In [1]:
```
# set tf 1.x for colab
%tensorflow_version 1.x
```

```
UsageError: Line magic function `%tensorflow_version` not found.
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore', category=DeprecationWarning)
warnings.filterwarnings('ignore', category=FutureWarning)
```

Generating human faces with Adversarial Networks

© research.nvidia.com

This time we'll train a neural net to generate plausible human faces in all their subtlty: appearance, expression, accessories, etc. 'Cuz when us machines gonna take over Earth, there won't be any more faces left. We want to preserve this data for future iterations. Yikes...

Based on https://github.com/Lasagne/Recipes/pull/94 (https://github.com/Lasagne/Recipes/pull/94) .

In [3]:
```
import sys
sys.path.append("..")
import grading
import download_utils
import tqdm_utils
```

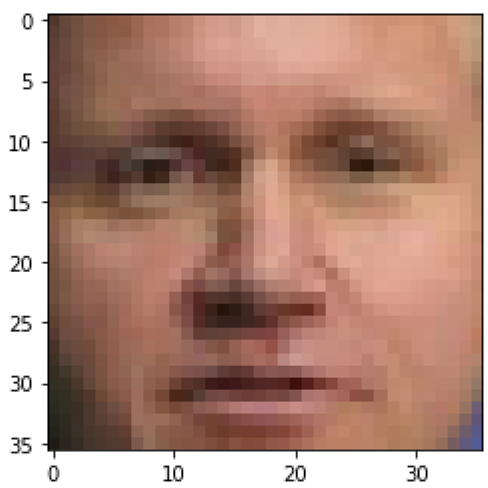
In [4]:
```
download_utils.link_week_4_resources()
```

In [5]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
plt.rcParams.update({'axes.titlesize': 'small'})

from sklearn.datasets import load_digits
#The following line fetches you two datasets: images, usable for autoencoder training and attributes.
#Those attributes will be required for the final part of the assignment (applying smiles), so please keep them in mi
from lfw_dataset import load_lfw_dataset
data,attrs = load_lfw_dataset(dimx=36,dimy=36)

#preprocess faces
data = np.float32(data)/255.

IMG_SHAPE = data.shape[1:]
```

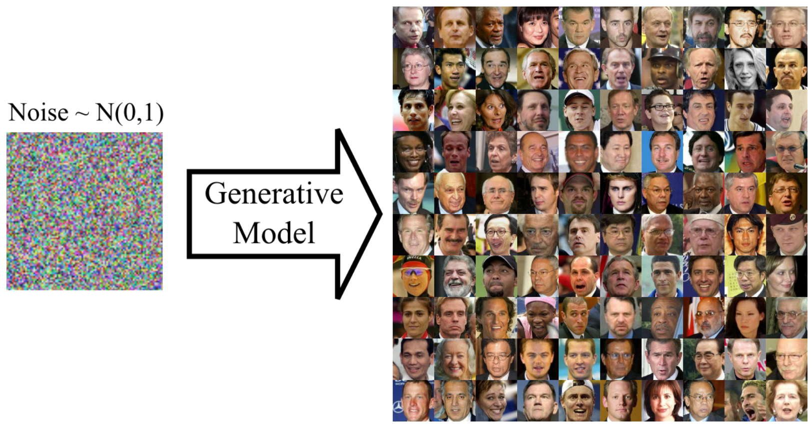
100% 13233/13233 [00:30<00:00, 438.69it/s]

In [6]:
```
#print random image
plt.imshow(data[np.random.randint(data.shape[0])], cmap="gray", interpolation="none")
```

Out[6]: <matplotlib.image.AxesImage at 0x2164cdf8b48>



Generative adversarial nets 101



© torch.github.io

Deep learning is simple, isn't it?

- build some network that generates the face (small image)
- make up a **measure** of **how good that face is**
- optimize with gradient descent :)

The only problem is: how can we engineers tell well-generated faces from bad? And i bet you we won't ask a designer for help.

If we can't tell good faces from bad, we delegate it to yet another neural network!

That makes the two of them:

- **G**enerator - takes random noize for inspiration and tries to generate a face sample.
  - Let's call him **G**(z), where z is a gaussian noize.
- **D**iscriminator - takes a face sample and tries to tell if it's great or fake.
  - Predicts the probability of input image being a **real face**
  - Let's call him **D**(x), x being an image.
  - **D(x)** is a predition for real image and **D(G(z))** is prediction for the face made by generator.

Before we dive into training them, let's construct the two networks.

In [7]:

