MP4

April 20, 2023

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 a dataset of cat face images.

[1]:
```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
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 "cpu")
```

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 $\log(1 - D(G(z)))$, 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

Use padding = 1 (not 0) for all the convolutional layers.

Instead of Relu we LeakyReLu throughout the discriminator (we use a negative slope value of 0.2). You can use simply use relu as well.

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]:
```
from gan.models import Discriminator, Generator
```

4 Data loading

The cat images we provide are RGB images with a resolution of 64x64. In order to prevent our discriminator from overfitting, we will need to perform some data augmentation.

TODO: Implement data augmentation by adding new transforms to the cell below. At the minimum, you should have a RandomCrop and a ColorJitter, but we encourage you to experiment with different augmentations to see how the performance of the GAN changes. See https://pytorch.org/vision/stable/transforms.html.

[ ]:
```
#!sh download_cat.sh
```

[ ]:
```
batch_size = 32
imsize = 64
```

```
cat_root = './cats'

cat_train = ImageFolder(root=cat_root, transform=transforms.Compose([
  transforms.ToTensor(),

  # Example use of RandomCrop:
  transforms.Resize(int(1.15 * imsize)),
  transforms.RandomCrop(imsize),
  transforms.ColorJitter(brightness=0.1, contrast=0.3, saturation=0.4, hue=0.
 ↪05), #Color Jitter
  #transforms.RandomAffine(degrees=15, translate=(0.1, 0.1), scale=(0.9, 1.1),␣
 ↪shear=10),
  transforms.GaussianBlur(kernel_size=3),
]))

cat_loader_train = DataLoader(cat_train, batch_size=batch_size, drop_last=True)
```

4.0.1 Visualize dataset

[ ]:
```
from gan.utils import show_images
imgs = next(cat_loader_train.__iter__())[0].numpy().squeeze()
show_images(imgs, color=True)
```

5 Training

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

[ ]:
```
# NOISE_DIM = 100
# NUM_EPOCHS = 50
# learning_rate = 0.001
```

5.0.1 Train GAN

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

[ ]:
```
# 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))
```

[ ]:
```
# # original gan
# train(D, G, D_optimizer, G_optimizer, discriminator_loss,
#           generator_loss, num_epochs=NUM_EPOCHS, show_every=1000,
#           batch_size=batch_size, train_loader=cat_loader_train, device=device)
```

5.0.2 Train LS-GAN

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

[ ]:
```
# D_optimizer = torch.optim.Adam(D.parameters(), lr=1e-3, betas = (0.5, 0.999))
# G_optimizer = torch.optim.Adam(G.parameters(), lr=1e-3, betas = (0.5, 0.999))
```

[ ]:
```
# # ls-gan
# train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
#           ls_generator_loss, num_epochs=NUM_EPOCHS, show_every=1000,
#           batch_size=batch_size, train_loader=cat_loader_train, device=device)
```

Extra Credit 1: Train Wasserstein GAN

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

[ ]:
```
# D_optimizer = torch.optim.Adam(D.parameters(), lr=5e-4, betas = (0.5, 0.999))
# G_optimizer = torch.optim.Adam(G.parameters(), lr=5e-4, betas = (0.5, 0.999))
```

[ ]:
```
# # Wassertein-gan
# from gan.losses import wass_discriminator_loss, wass_generator_loss
# train(D, G, D_optimizer, G_optimizer, wass_discriminator_loss,
#           wass_generator_loss, num_epochs=40, show_every=1000,
#           batch_size=batch_size, train_loader=cat_loader_train, device=device)
```

Extra Credit 2: Different Dataset - Celeb Faces

[7]:
```
batch_size = 128
imsize = 64
faces_root = './celeb'
faces_train = ImageFolder(root=faces_root, transform=transforms.Compose([
   transforms.ToTensor(),

   # Example use of RandomCrop:
    transforms.Resize(int(1.1 * imsize)),
    transforms.RandomCrop(imsize),
    transforms.ColorJitter(brightness=0.1, contrast=0.3, saturation=0.4, hue=0.
 ↪05),
    transforms.GaussianBlur(kernel_size=5)
 ]))

faces_loader_train = DataLoader(faces_train, batch_size=batch_size,␣
 ↪drop_last=True)
```

