MP4_P1

November 16, 2020

1 Generative Adversarial Networks

For this part of the assignment you implement two different types of generative adversarial networks. We will train the networks on the Celeb A dataset which is a large set of celebrity face images.

[1]:
```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
from gan.utils import show_images
import numpy as np

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

%load_ext autoreload
%autoreload 2
```

[2]:
```
from gan.train import train
```

[3]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "gpu")
```

2 GAN loss functions

In this assignment you will implement two different types of GAN cost functions. You will first implement the loss from the original GAN paper. You will also implement the loss from LS-GAN.

2.0.1 GAN loss

TODO: Implement the `discriminator_loss` and `generator_loss` functions in `gan/losses.py`.

The generator loss is given by:

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

and the discriminator loss is:

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

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

HINTS: You should use the `torch.nn.functional.binary_cross_entropy_with_logits` function to compute the binary cross entropy loss since it is more numerically stable than using a softmax followed by BCE loss. The BCE loss is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log\left(1 - D(G(z))\right)$, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

[4]:
```
from gan.losses import discriminator_loss, generator_loss
```

2.0.2 Least Squares GAN loss

TODO: Implement the `ls_discriminator_loss` and `ls_generator_loss` functions in `gan/losses.py`.

We'll now look at Least Squares GAN, 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 equation (9) in the paper, with the generator loss:

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

and the discriminator loss:

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

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 (`scores_real` and `scores_fake`).

[5]:
```
from gan.losses import ls_discriminator_loss, ls_generator_loss
```

3 GAN model architecture

TODO: Implement the `Discriminator` and `Generator` networks in `gan/models.py`.

We recommend the following architectures which are inspired by DCGAN:

Discriminator:

- convolutional layer with in_channels=3, out_channels=128, kernel=4, stride=2
- convolutional layer with in_channels=128, out_channels=256, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=256, out_channels=512, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=512, out_channels=1024, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=1024, out_channels=1, kernel=4, stride=1

Instead of Relu we LeakyReLu throughout the discriminator (we use a negative slope value of 0.2).

The output of your discriminator should be a single value score corresponding to each input sample. See `torch.nn.LeakyReLU`.

Generator:

Note: In the generator, you will need to use transposed convolution (sometimes known as fractionally-strided convolution or deconvolution). This function is implemented in pytorch as `torch.nn.ConvTranspose2d`.

- transpose convolution with in_channels=NOISE_DIM, out_channels=1024, kernel=4, stride=1
- batch norm
- transpose convolution with in_channels=1024, out_channels=512, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=512, out_channels=256, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=256, out_channels=128, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=128, out_channels=3, kernel=4, stride=2

The output of the final layer of the generator network should have a `tanh` nonlinearity to output values between -1 and 1. The output should be a 3x64x64 tensor for each sample (equal dimensions to the images from the dataset).

[6]:
```
import sys
sys.path.insert(0, "gan/")

from gan.models import Discriminator, Generator
```

4 Data loading: Celeb A Dataset

The CelebA images we provide have been filtered to obtain only images with clear faces and have been cropped and downsampled to 128x128 resolution.

[7]:
```
batch_size = 128
scale_size = 64  # We resize the images to 64x64 for training

celeba_root = 'celeba_data'
```

[8]:
```
celeba_train = ImageFolder(root=celeba_root, transform=transforms.Compose([
  transforms.Resize(scale_size),
  transforms.ToTensor(),
]))

celeba_loader_train = DataLoader(celeba_train, batch_size=batch_size,␣
 ↪drop_last=True)
```

4.0.1 Visualize dataset

[9]:
```
imgs = celeba_loader_train.__iter__().next()[0].numpy().squeeze()
show_images(imgs, color=True)
```

5 Training

TODO: Fill in the training loop in `gan/train.py`.

[10]:
```
NOISE_DIM = 100
NUM_EPOCHS = 15
learning_rate = 0.0002
```

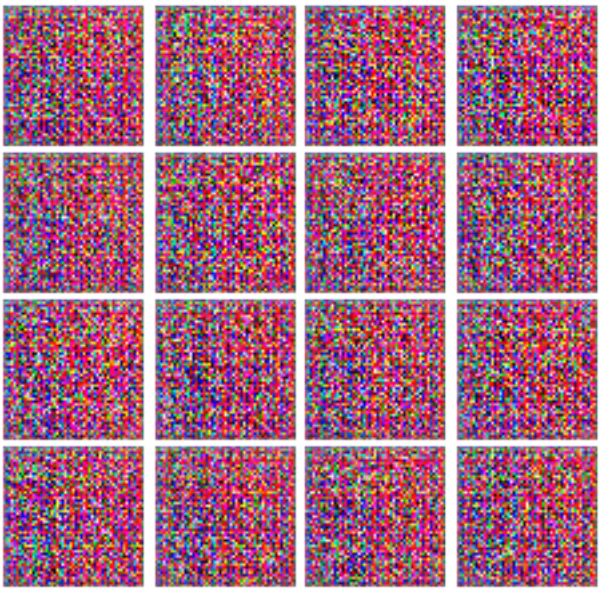
5.0.1 Train GAN

[11]:
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

[12]:
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.
↪999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.
↪999))
```

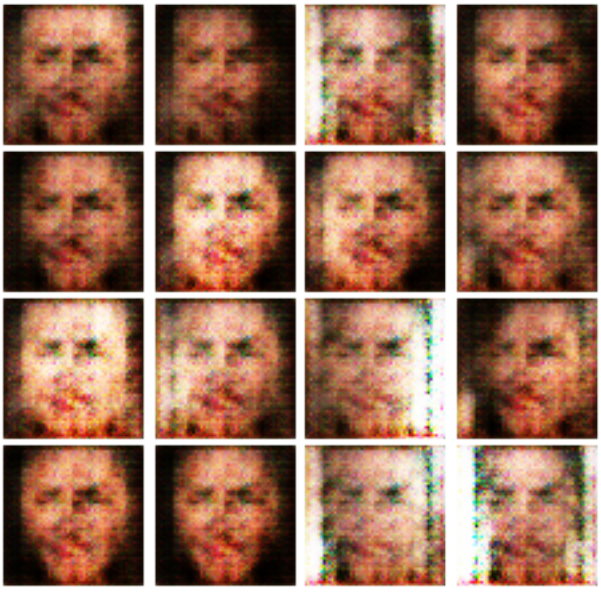
[13]:
```
# original gan
train(D, G, D_optimizer, G_optimizer, discriminator_loss,
          generator_loss, num_epochs=NUM_EPOCHS, show_every=150,
          train_loader=celeba_loader_train, device=device)
```

```
EPOCH:  1
Iter: 0, D: 1.272, G:11.46
```

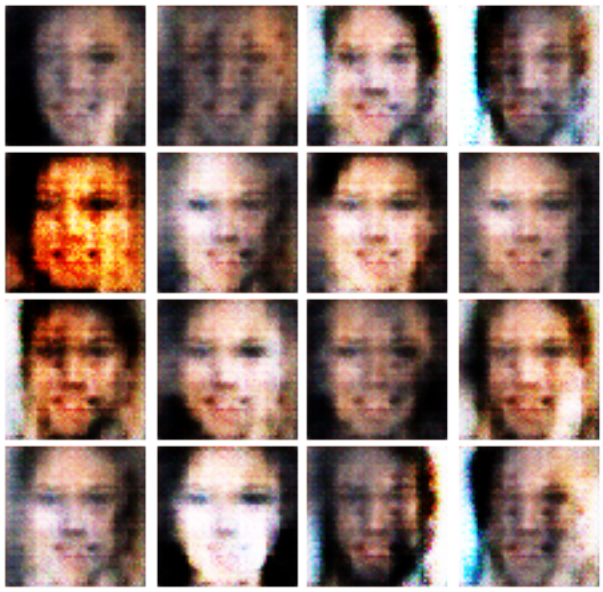


```
Iter: 150, D: 0.8672, G:1.586
```

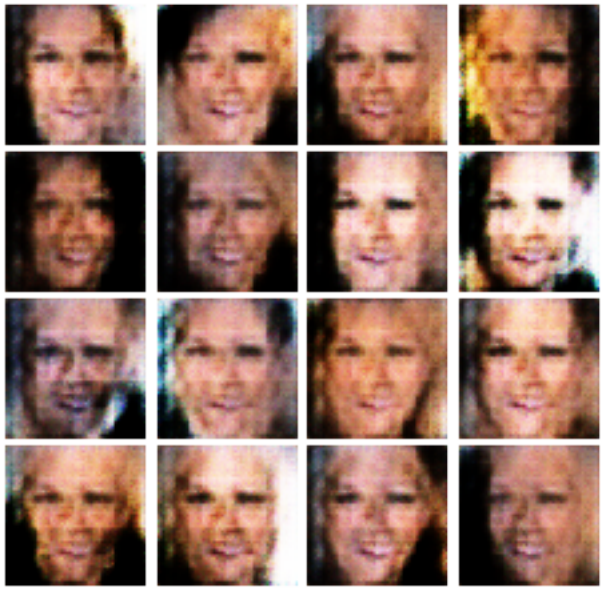
Iter: 300, D: 1.979, G:3.622



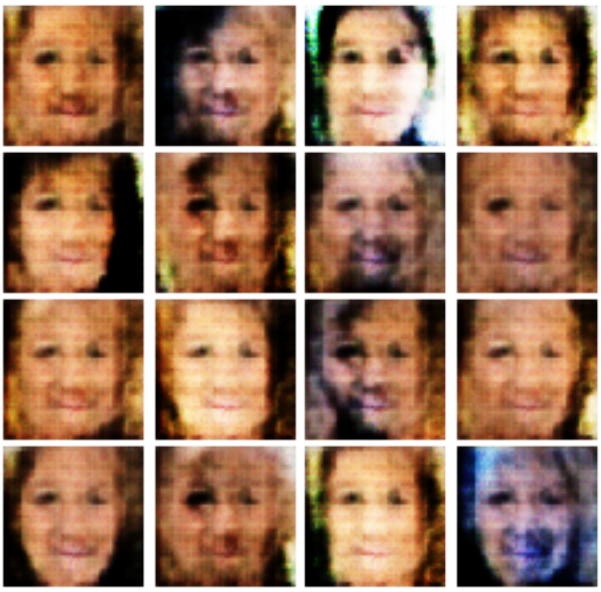
Iter: 450, D: 0.7691, G:2.214



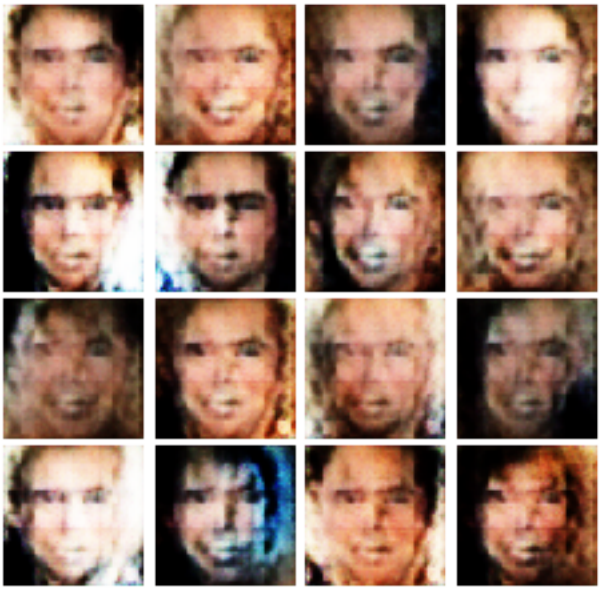
Iter: 600, D: 0.8871, G:4.267



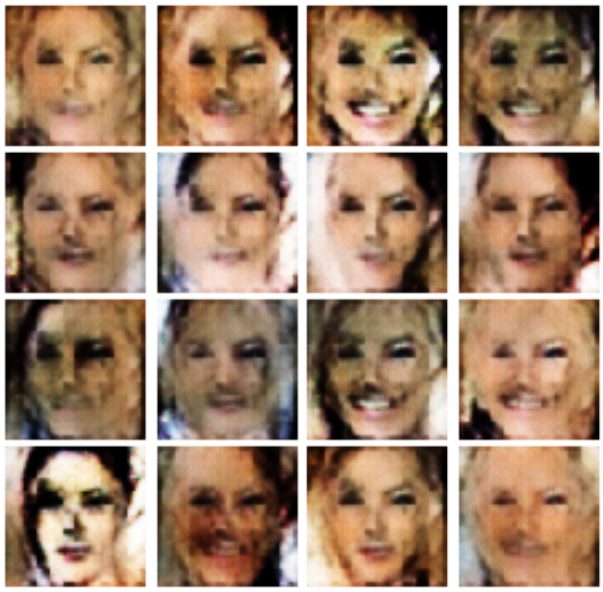
Iter: 750, D: 0.5991, G:2.303



Iter: 900, D: 0.4315, G:3.691



EPOCH: 2
Iter: 1050, D: 0.4785, G:2.8



Iter: 1200, D: 0.7259, G:2.488

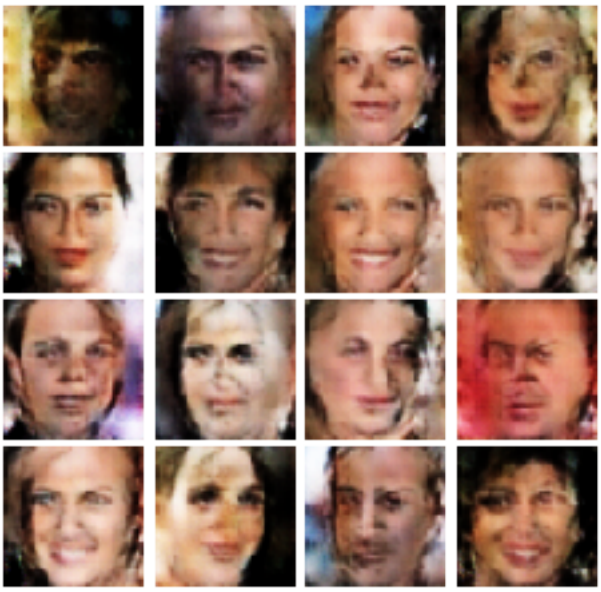
Iter: 1350, D: 0.4909, G:3.027

Iter: 1500, D: 0.7369, G:5.277

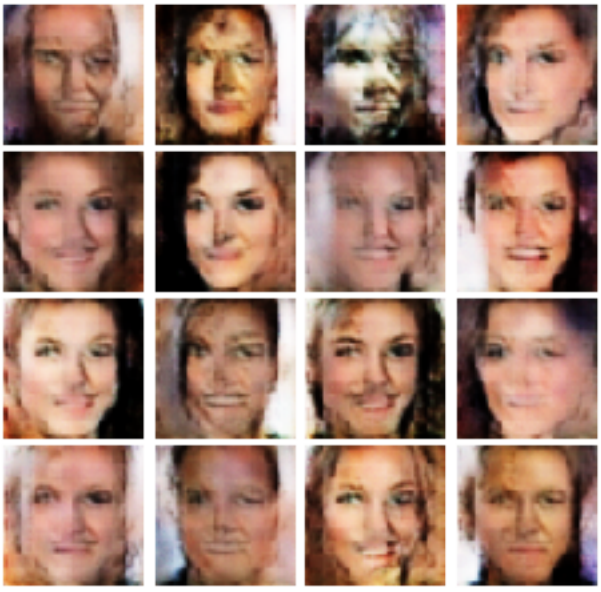
Iter: 1650, D: 0.7613, G:4.393

Iter: 1800, D: 0.2669, G:2.408

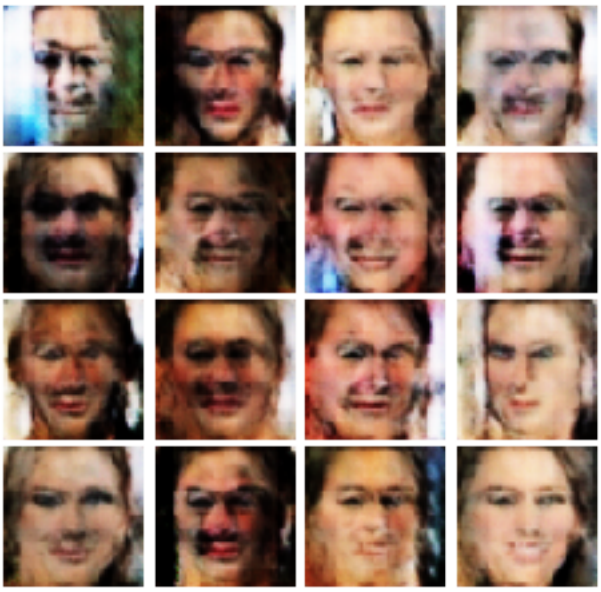
Iter: 1950, D: 0.1627, G:3.022



```
EPOCH:  3
Iter: 2100, D: 1.57, G:6.524
```