```
import tensorflow as tf
from keras_utils import reset_tf_session
s = reset_tf_session()

import keras
from keras.models import Sequential
from keras import layers as L
```

```
WARNING:tensorflow:From ..\keras_utils.py:68: The name tf.get_default_session is deprecated. Please use tf.compat.v1.ge
t_default_session instead.

WARNING:tensorflow:From ..\keras_utils.py:75: The name tf.ConfigProto is deprecated. Please use tf.compat.v1.ConfigProt
o instead.

WARNING:tensorflow:From ..\keras_utils.py:77: The name tf.InteractiveSession is deprecated. Please use tf.compat.v1.Int
eractiveSession instead.


Using TensorFlow backend.
```

In [8]:

```
CODE_SIZE = 256

generator = Sequential()
generator.add(L.InputLayer([CODE_SIZE],name='noise'))
generator.add(L.Dense(10*8*8, activation='elu'))

generator.add(L.Reshape((8,8,10)))
generator.add(L.Deconvolution2D(64,kernel_size=(5,5),activation='elu'))
generator.add(L.Deconvolution2D(64,kernel_size=(5,5),activation='elu'))
generator.add(L.UpSampling2D(size=(2,2)))
generator.add(L.Deconvolution2D(32,kernel_size=3,activation='elu'))
generator.add(L.Deconvolution2D(32,kernel_size=3,activation='elu'))
generator.add(L.Deconvolution2D(32,kernel_size=3,activation='elu'))

generator.add(L.Conv2D(3,kernel_size=3,activation=None))

```

In [9]:

```
assert generator.output_shape[1:] == IMG_SHAPE, "generator must output an image of shape %s, but instead it produces
```

Discriminator

- Discriminator is your usual convolutional network with interlooping convolution and pooling layers
- The network does not include dropout/batchnorm to avoid learning complications.
- We also regularize the pre-output layer to prevent discriminator from being too certain.

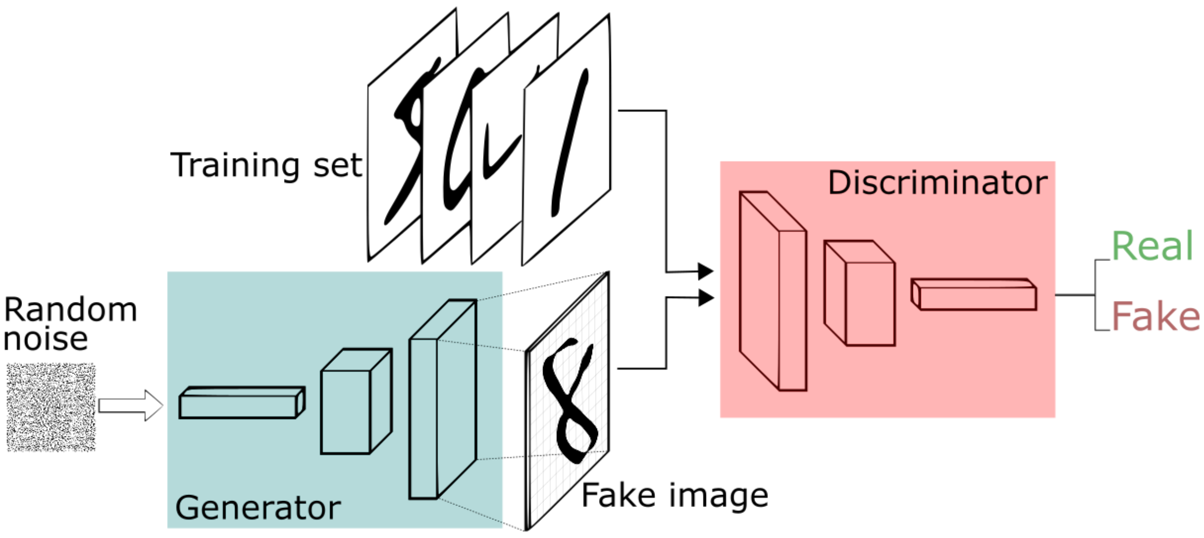
In [10]:
```python
discriminator = Sequential()