Visualize this Dataset

[8]:
```python
from gan.utils import show_images
imgs = next(faces_loader_train.__iter__())[0].numpy().squeeze()
show_images(imgs, color=True)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

Train with LS-GAN with Faces Dataset

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

[ ]:
```
# D_optimizer = torch.optim.Adam(D.parameters(), lr=1e-4, betas = (0.5, 0.999))
# G_optimizer = torch.optim.Adam(G.parameters(), lr=1e-4, betas = (0.5, 0.999))
```

[ ]:
```
# # ls-gan
# train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
#           ls_generator_loss, num_epochs=30, show_every=1000,
#           batch_size=batch_size, train_loader=faces_loader_train,␣
 ↪device=device)
```

Extra Credit 3: Training with LSGAN with Spectral Normalization on Cat Faces Dataset

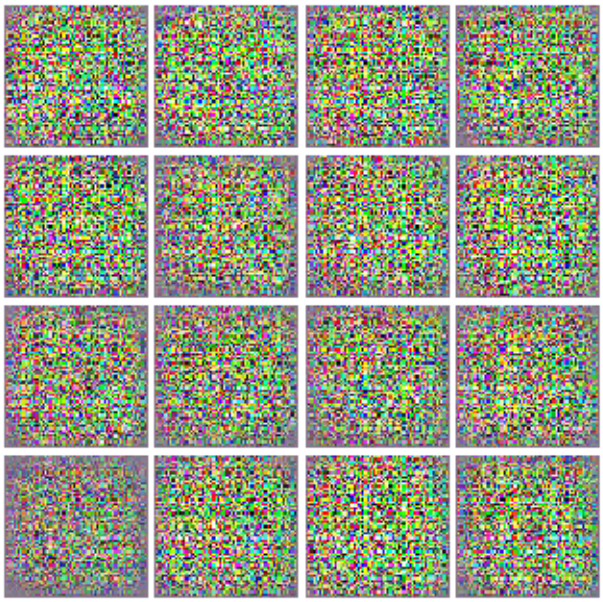
[9]:
```
NOISE_DIM = 100
```

[10]:
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

[11]:
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=1e-4, betas = (0.5, 0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=1e-4, betas = (0.5, 0.999))
```

[12]:
```
# ls-gan
train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
          ls_generator_loss, num_epochs=50, show_every=500,
          batch_size=batch_size, train_loader=faces_loader_train, device=device)
```

```
EPOCH:  1
Iter: 0, D: 0.5574, G:0.4724
```

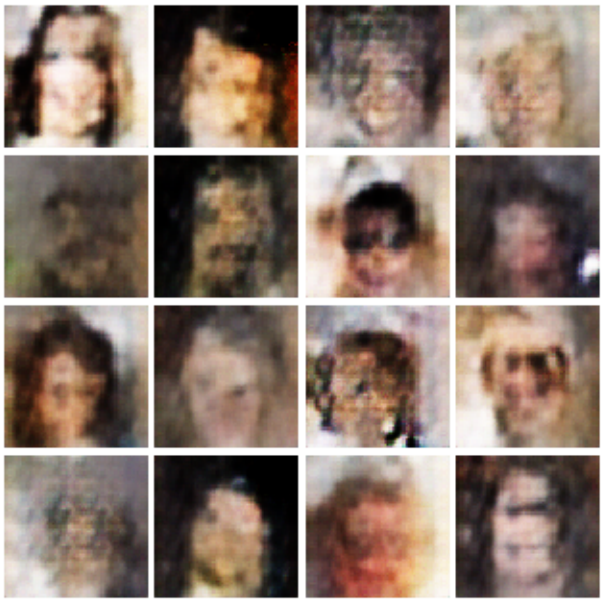


```
EPOCH:  2
Iter: 500, D: 0.0173, G:0.5207
```

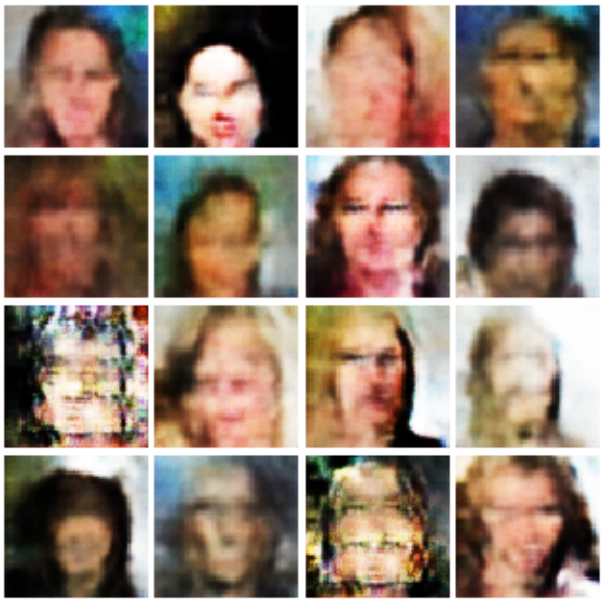
EPOCH: 3
Iter: 1000, D: 0.1303, G:0.4436