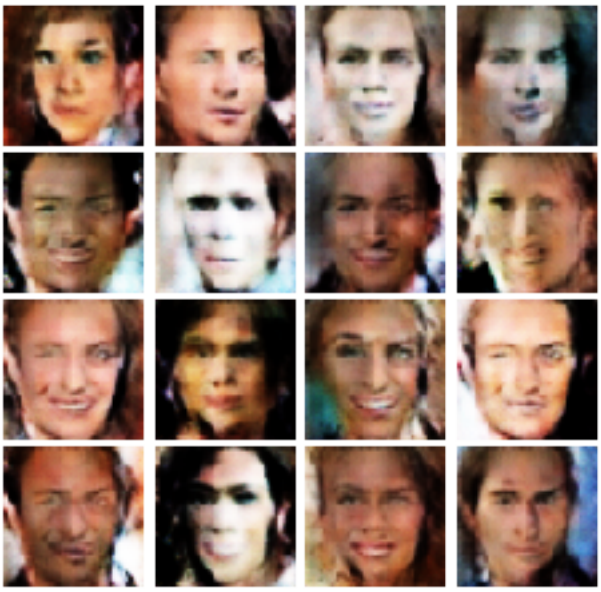


```
Iter: 2250, D: 0.9423, G:2.493
```



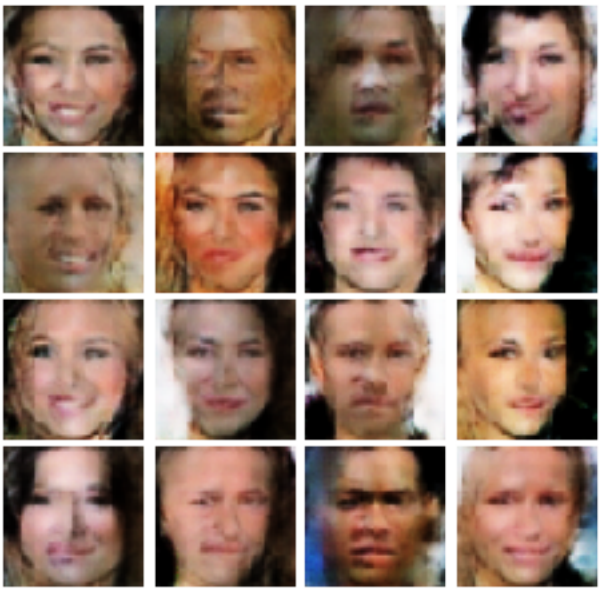
Iter: 2400, D: 0.1711, G:2.825

Iter: 2550, D: 0.138, G:2.724



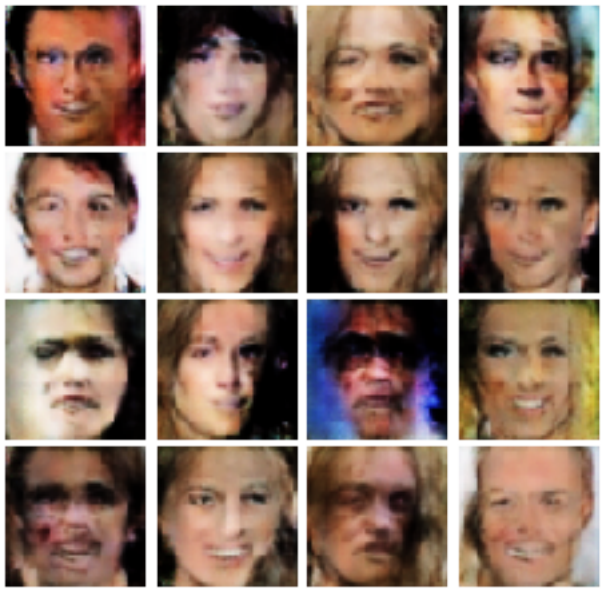
Iter: 2700, D: 0.3795, G:3.612

Iter: 2850, D: 0.7126, G:5.387

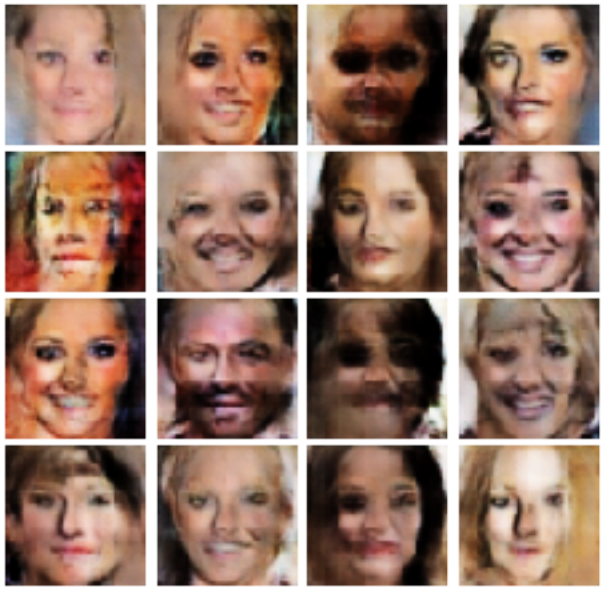


```
EPOCH:  4
Iter: 3000, D: 0.4624, G:4.246
```

```
Iter: 3150, D: 0.2361, G:4.092
```



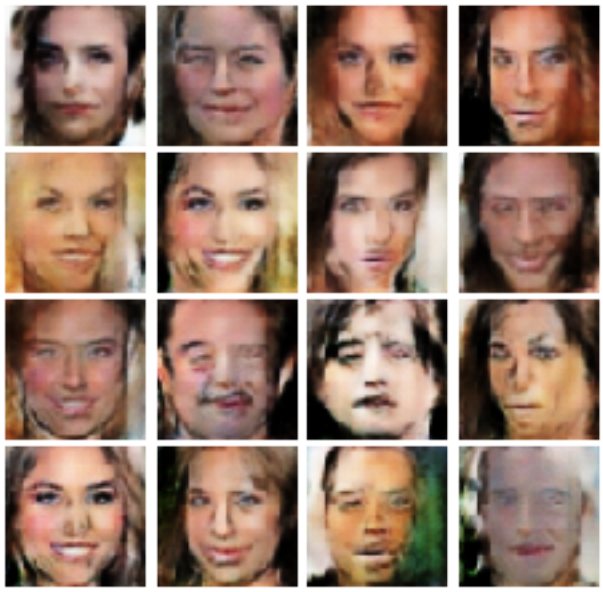
Iter: 3300, D: 0.04706, G:4.287



Iter: 3450, D: 0.04108, G:5.351

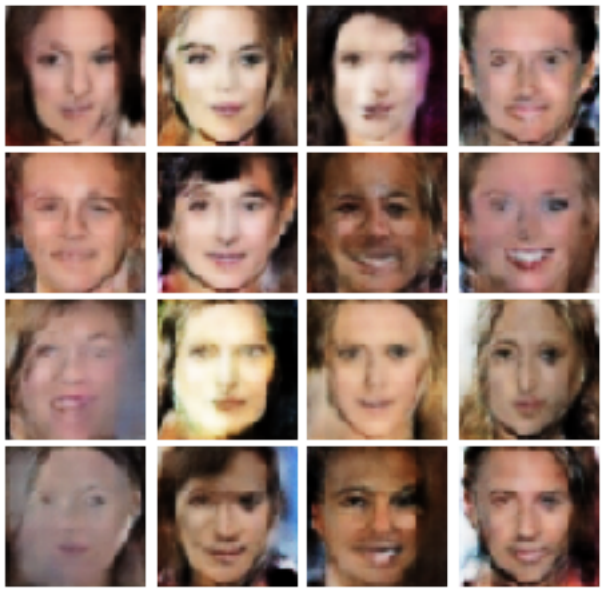
Iter: 3600, D: 0.4525, G:4.946

Iter: 3750, D: 0.7724, G:6.248

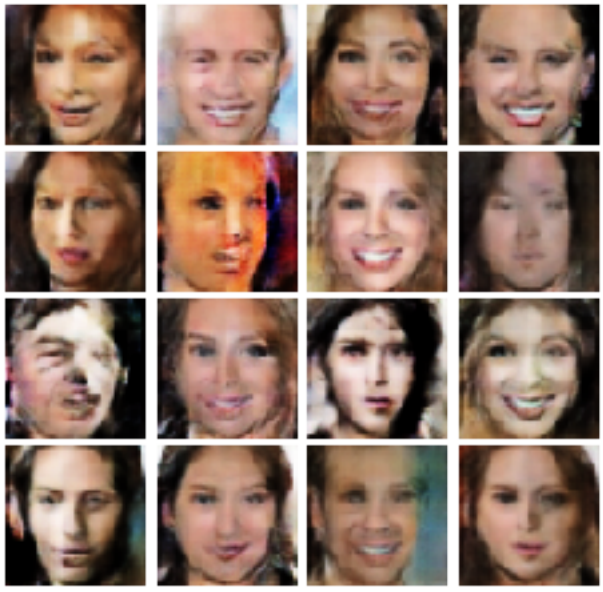


Iter: 3900, D: 0.1239, G:2.954

EPOCH: 5
Iter: 4050, D: 0.586, G:5.22

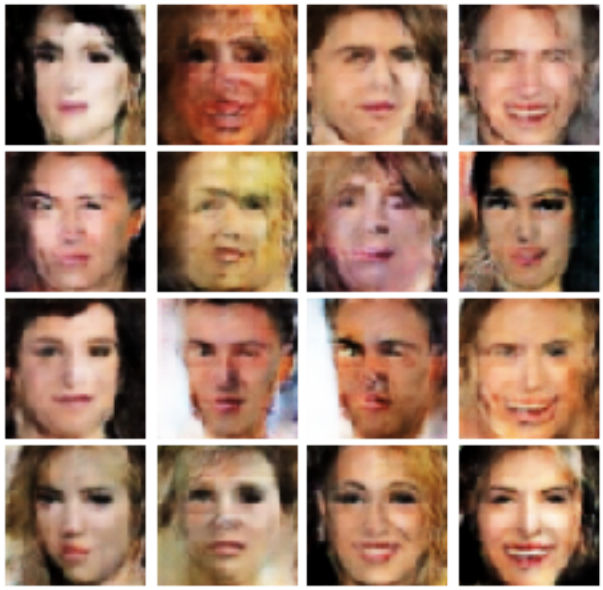


Iter: 4200, D: 0.05658, G:3.461



Iter: 4350, D: 0.05471, G:6.47

Iter: 4500, D: 0.33, G:3.028

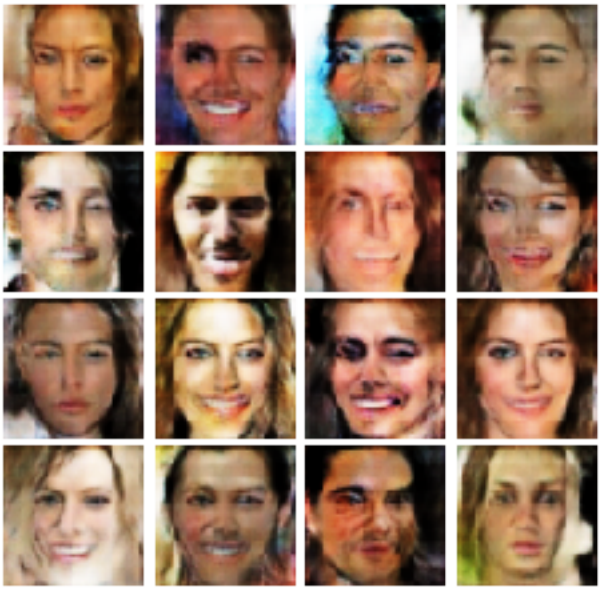


Iter: 4650, D: 0.05662, G:3.678

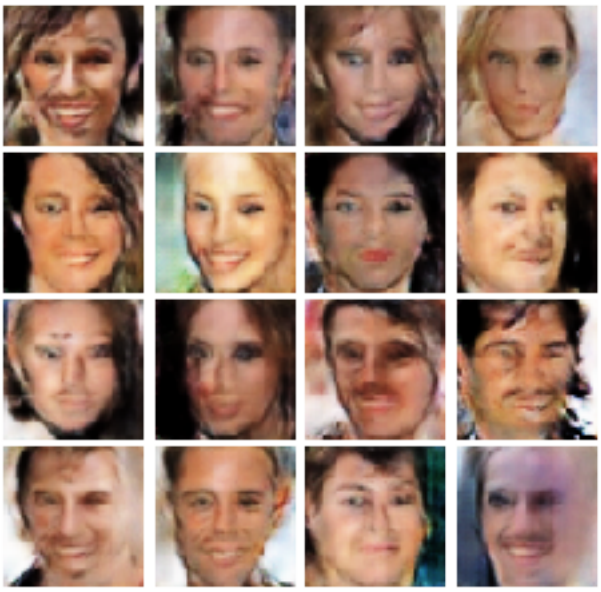
```
Iter: 4800, D: 0.08579, G:1.966
```

```
EPOCH:  6
Iter: 4950, D: 0.04087, G:4.593
```

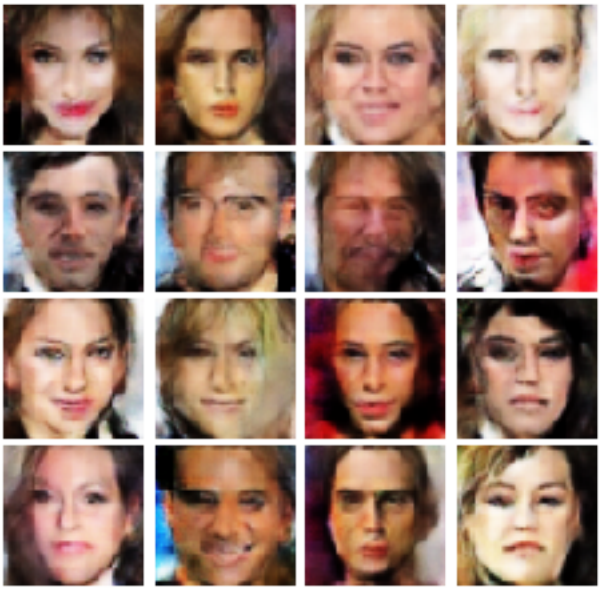
Iter: 5100, D: 0.1295, G:2.201



Iter: 5250, D: 0.0965, G:5.25

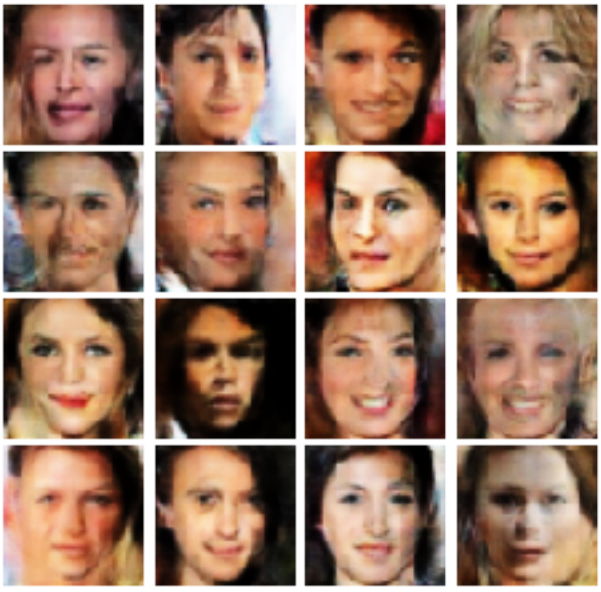


Iter: 5400, D: 0.07719, G:4.357



Iter: 5550, D: 0.1246, G:3.164

Iter: 5700, D: 0.4025, G:5.581

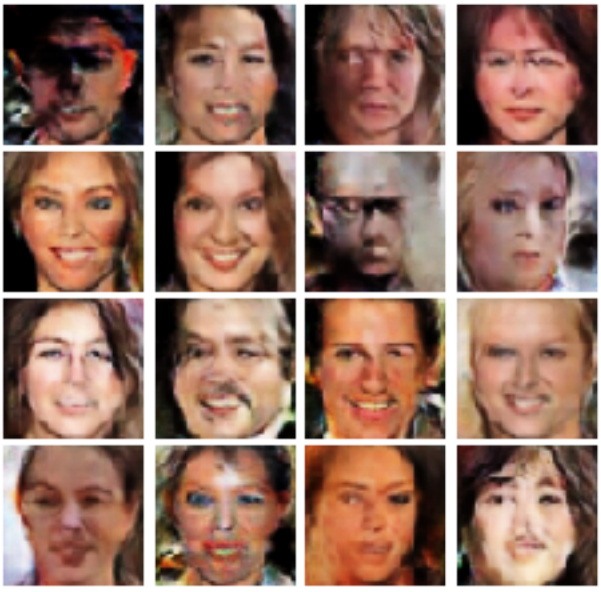


Iter: 5850, D: 0.1157, G:2.967

```
EPOCH:  7
Iter: 6000, D: 0.1989, G:5.206
```

```
Iter: 6150, D: 2.444, G:4.699
```