discriminator.add(L.InputLayer(IMG_SHAPE))

# <build discriminator body>
discriminator.add(L.Conv2D(16,[2,2],padding='same',activation='relu'))
discriminator.add(L.Conv2D(32,[2,2],padding='same',activation='relu'))
discriminator.add(L.MaxPool2D())

discriminator.add(L.Conv2D(64,[2,2],padding='same',activation='relu'))
discriminator.add(L.Conv2D(128,[2,2],padding='same',activation='relu'))
discriminator.add(L.MaxPool2D())


discriminator.add(L.Flatten())
discriminator.add(L.Dense(256,activation='tanh'))
discriminator.add(L.Dense(2,activation=tf.nn.log_softmax))
```

```
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\keras\backend\tensorflow_backend.py:4
070: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.
```

Training

We train the two networks concurrently:

- Train **discriminator** to better distinguish real data from **current** generator
- Train **generator** to make discriminator think generator is real
- Since discriminator is a differentiable neural network, we train both with gradient descent.



© deeplearning4j.org

Training is done iteratively until discriminator is no longer able to find the difference (or until you run out of patience).

Tricks:

- Regularize discriminator output weights to prevent explosion
- Train generator with **adam** to speed up training. Discriminator trains with SGD to avoid problems with momentum.
- More: [https://github.com/soumith/ganhacks (https://github.com/soumith/ganhacks)](https://github.com/soumith/ganhacks)

In [11]:
```python
noise = tf.placeholder('float32',[None,CODE_SIZE])
real_data = tf.placeholder('float32',[None,]+list(IMG_SHAPE))

logp_real = discriminator(real_data)

generated_data = generator(noise) #<gen(noise)>

logp_gen = discriminator(generated_data) #<log P(real | gen(noise))
```

In [12]:
```
########################
#discriminator training#
########################

d_loss = -tf.reduce_mean(logp_real[:,1] + logp_gen[:,0])

#regularize
d_loss += tf.reduce_mean(discriminator.layers[-1].kernel**2)

#optimize
disc_optimizer =  tf.train.GradientDescentOptimizer(1e-3).minimize(d_loss,var_list=discriminator.trainable_weights)
```

```
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\math_grad.py:12
05: add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

In [13]:
```
########################
###generator training###
########################

g_loss = -tf.reduce_mean(logp_gen[:,1]) #<generator loss>

gen_optimizer = tf.train.AdamOptimizer(1e-4).minimize(g_loss,var_list=generator.trainable_weights)
```

In [14]:
```
s.run(tf.global_variables_initializer())
```

Auxiliary functions

Here we define a few helper functions that draw current data distributions and sample training batches.

In [15]:
```
def sample_noise_batch(bsize):
    return np.random.normal(size=(bsize, CODE_SIZE)).astype('float32')

def sample_data_batch(bsize):
    idxs = np.random.choice(np.arange(data.shape[0]), size=bsize)
    return data[idxs]

def sample_images(nrow,ncol, sharp=False):
    images = generator.predict(sample_noise_batch(bsize=nrow*ncol))
    if np.var(images)!=0:
        images = images.clip(np.min(data),np.max(data))
    for i in range(nrow*ncol):
        plt.subplot(nrow,ncol,i+1)
        if sharp:
            plt.imshow(images[i].reshape(IMG_SHAPE),cmap="gray", interpolation="none")
        else:
            plt.imshow(images[i].reshape(IMG_SHAPE),cmap="gray")
    plt.show()

def sample_probas(bsize):
    plt.title('Generated vs real data')
    plt.hist(np.exp(discriminator.predict(sample_data_batch(bsize)))[:,1],
             label='D(x)', alpha=0.5,range=[0,1])
    plt.hist(np.exp(discriminator.predict(generator.predict(sample_noise_batch(bsize))))[:,1],
             label='D(G(z))',alpha=0.5,range=[0,1])
    plt.legend(loc='best')
    plt.show()
```