EPOCH: 4
EPOCH: 5

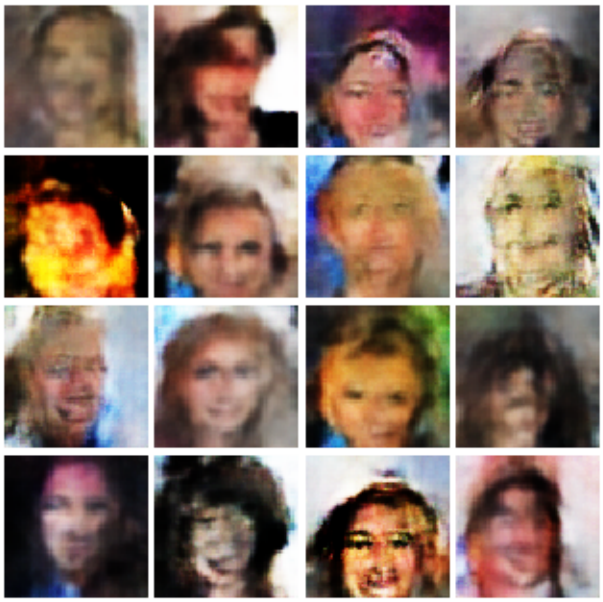
```
Iter: 1500, D: 0.2941, G:0.4789
```



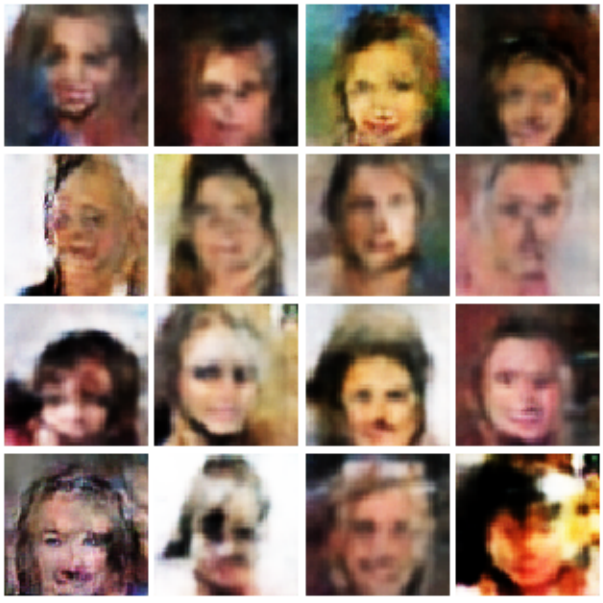
```
EPOCH:  6
Iter: 2000, D: 0.412, G:0.5328
```



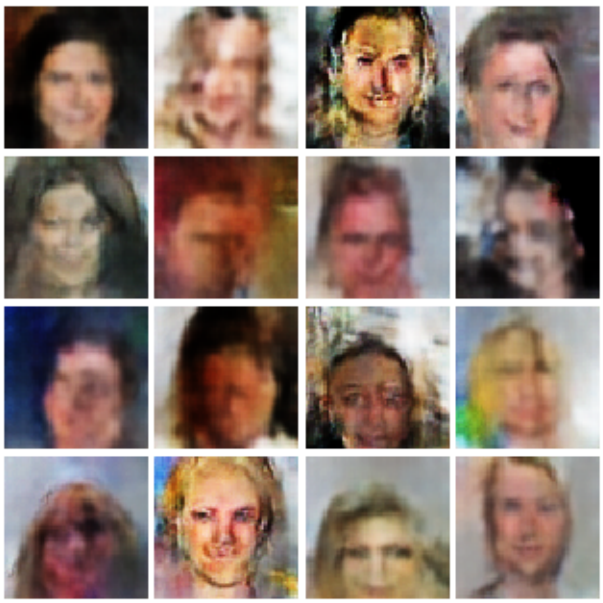
EPOCH: 7
Iter: 2500, D: 0.05499, G:0.5427