Iter: 6300, D: 0.2108, G:5.43



Iter: 6450, D: 0.3089, G:3.936

Iter: 6600, D: 0.5252, G:5.044

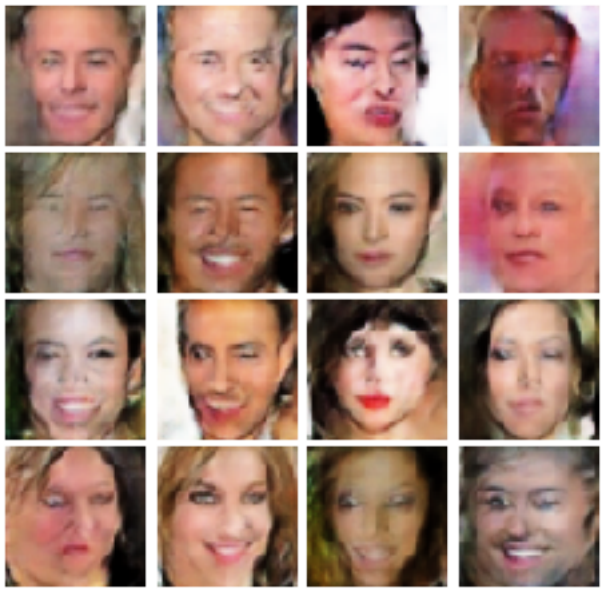
Iter: 6750, D: 0.04469, G:4.414

Iter: 6900, D: 0.6289, G:3.159

EPOCH: 8
Iter: 7050, D: 0.01218, G:5.025

Iter: 7200, D: 0.2076, G:4.304

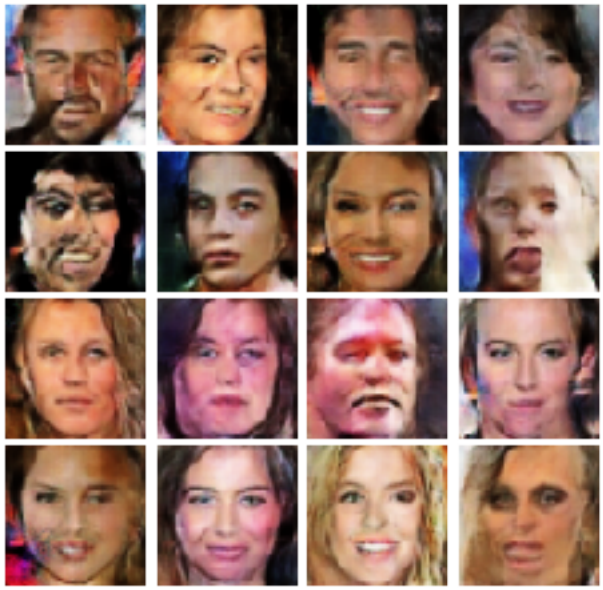
Iter: 7350, D: 0.1431, G:3.173



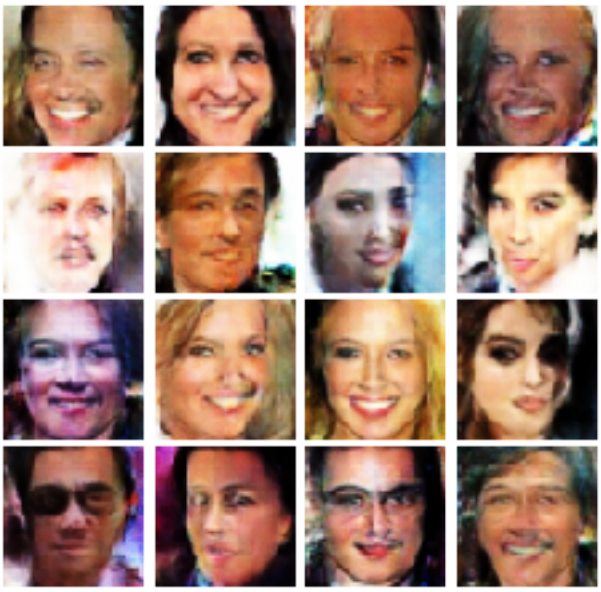
Iter: 7500, D: 0.07339, G:2.602

Iter: 7650, D: 0.3263, G:4.715

```
Iter: 7800, D: 0.1363, G:5.405
```

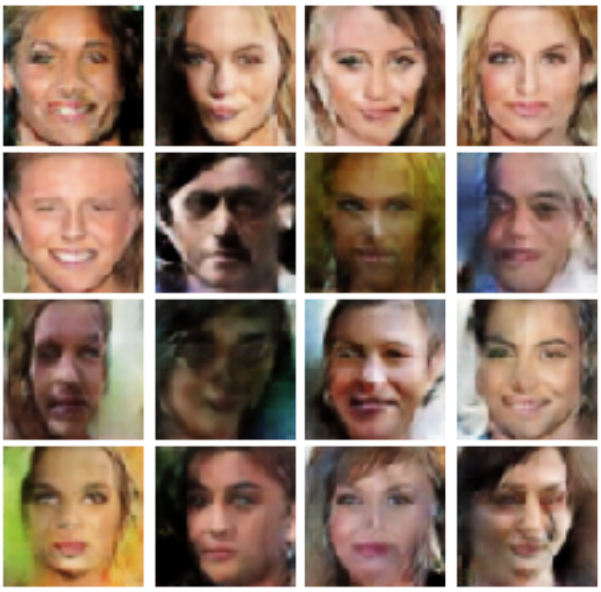


```
EPOCH:  9
Iter: 7950, D: 0.09827, G:4.293
```

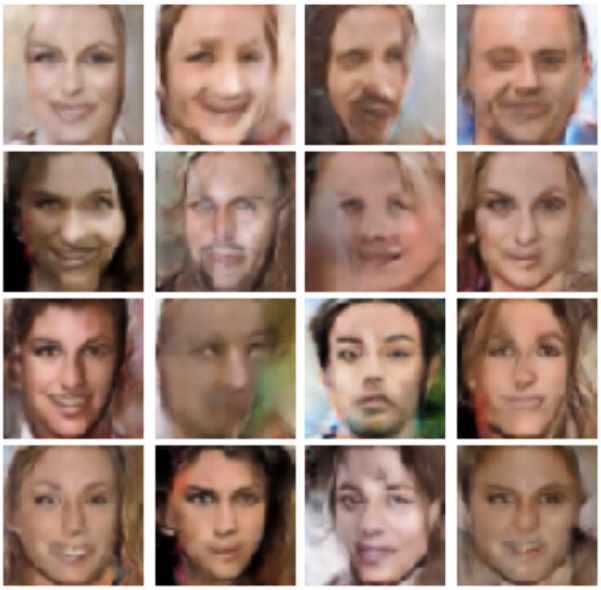


Iter: 8100, D: 0.09024, G:3.855

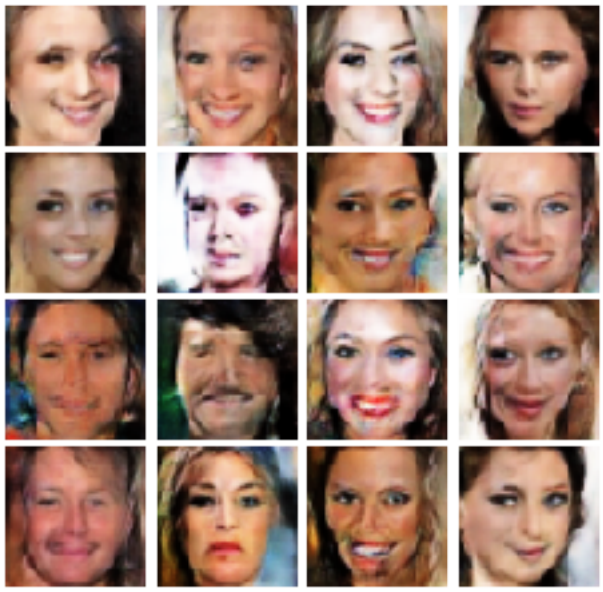
Iter: 8250, D: 0.1479, G:4.253



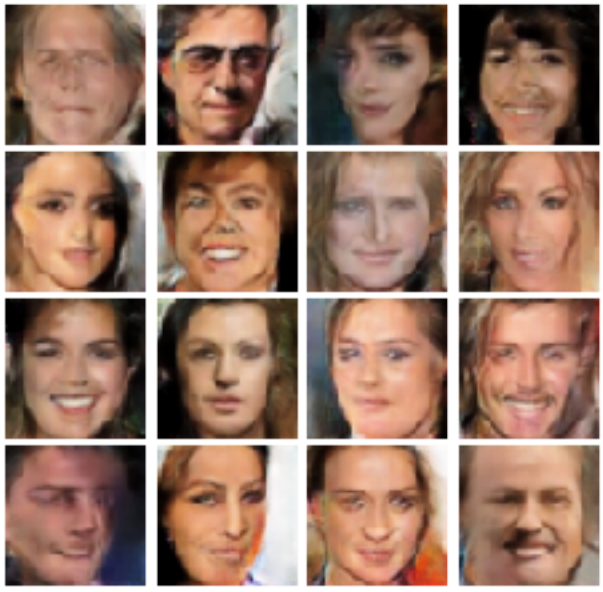
Iter: 8400, D: 0.3961, G:4.145



Iter: 8550, D: 0.3249, G:5.461

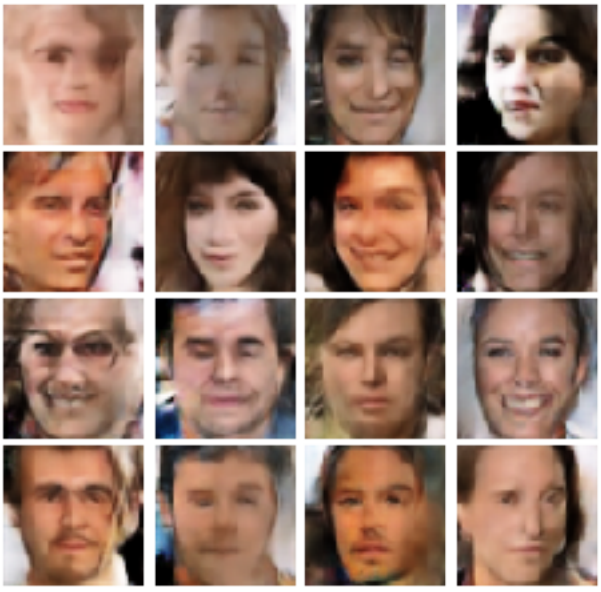


Iter: 8700, D: 0.7283, G:6.875

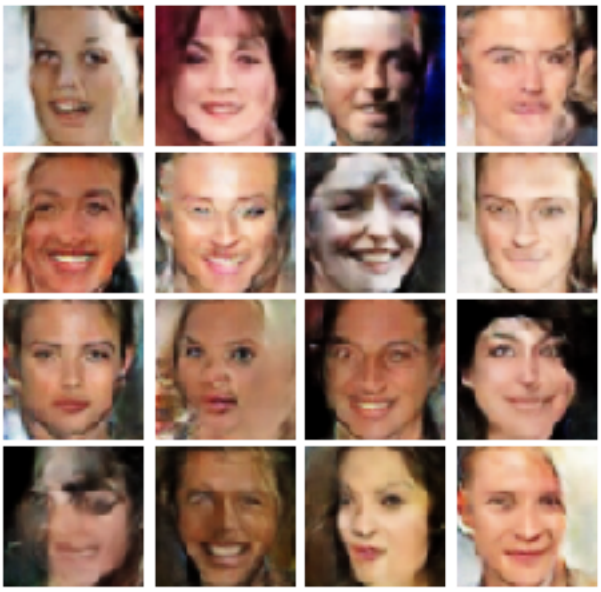


Iter: 8850, D: 0.1049, G:3.991

```
EPOCH:  10
Iter: 9000, D: 0.03767, G:4.114
```



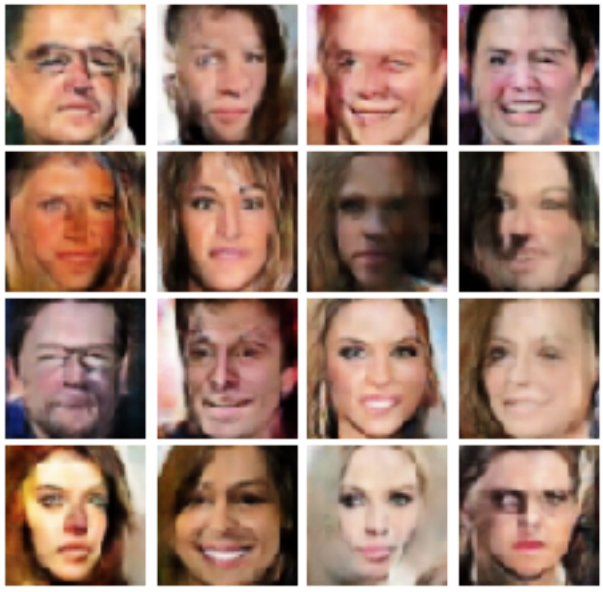
```
Iter: 9150, D: 0.2483, G:4.04
```



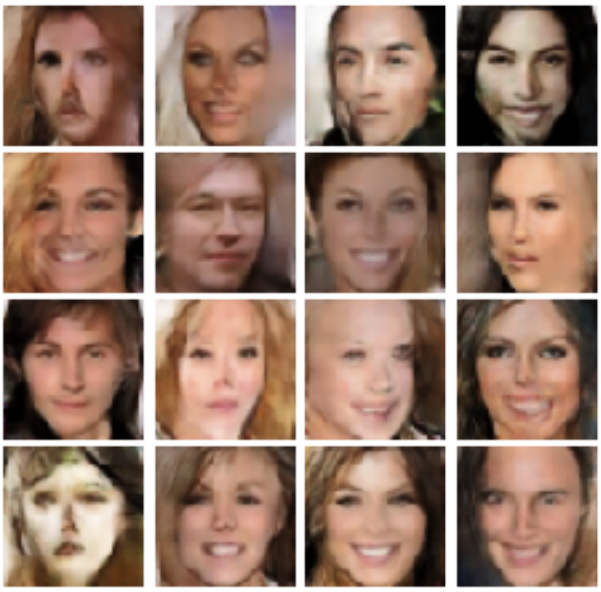
```
Iter: 9300, D: 0.1272, G:4.391
```

```
Iter: 9450, D: 0.2753, G:4.427
```

Iter: 9600, D: 0.1888, G:2.2



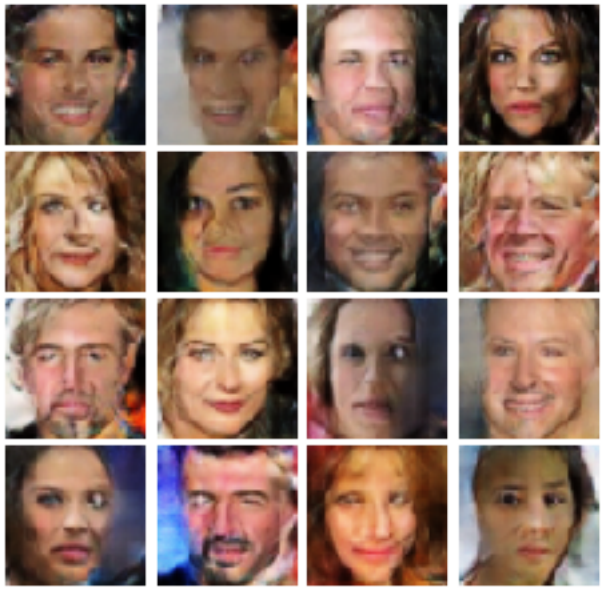
Iter: 9750, D: 0.3219, G:2.943



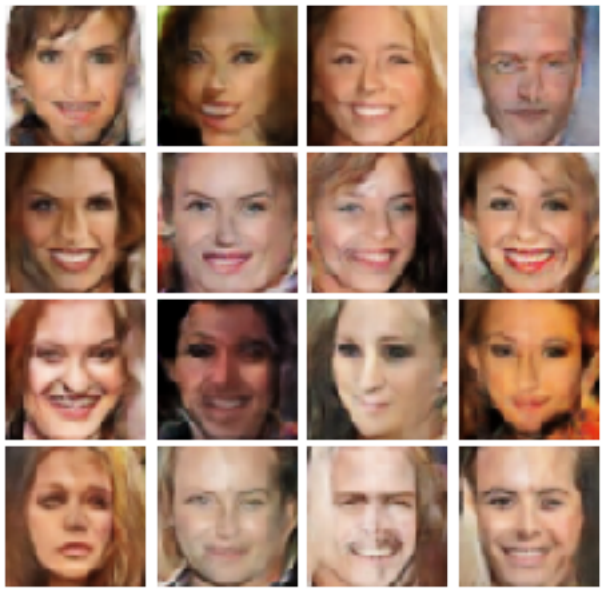
```
EPOCH:  11
Iter: 9900, D: 0.1172, G:4.978
```

```
Iter: 10050, D: 0.2626, G:3.738
```