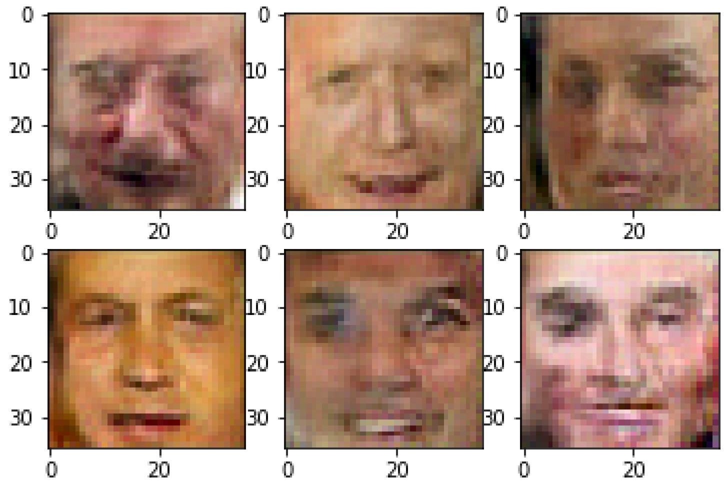
Training

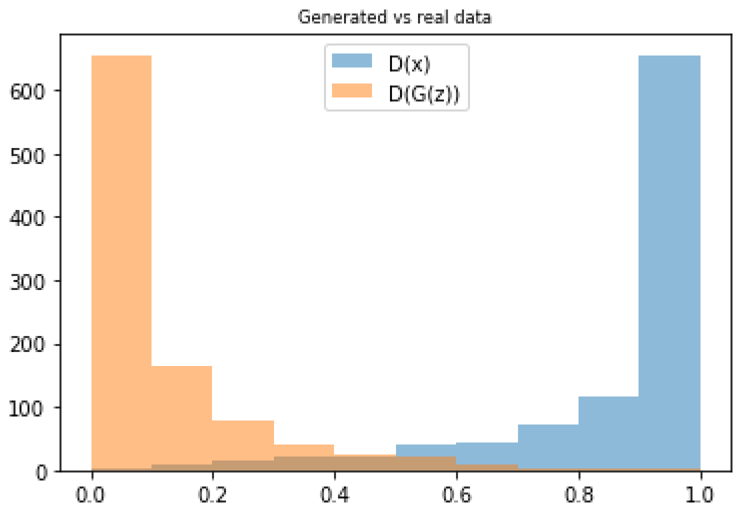
Main loop. We just train generator and discriminator in a loop and plot results once every N iterations.

In [16]:

```
%%time
from IPython import display

for epoch in tqdm_utils.tqdm_notebook_failsafe(range(15000)):

    feed_dict = {
        real_data:sample_data_batch(100),
        noise:sample_noise_batch(100)
    }

    for i in range(5):
        s.run(disc_optimizer,feed_dict)

    s.run(gen_optimizer,feed_dict)

    if epoch %100==0:
        display.clear_output(wait=True)
        print('Epoch status:', epoch)
        sample_images(2,3,True)
        sample_probas(1000)