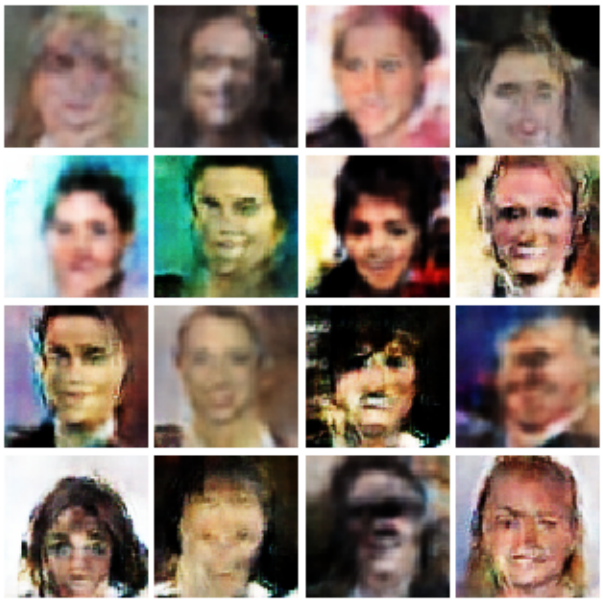
EPOCH: 8
EPOCH: 9
Iter: 3000, D: 0.1079, G:0.3632



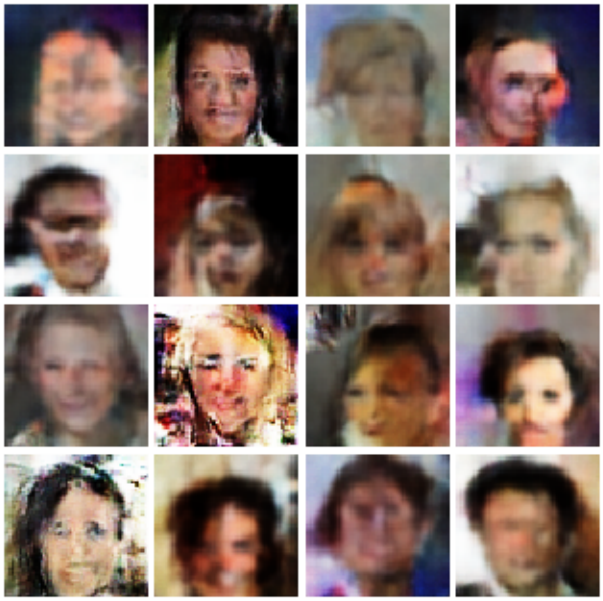
```
EPOCH:  10
Iter: 3500, D: 0.04849, G:0.5032
```



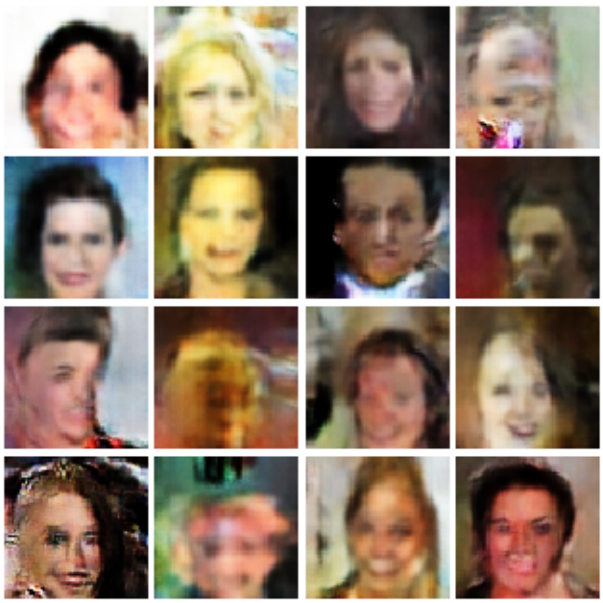
```
EPOCH:  11
Iter: 4000, D: 0.08228, G:0.3931
```



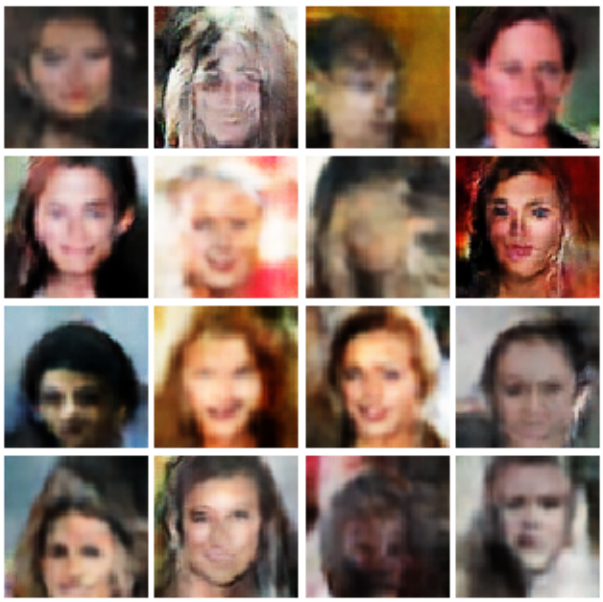
```
EPOCH:  12
EPOCH:  13
Iter: 4500, D: 0.04605, G:0.4611
```



```
EPOCH:  14
Iter: 5000, D: 0.035, G:0.5252
```

