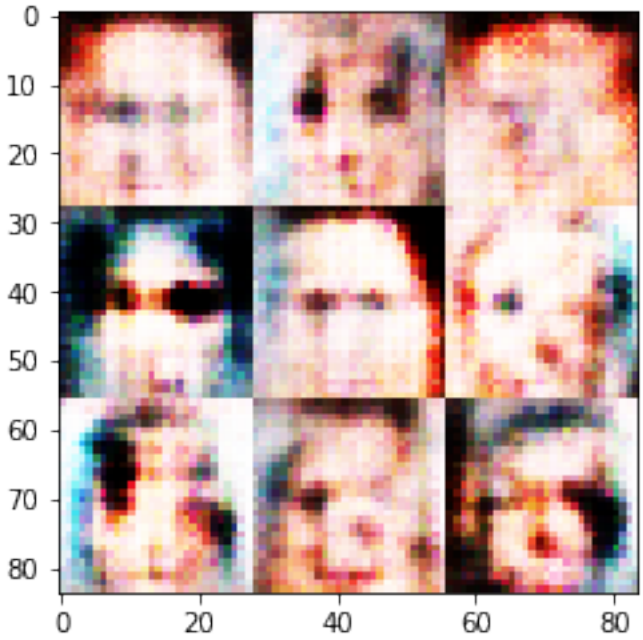

```
Summary at 1000 batches
discriminator model loss = 0.047982364892959595
generator model loss = 4.353063583374023
```




```
Summary at 1200 batches
discriminator model loss = 0.055525705218315125
generator model loss = 4.489264488220215
```



```
Summary at 1400 batches
discriminator model loss = 0.061971329152584076
generator model loss = 4.829124450683594
```




```
CPU times: user 4min 6s, sys: 21.8 s, total: 4min 28s
Wall time: 4min 34s
```

1.3.5 Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_face_generation.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "helper.py" and "problem_unittests.py" files in your submission.