```

```
Epoch status: 14900
```





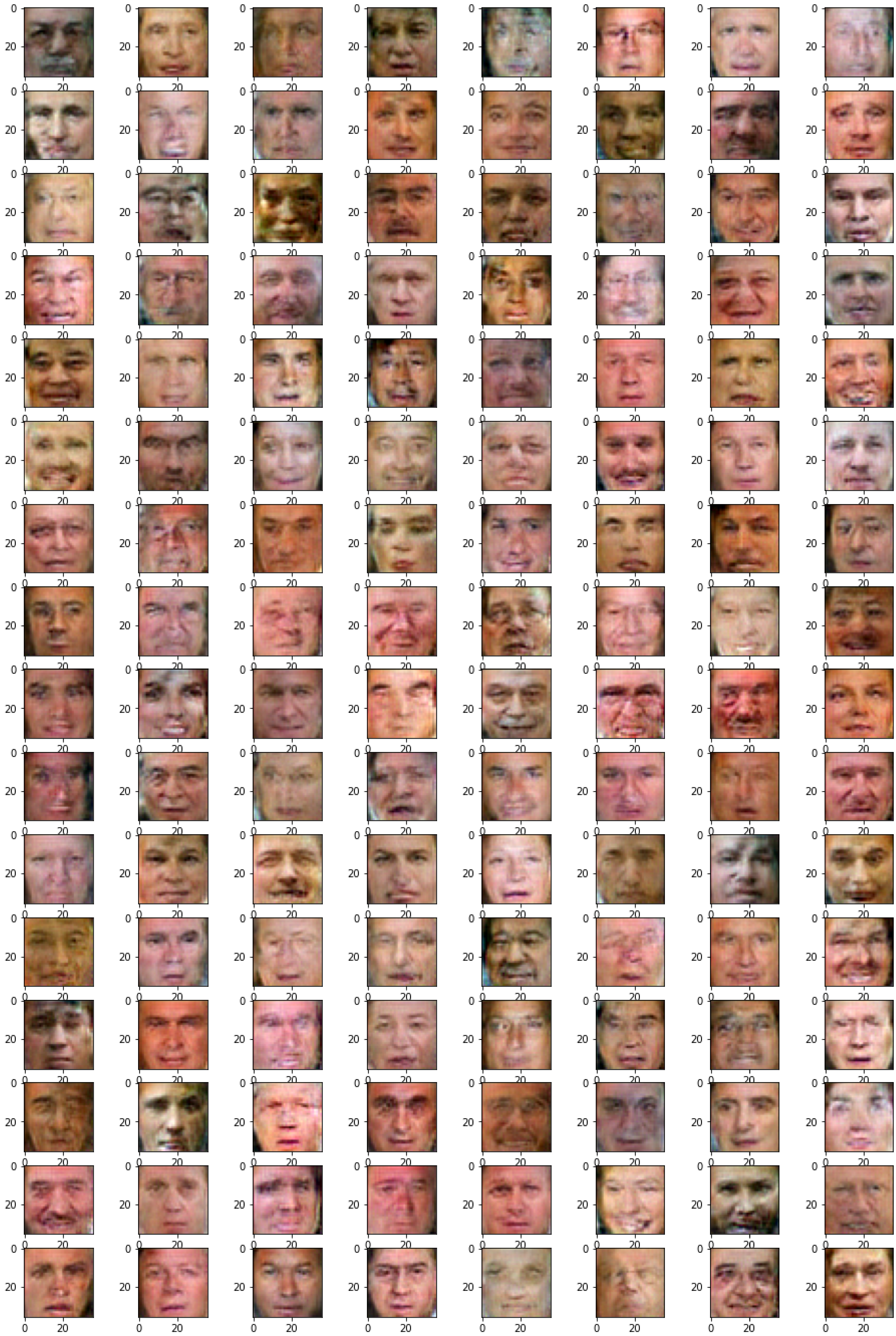

```
Wall time: 51min 24s
```

In [ ]:

```
from submit_honor import submit_honor
submit_honor((generator, discriminator), <YOUR_EMAIL>, <YOUR_TOKEN>)
```

In [18]:

```python
#The network was trained for about 15k iterations.
#Training for longer yields MUCH better results
plt.figure(figsize=[16,24])
sample_images(16,8)
```



In [ ]:

```

```