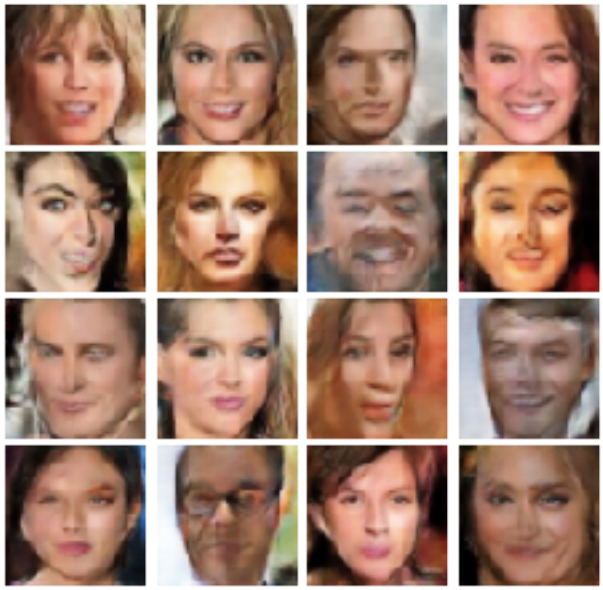
Iter: 10200, D: 0.1627, G:3.953



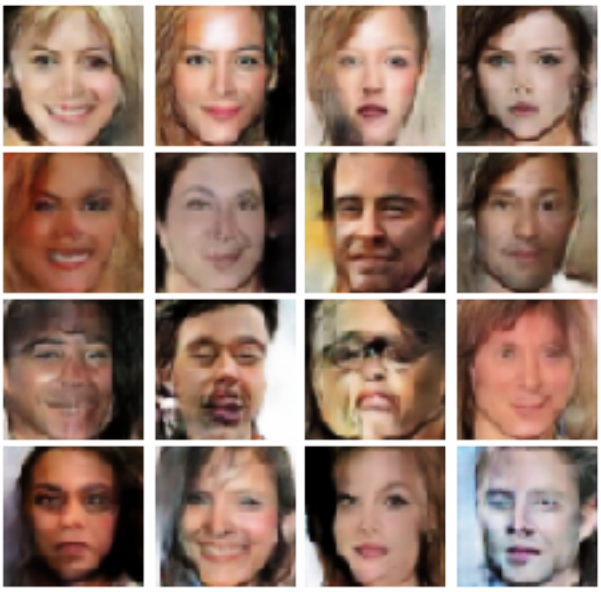
Iter: 10350, D: 5.619, G:5.265



Iter: 10500, D: 0.1802, G:4.525

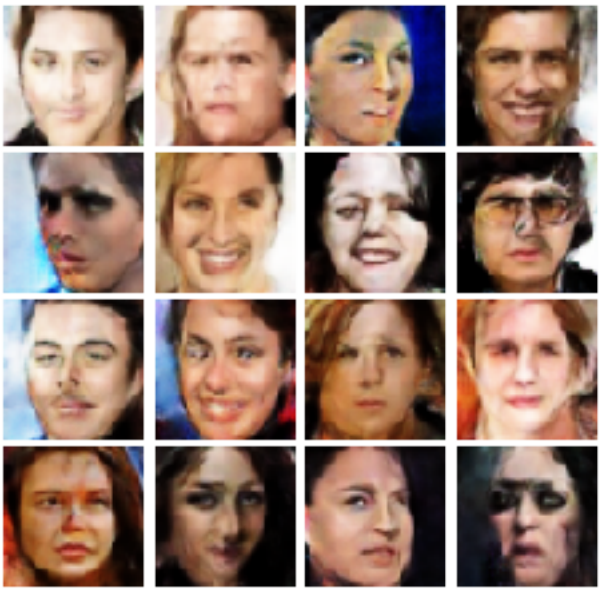


Iter: 10650, D: 0.1819, G:4.185

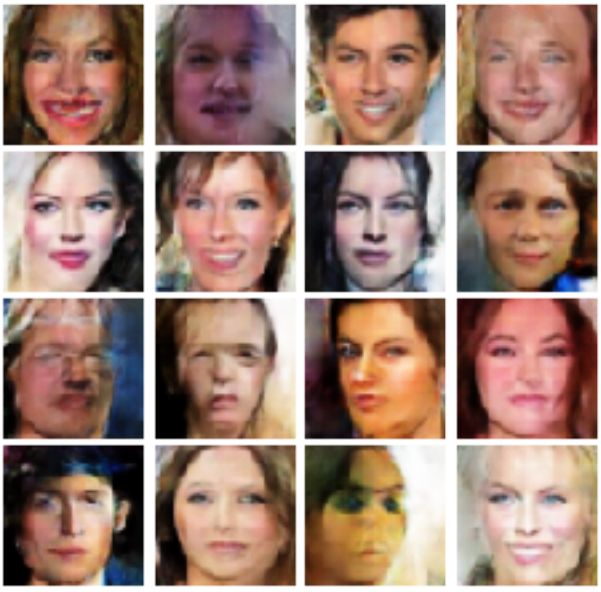


Iter: 10800, D: 0.04951, G:4.377

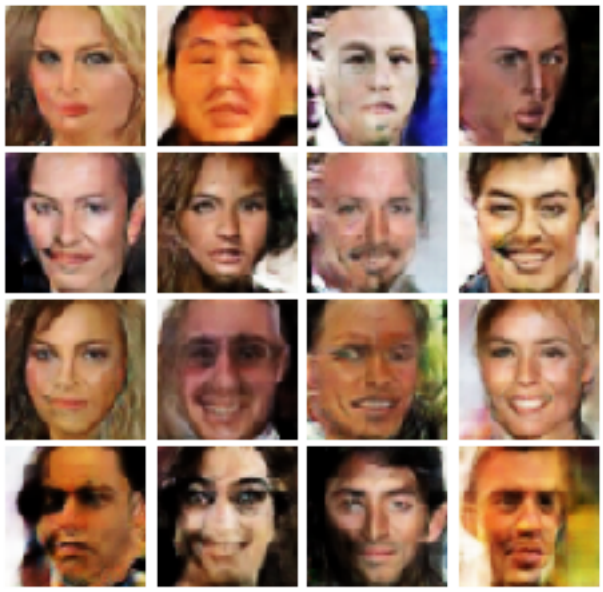
EPOCH: 12
Iter: 10950, D: 0.05802, G:1.892



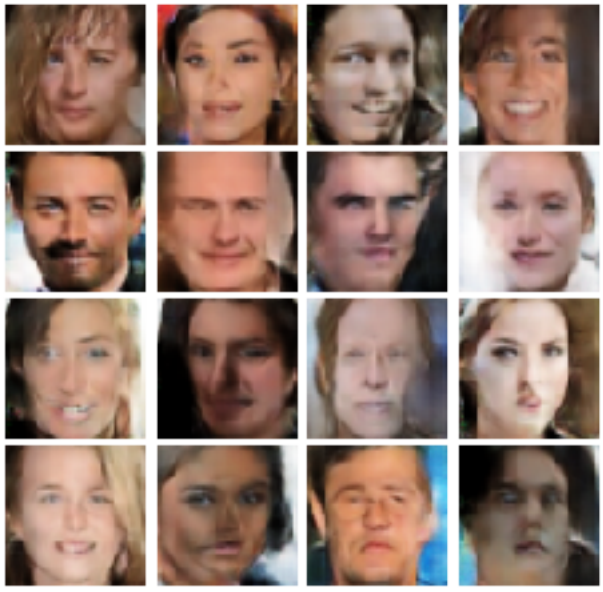
Iter: 11100, D: 0.2046, G:3.385



Iter: 11250, D: 0.1171, G:3.382

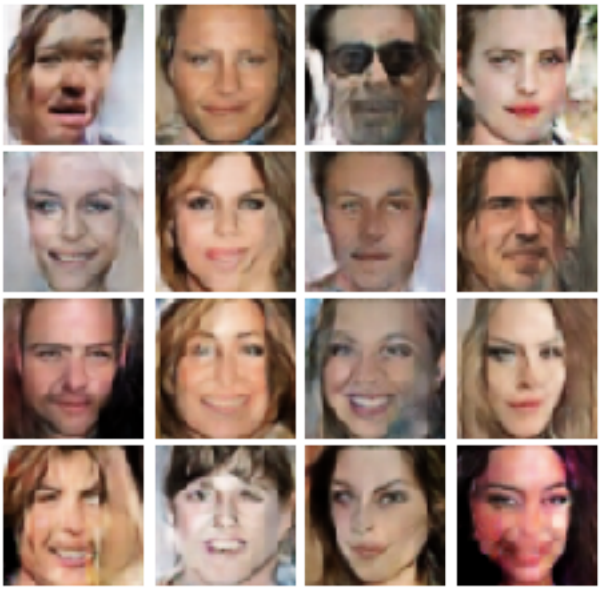


Iter: 11400, D: 0.3769, G:5.405

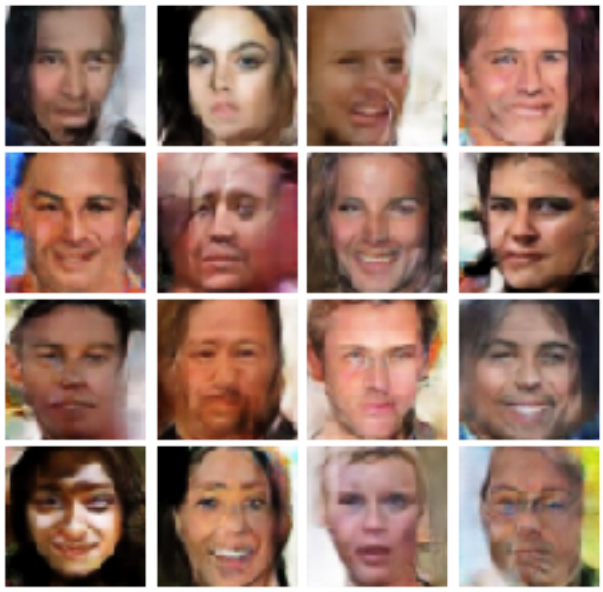


Iter: 11550, D: 0.09926, G:4.508

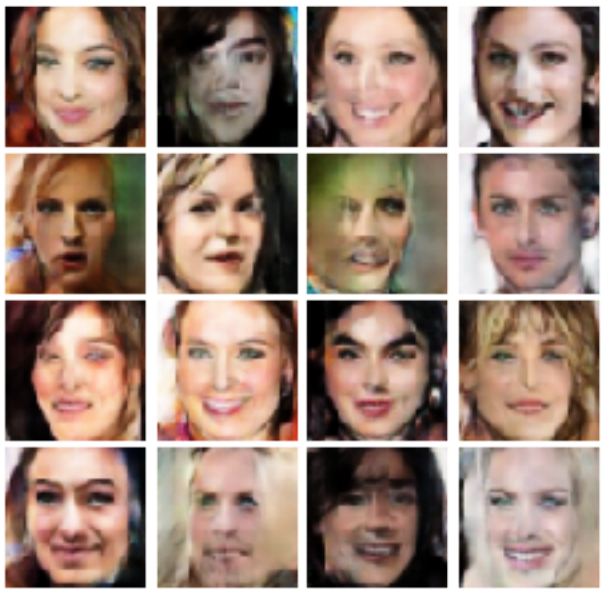
Iter: 11700, D: 0.1162, G:2.884



EPOCH: 13
Iter: 11850, D: 0.1642, G:5.728

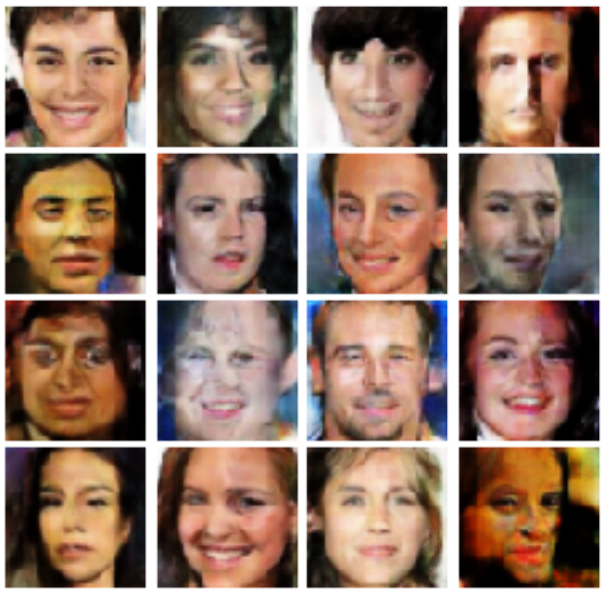


Iter: 12000, D: 0.1193, G:5.133

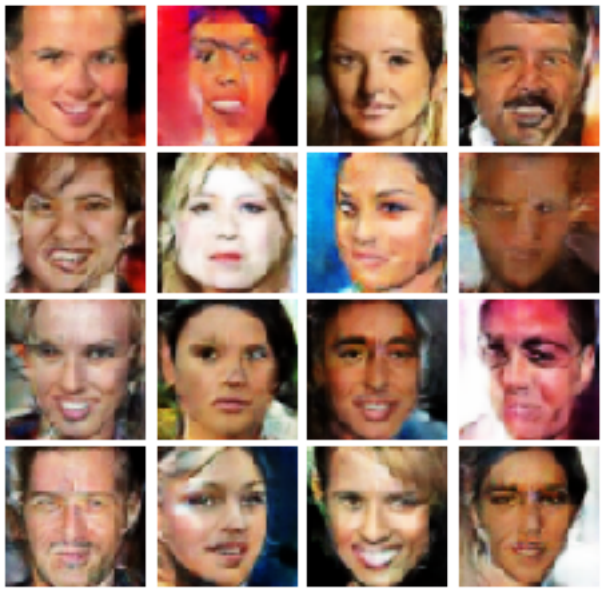


Iter: 12150, D: 0.2907, G:3.402

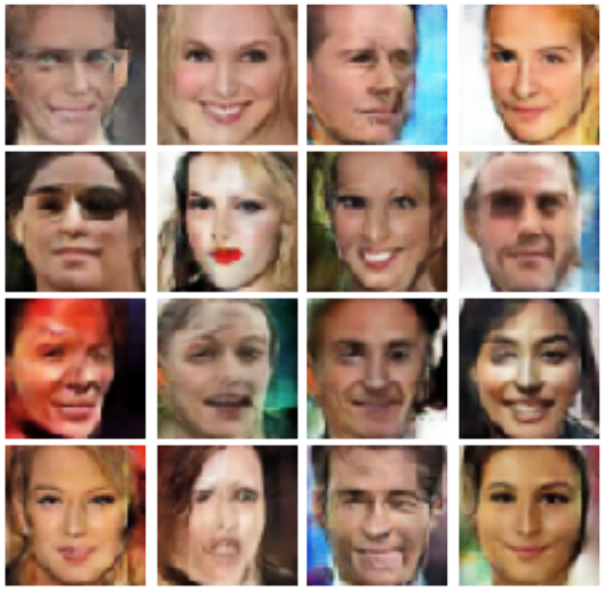
Iter: 12300, D: 0.1085, G:5.191



Iter: 12450, D: 0.1214, G:6.515

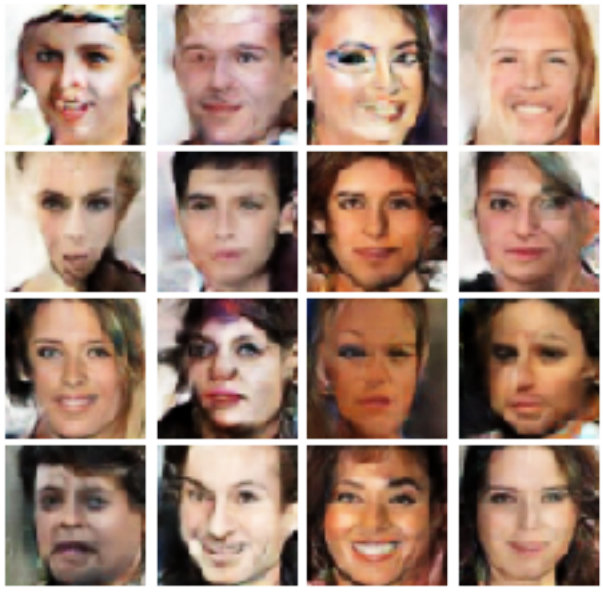


Iter: 12600, D: 0.7718, G:4.171

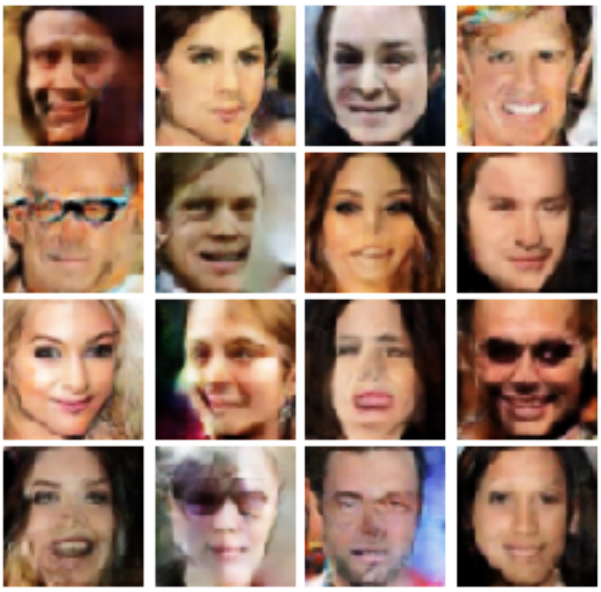


Iter: 12750, D: 1.016, G:9.094

```
EPOCH:  14
Iter: 12900, D: 0.05554, G:4.271
```

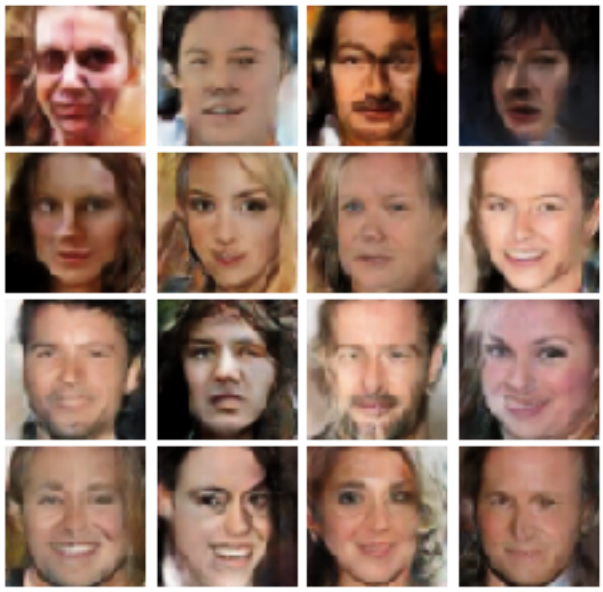


```
Iter: 13050, D: 1.605, G:1.306
```



Iter: 13200, D: 0.2565, G:4.921

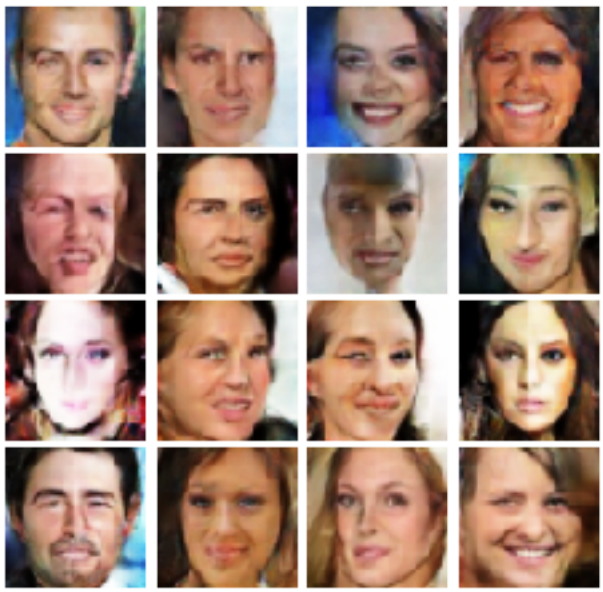
Iter: 13350, D: 0.9365, G:3.736



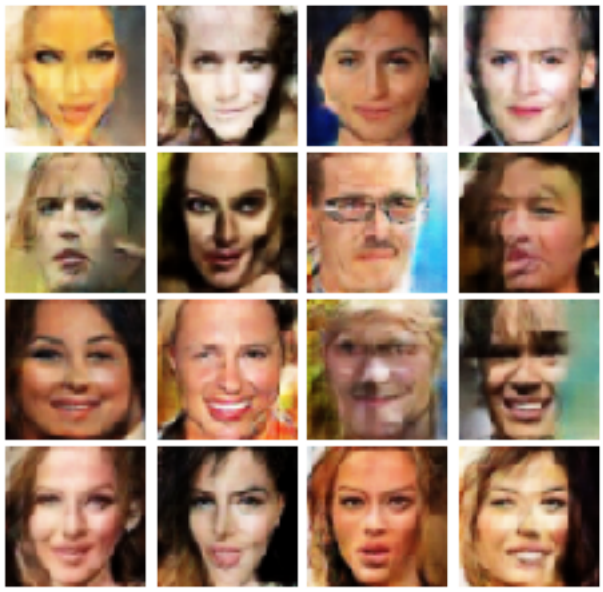
Iter: 13500, D: 0.3221, G:2.356

Iter: 13650, D: 0.1428, G:3.159

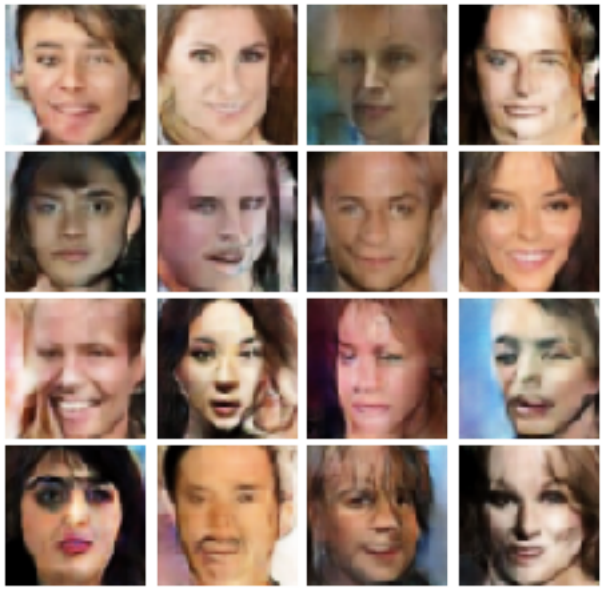
Iter: 13800, D: 0.2145, G:5.252



EPOCH: 15
Iter: 13950, D: 0.353, G:3.559

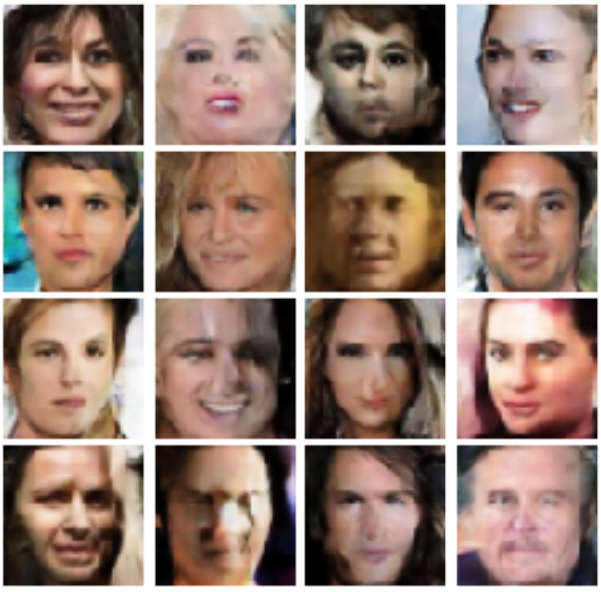


Iter: 14100, D: 0.2326, G:5.819

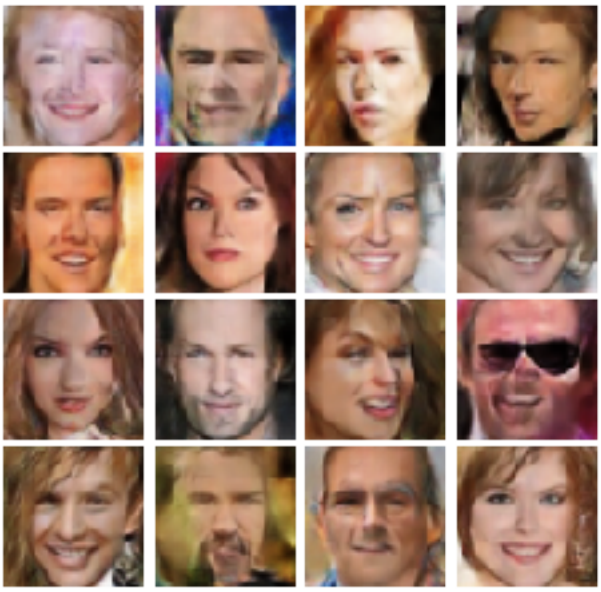


Iter: 14250, D: 0.1462, G:4.458

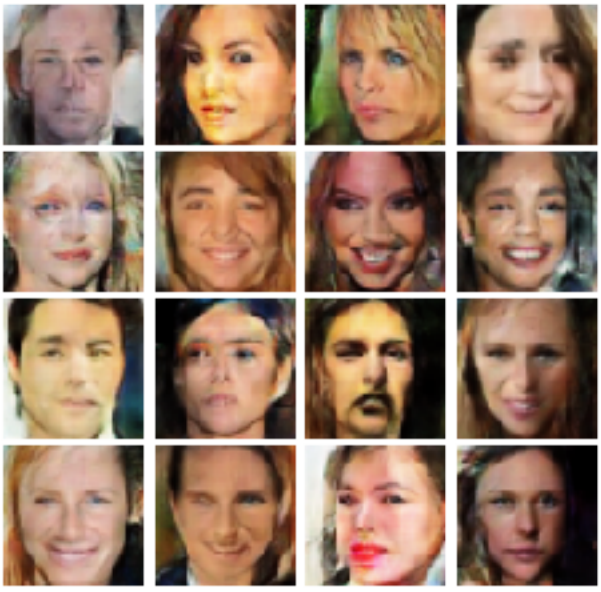
Iter: 14400, D: 1.052, G:1.516



Iter: 14550, D: 0.2075, G:3.163



Iter: 14700, D: 0.3034, G:3.74



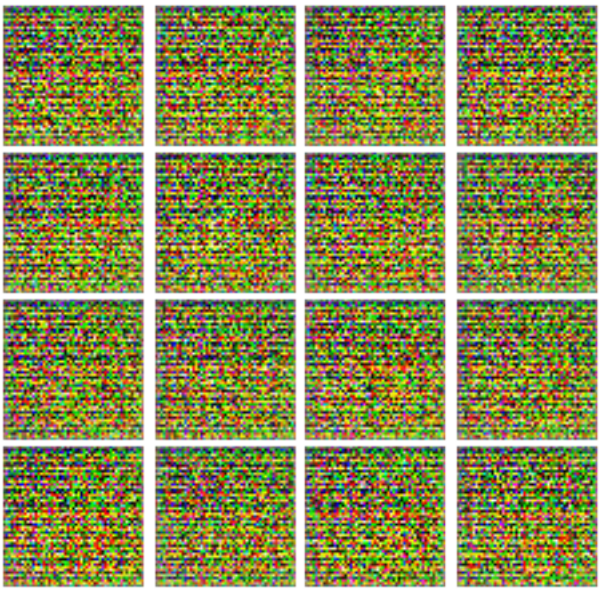
5.0.2 Train LS-GAN

[14]:
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

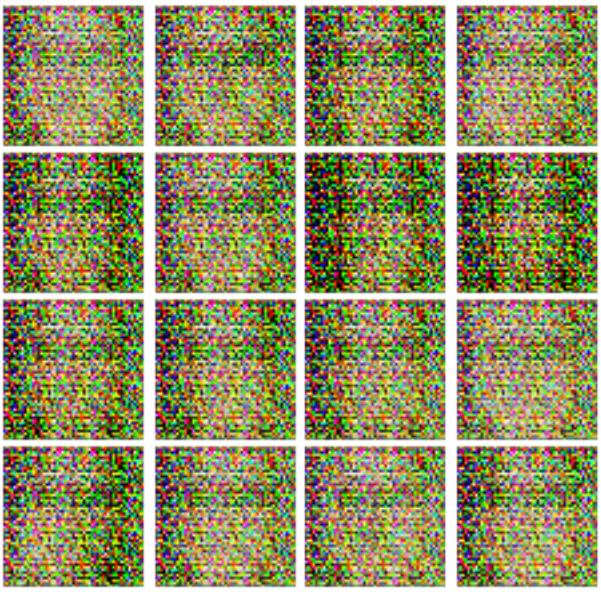
[15]:
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.
↪999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.
↪999))
```

[16]:
```
# ls-gan
train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
          ls_generator_loss, num_epochs=NUM_EPOCHS, show_every=200,
          train_loader=celeba_loader_train, device=device)
```

```
EPOCH:  1
Iter: 0, D: 1.126, G:21.14
```



```
Iter: 200, D: 0.08874, G:0.2953
```



Iter: 400, D: 0.1626, G:0.128

Iter: 600, D: 0.09781, G:0.7947

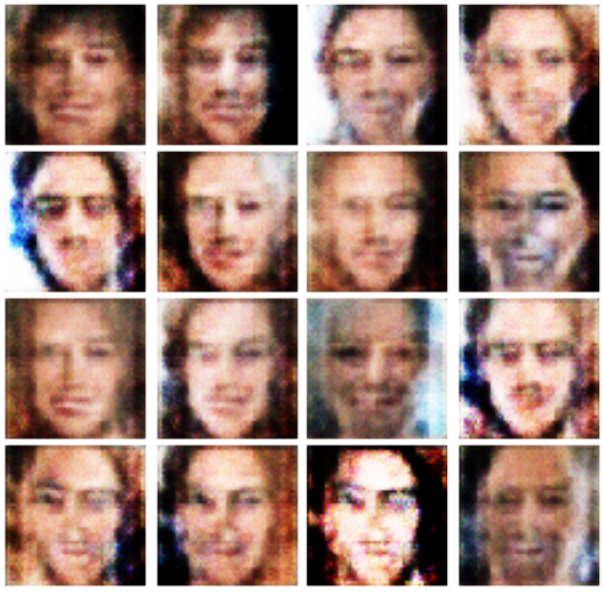
```
Iter: 800, D: 0.1686, G:0.7874
```

```
EPOCH:  2
Iter: 1000, D: 0.2602, G:0.6705
```

```
Iter: 1200, D: 0.2366, G:0.1351
```

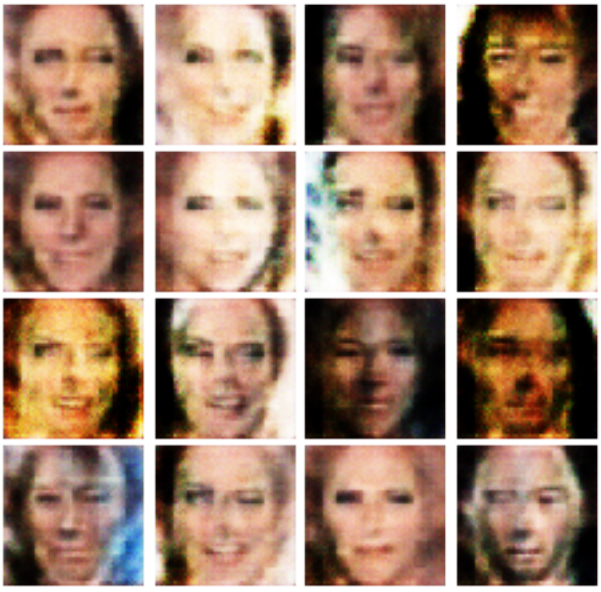
```
Iter: 1400, D: 0.1179, G:0.2623
```

```
Iter: 1600, D: 0.1137, G:0.1889
```

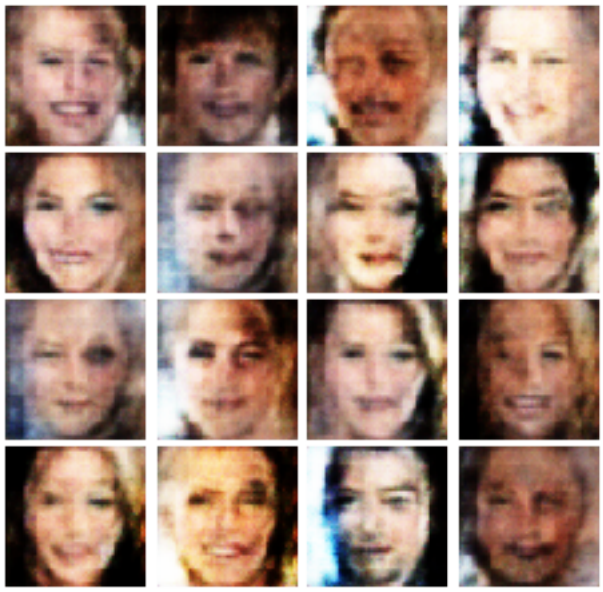


```
Iter: 1800, D: 0.2541, G:0.2395
```

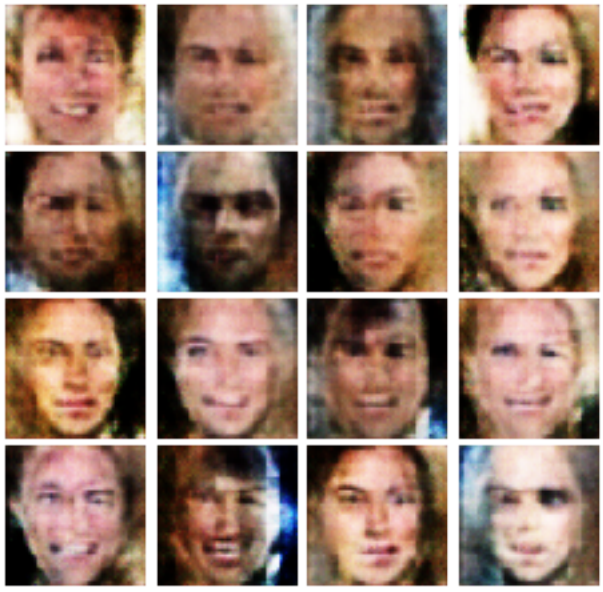
```
EPOCH:  3
Iter: 2000, D: 0.2508, G:0.3151
```



```
Iter: 2200, D: 0.1136, G:0.4612
```



```
Iter: 2400, D: 0.4136, G:0.8757
```

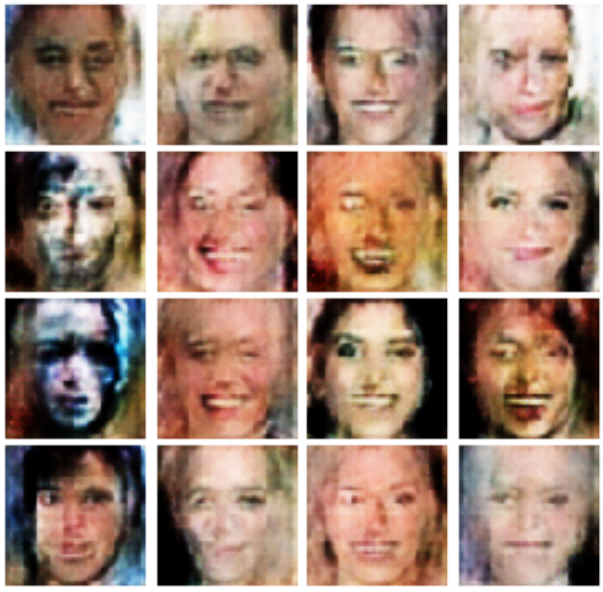


```
Iter: 2600, D: 0.1284, G:0.2979
```

```
Iter: 2800, D: 0.4212, G:0.4918
```

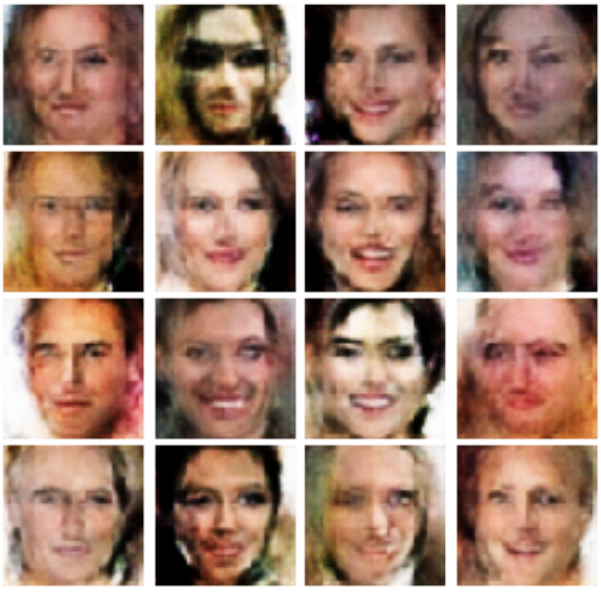
```
EPOCH:  4
Iter: 3000, D: 0.4043, G:0.08354
```

Iter: 3200, D: 0.1756, G:0.33



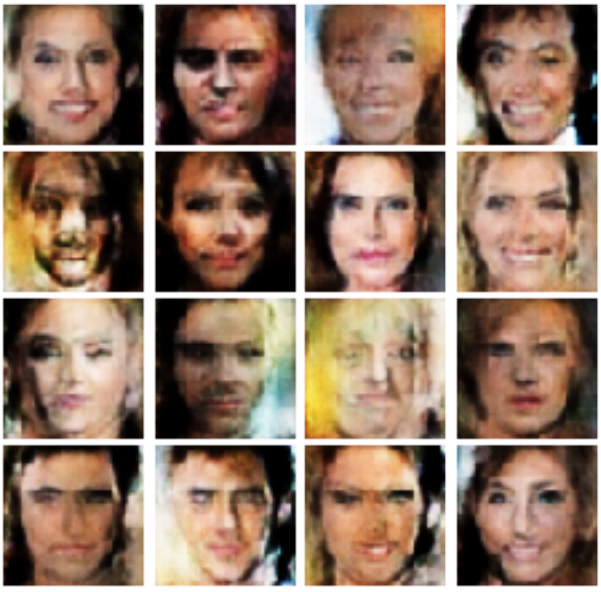
Iter: 3400, D: 0.356, G:1.239

```
Iter: 3600, D: 0.2411, G:0.2179
```

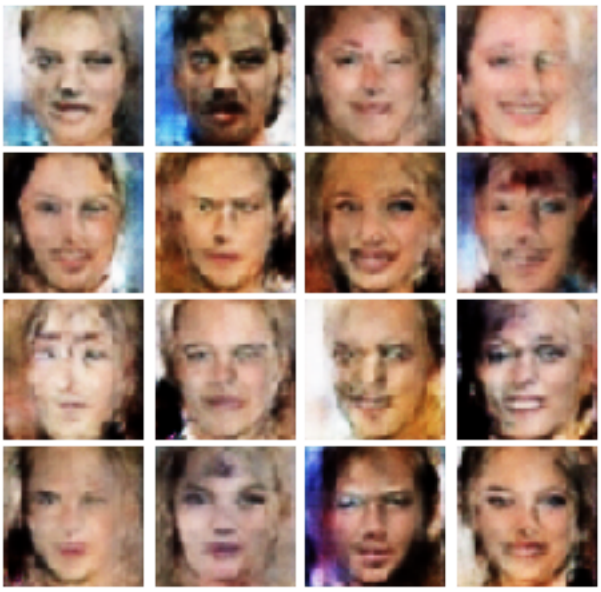


```
Iter: 3800, D: 0.1399, G:0.3887
```

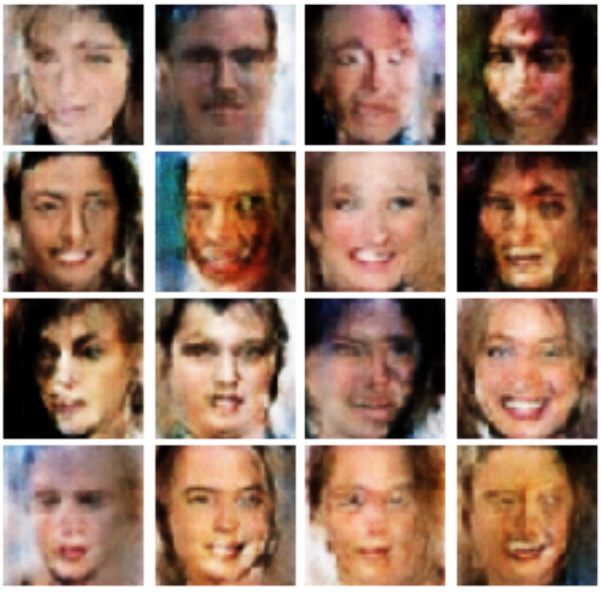
```
EPOCH:  5
Iter: 4000, D: 0.1381, G:0.3465
```



```
Iter: 4200, D: 0.1666, G:0.3524
```

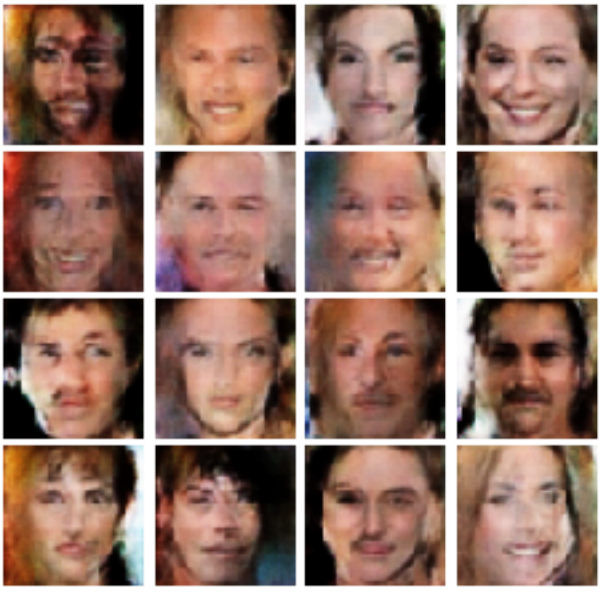


```
Iter: 4400, D: 0.1949, G:0.474
```



```
Iter: 4600, D: 0.4316, G:0.3245
```

```
Iter: 4800, D: 0.1164, G:0.7428
```



```
EPOCH:  6
Iter: 5000, D: 0.04333, G:0.4991
```

Iter: 5200, D: 0.1848, G:0.2321

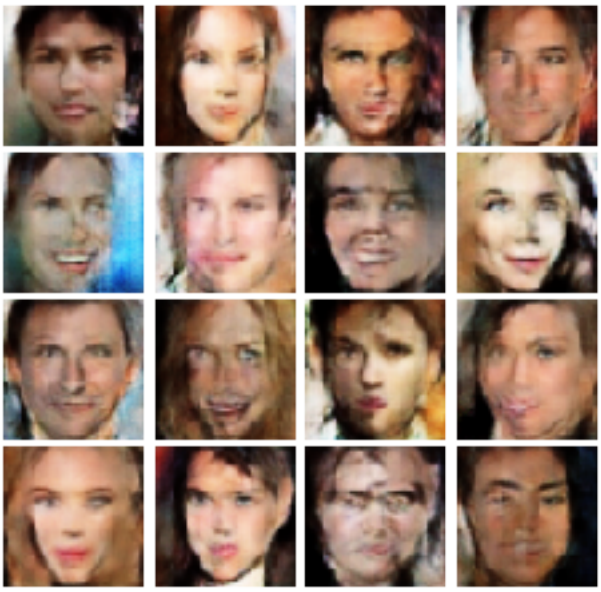
Iter: 5400, D: 0.3831, G:0.2178

Iter: 5600, D: 0.1606, G:0.2322

Iter: 5800, D: 0.09249, G:0.3506

```
EPOCH:  7
Iter: 6000, D: 0.1816, G:0.5065
```

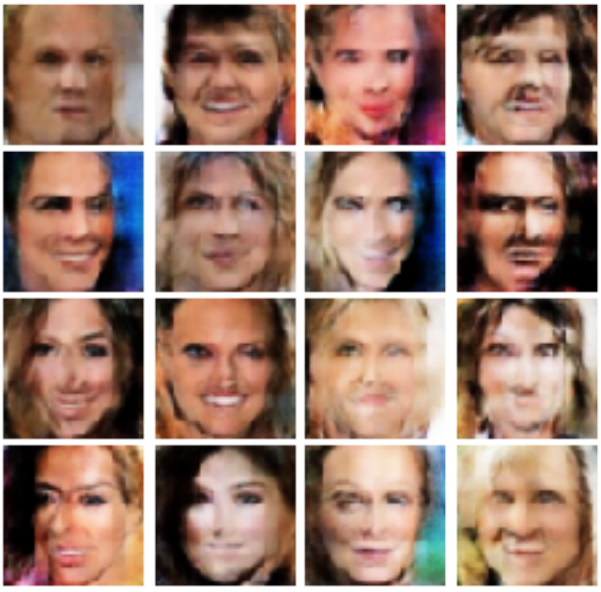
```
Iter: 6200, D: 0.1331, G:0.2833
```



Iter: 6400, D: 0.2624, G:0.8327

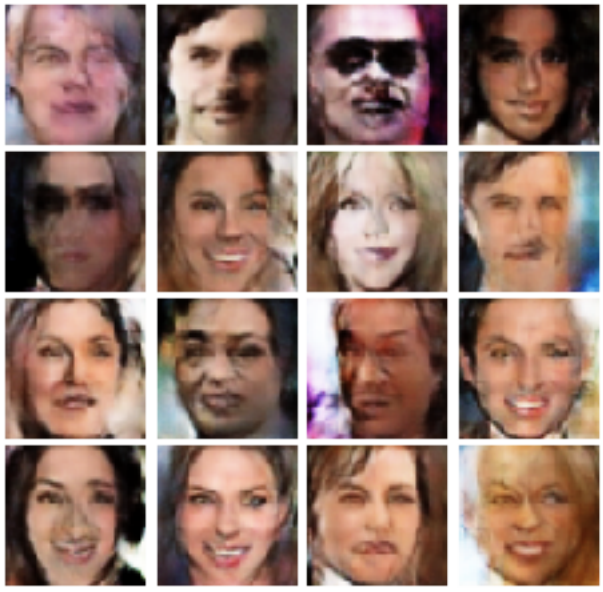
Iter: 6600, D: 0.192, G:0.4704

Iter: 6800, D: 0.0617, G:0.4988

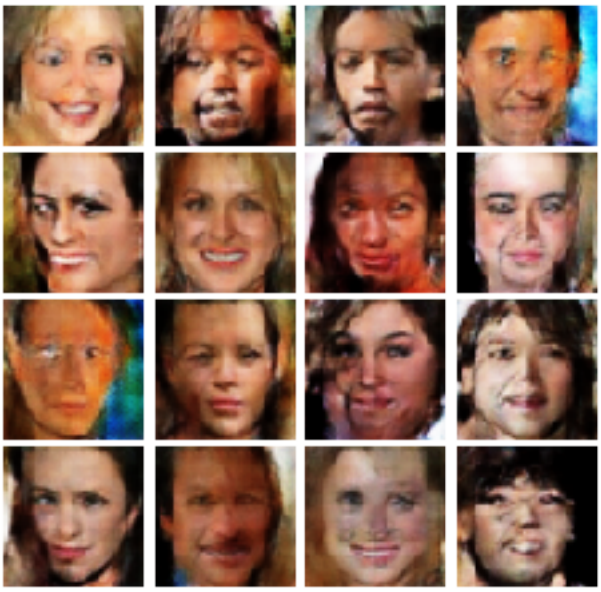


EPOCH: 8
Iter: 7000, D: 0.108, G:0.3278

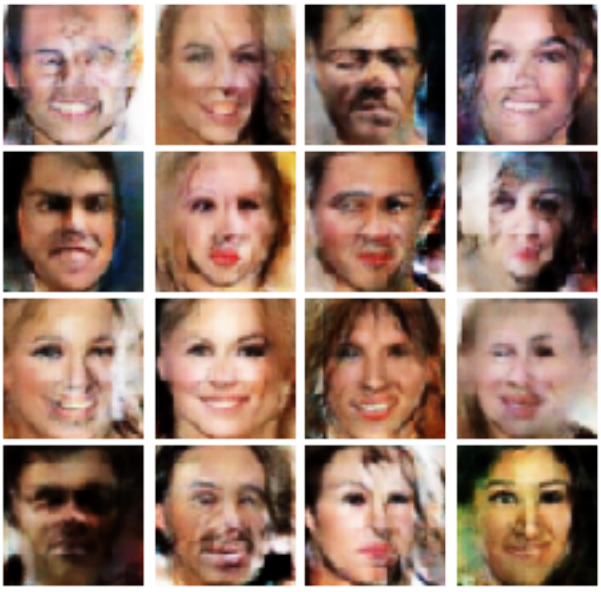
Iter: 7200, D: 0.116, G:0.3003



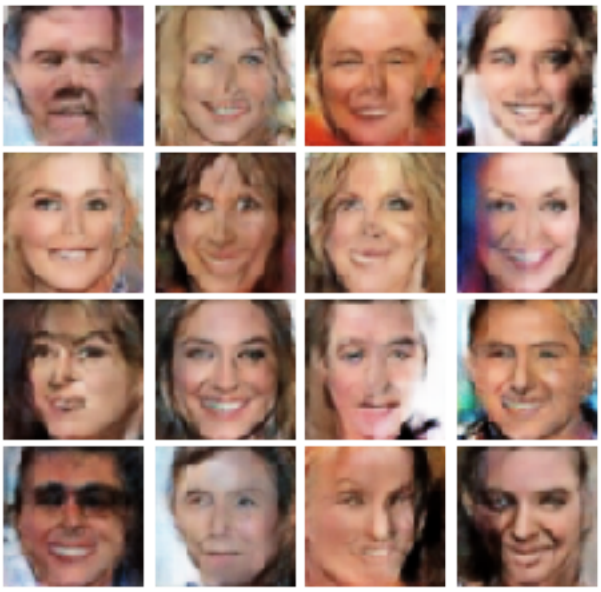
Iter: 7400, D: 0.1847, G:0.3113



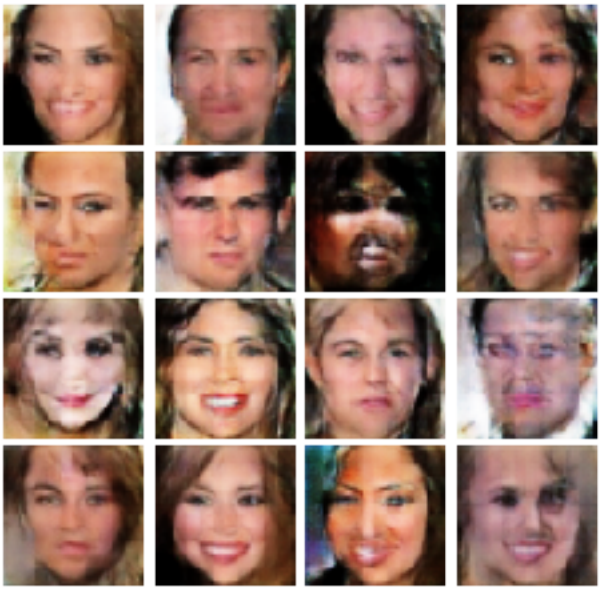
Iter: 7600, D: 0.1174, G:0.4725



Iter: 7800, D: 0.1766, G:0.5269



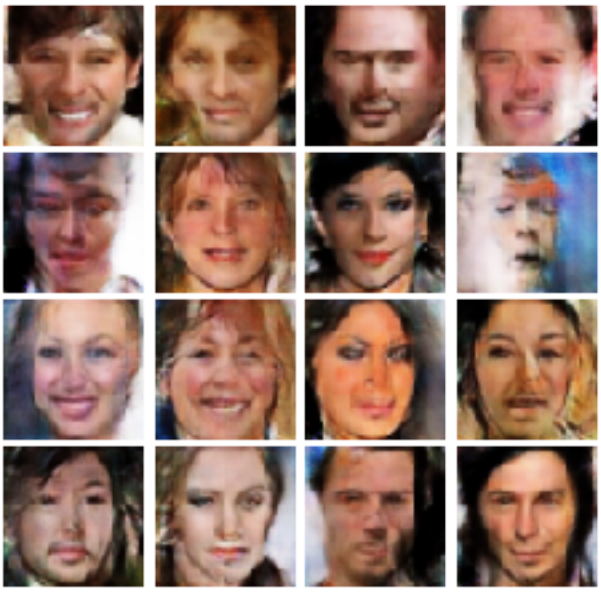
```
EPOCH:  9
Iter: 8000, D: 0.3225, G:1.316
```



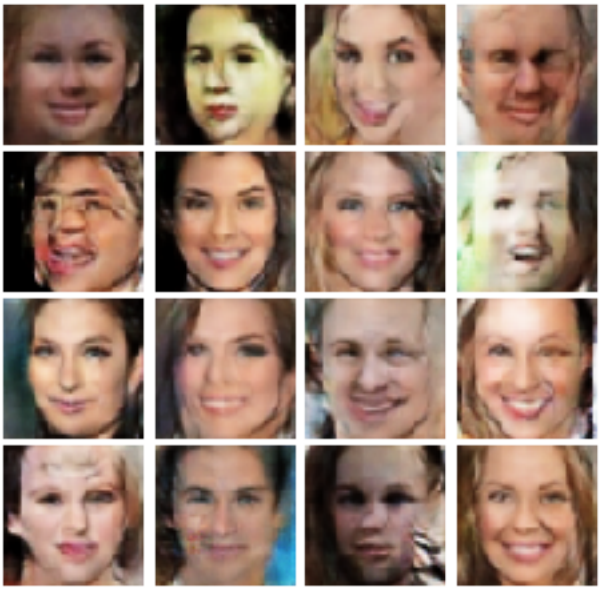
```
Iter: 8200, D: 0.06168, G:0.3453
```

Iter: 8400, D: 0.07306, G:0.5548

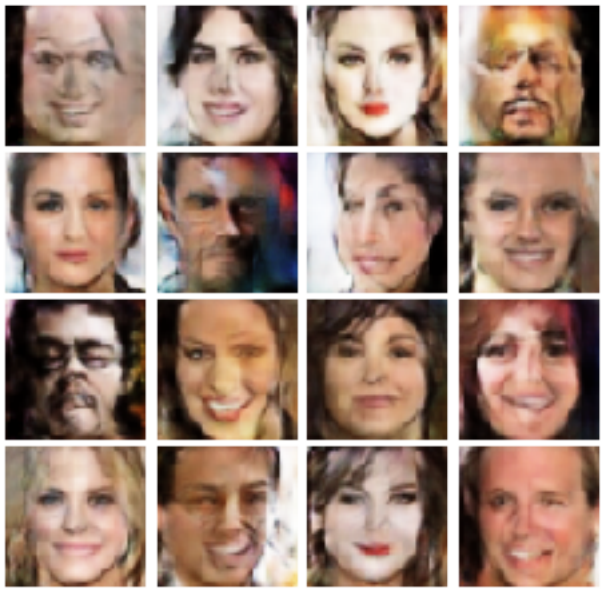
Iter: 8600, D: 0.2378, G:0.3291



```
Iter: 8800, D: 0.08436, G:0.5598
```

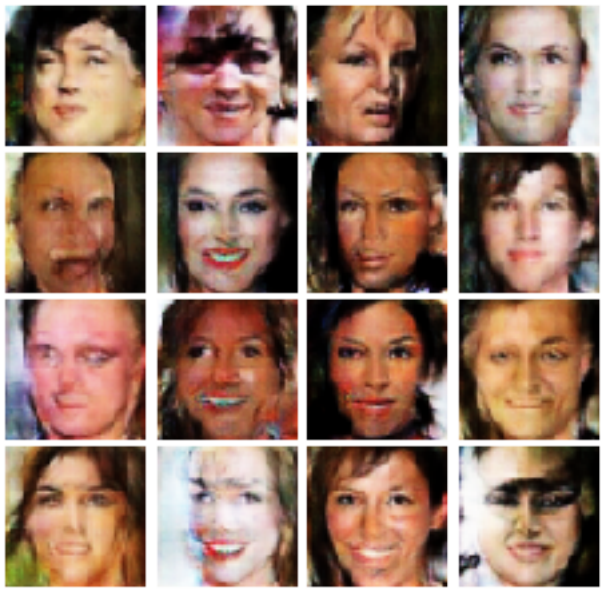


```
EPOCH:  10
Iter: 9000, D: 0.06918, G:0.3911
```

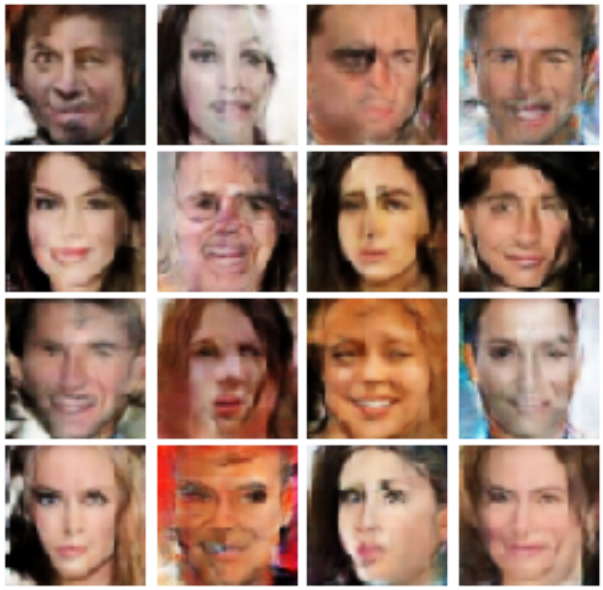


Iter: 9200, D: 0.05487, G:0.5438

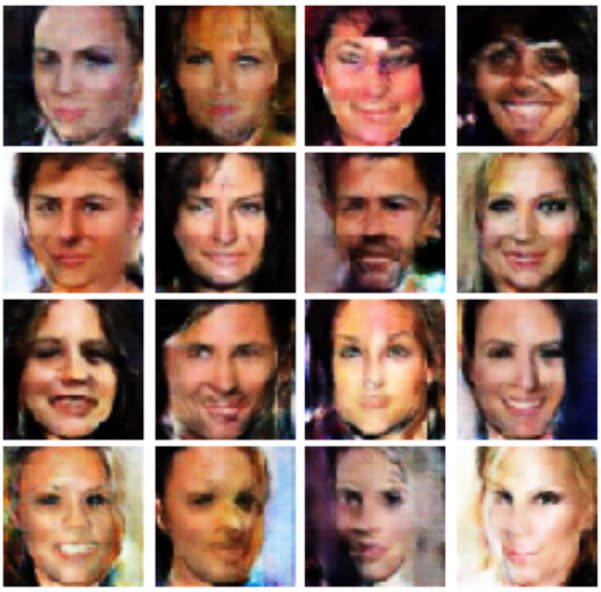
Iter: 9400, D: 0.09616, G:0.76



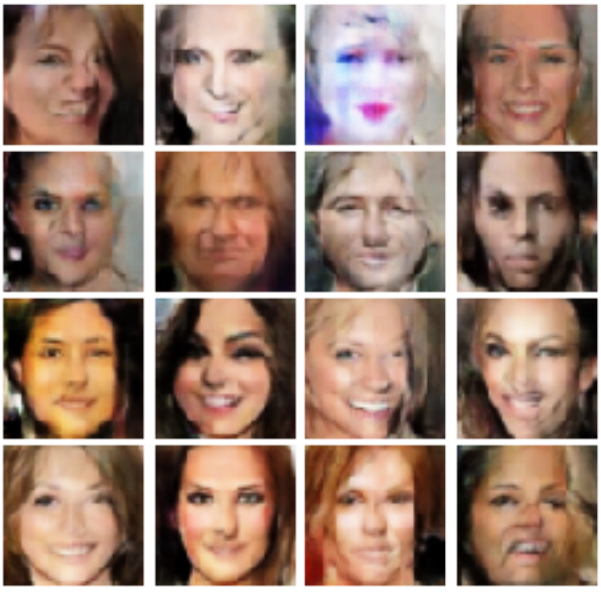
Iter: 9600, D: 0.1348, G:0.578



Iter: 9800, D: 0.05261, G:0.7094



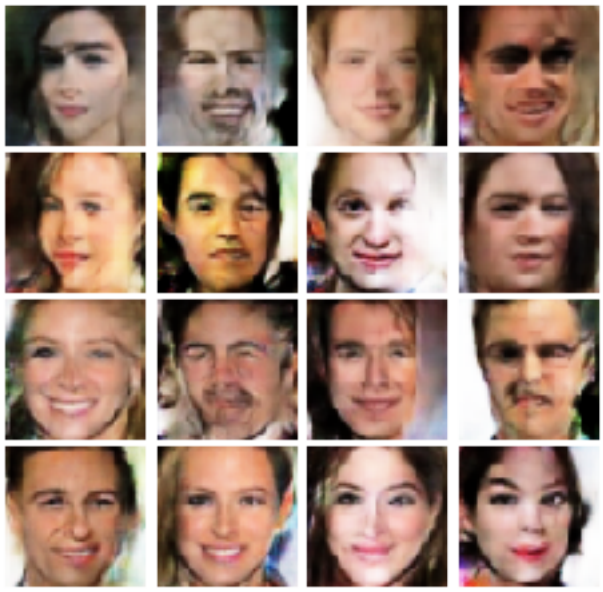
```
EPOCH:  11
Iter: 10000, D: 0.08958, G:0.4279
```



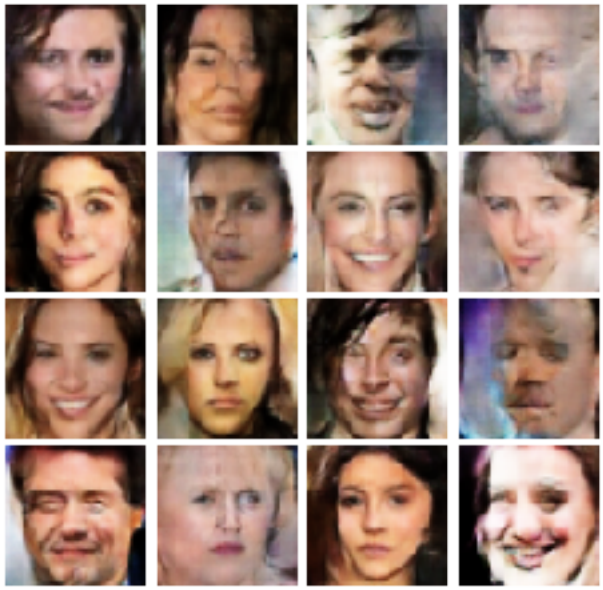
```
Iter: 10200, D: 0.3837, G:0.09665
```

Iter: 10400, D: 0.08025, G:0.4032

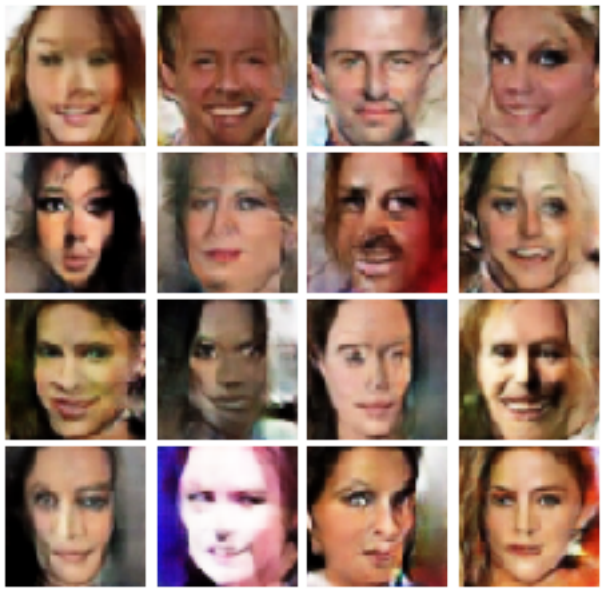
Iter: 10600, D: 0.156, G:0.3459



Iter: 10800, D: 0.1653, G:0.389

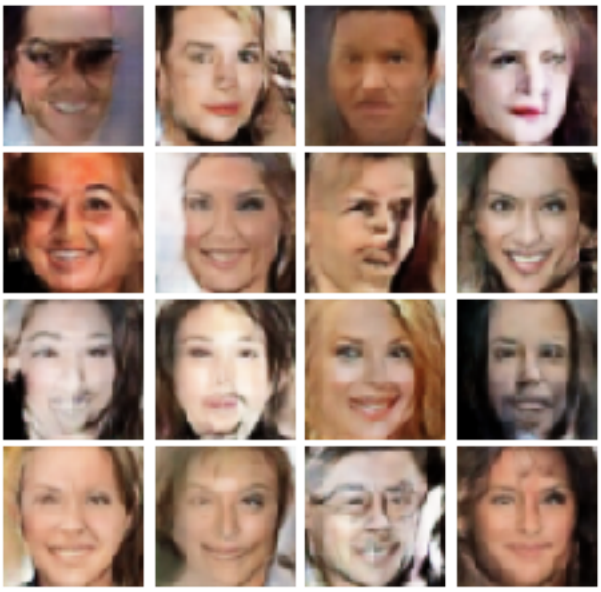


EPOCH: 12
Iter: 11000, D: 0.06717, G:0.3078

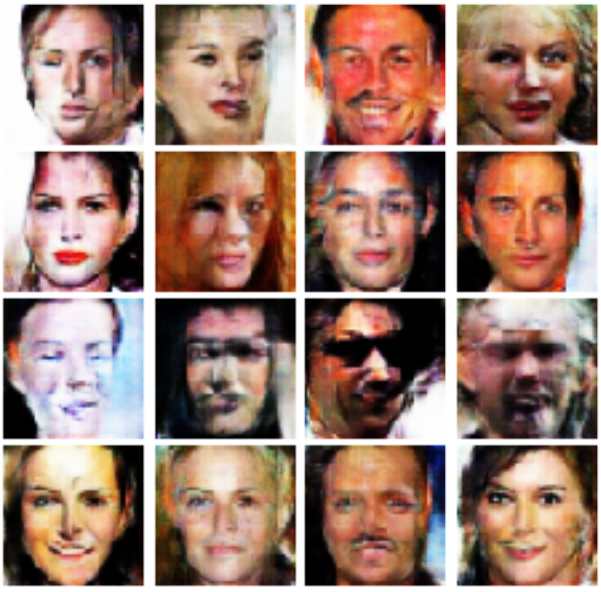


Iter: 11200, D: 0.1218, G:0.2829

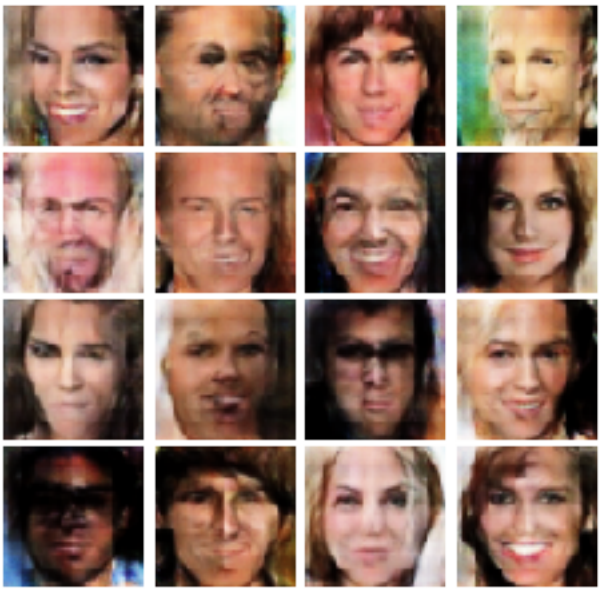
Iter: 11400, D: 0.03085, G:0.3041



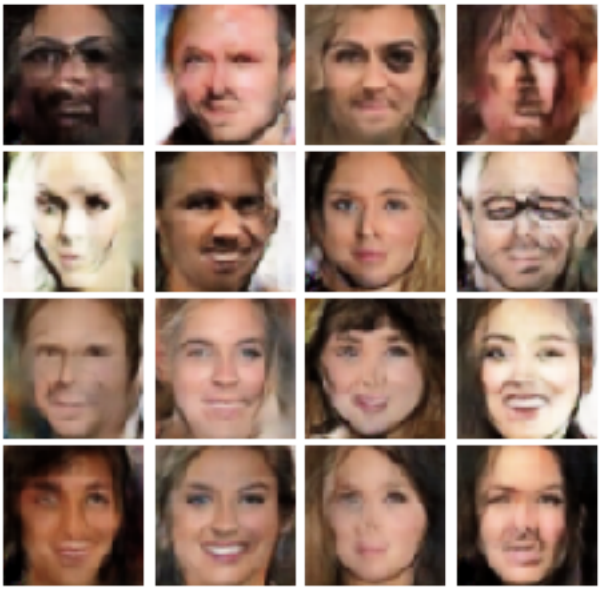
Iter: 11600, D: 0.02614, G:0.3798



Iter: 11800, D: 0.2892, G:0.2163

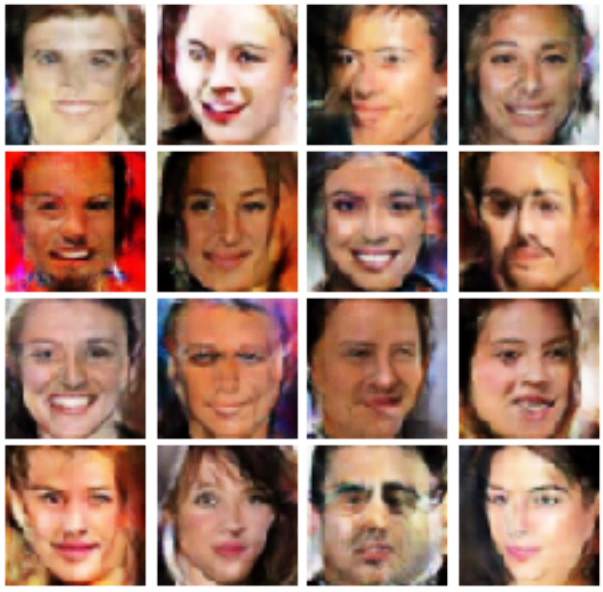


```
EPOCH:  13
Iter: 12000, D: 0.1454, G:0.3974
```

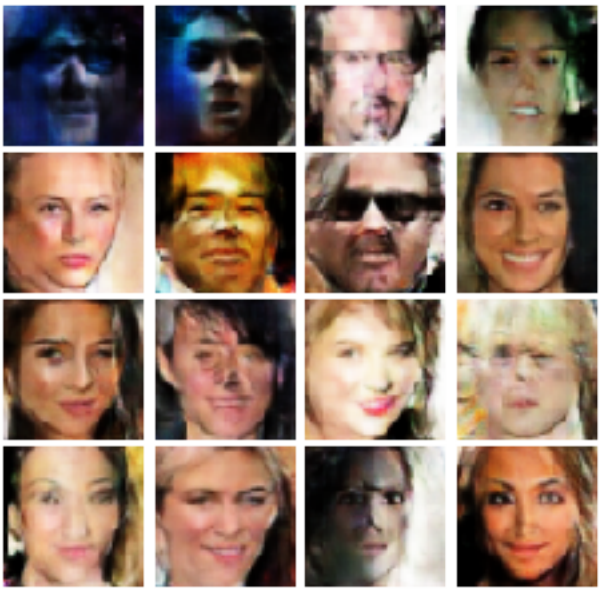


```
Iter: 12200, D: 0.1024, G:0.4802
```

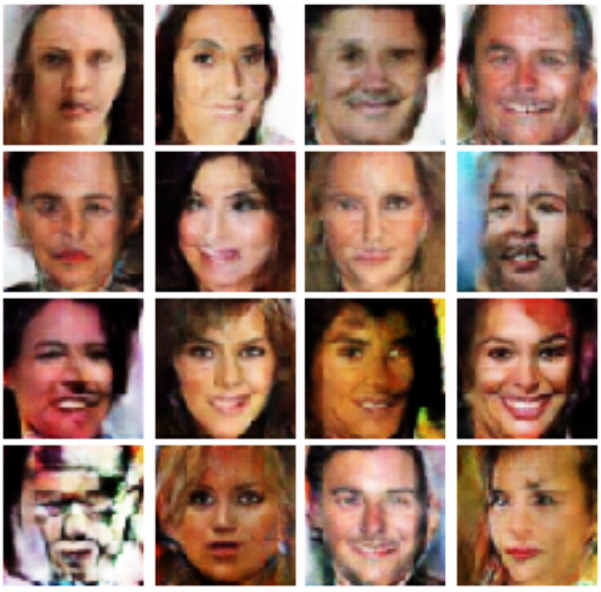
Iter: 12400, D: 0.07528, G:0.4849



Iter: 12600, D: 0.0866, G:0.3555

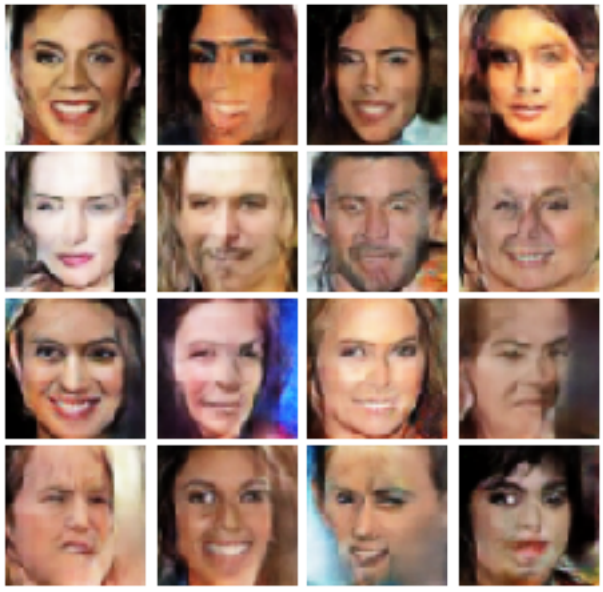


```
Iter: 12800, D: 0.1109, G:0.3726
```



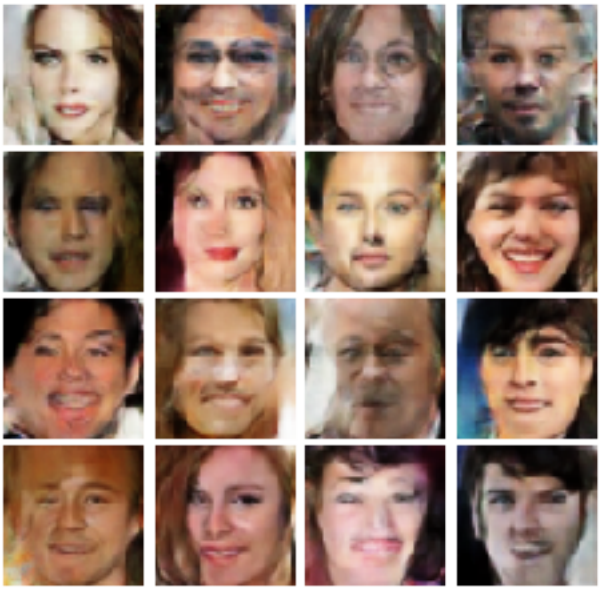
```
EPOCH:  14
Iter: 13000, D: 0.07298, G:0.5951
```

Iter: 13200, D: 0.08095, G:0.2742

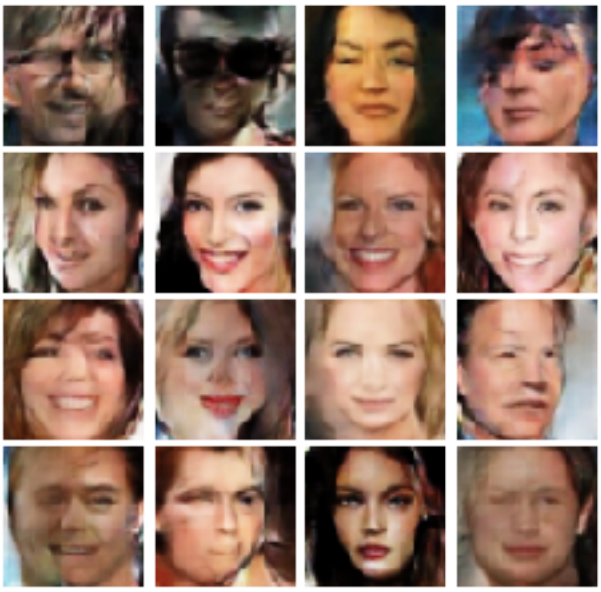


Iter: 13400, D: 0.05349, G:0.2711

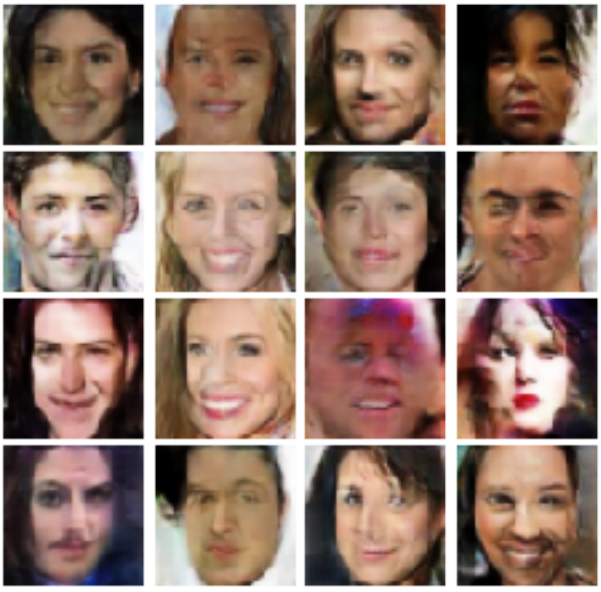
```
Iter: 13600, D: 0.07959, G:0.3187
```



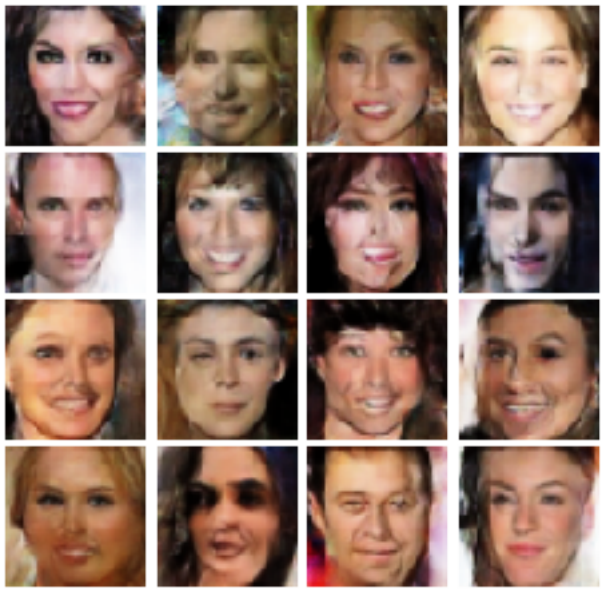
```
Iter: 13800, D: 0.09426, G:0.4375
```



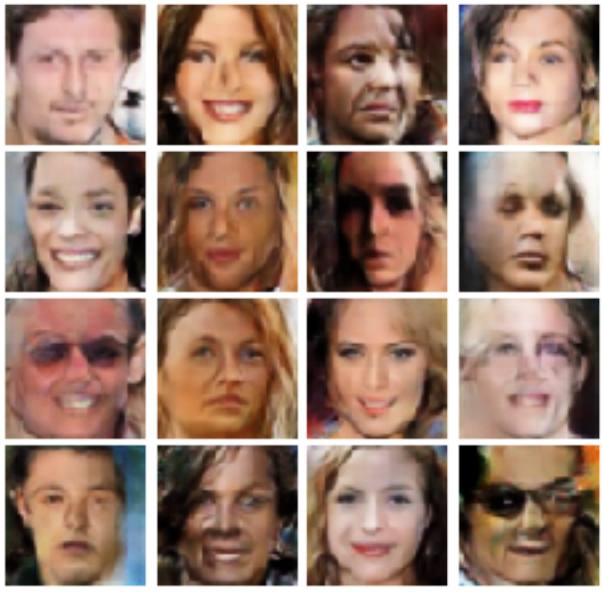
EPOCH: 15
Iter: 14000, D: 0.07465, G:0.6256



Iter: 14200, D: 0.1239, G:0.5692



Iter: 14400, D: 0.04065, G:0.2812



Iter: 14600, D: 0.03472, G:0.341