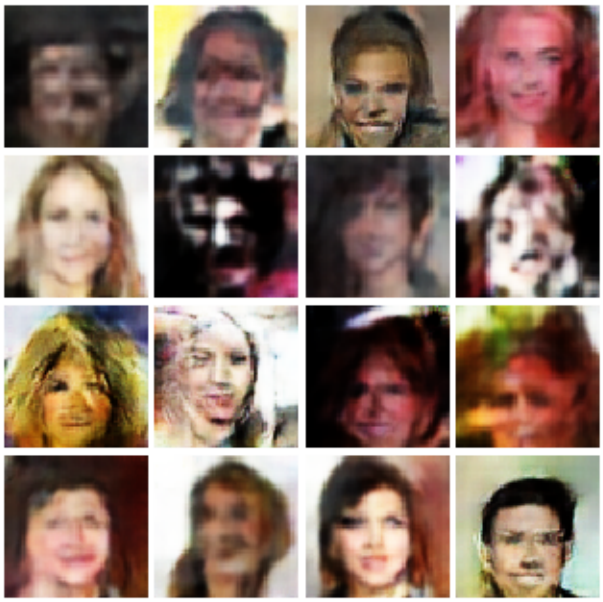


```
EPOCH:  15
EPOCH:  16
Iter: 5500, D: 0.1098, G:0.3469
```

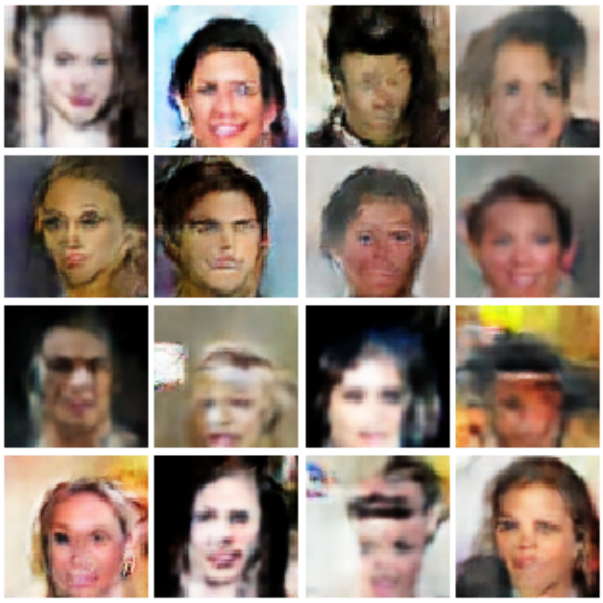


```
EPOCH:  17
```

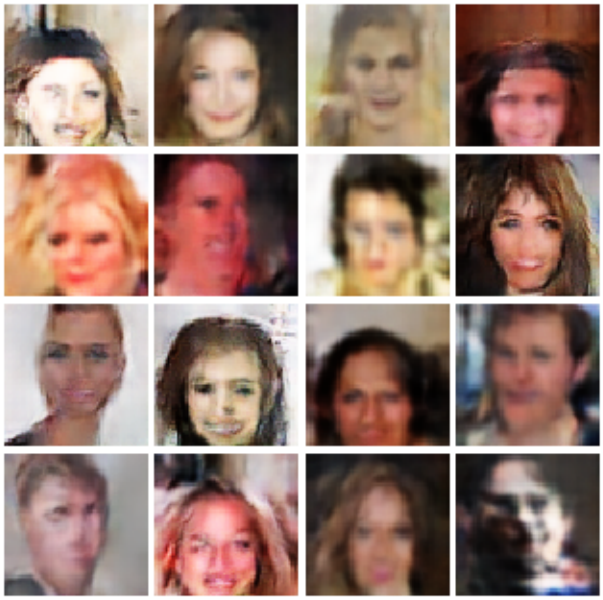
```
Iter: 6000, D: 0.04017, G:0.5011
```



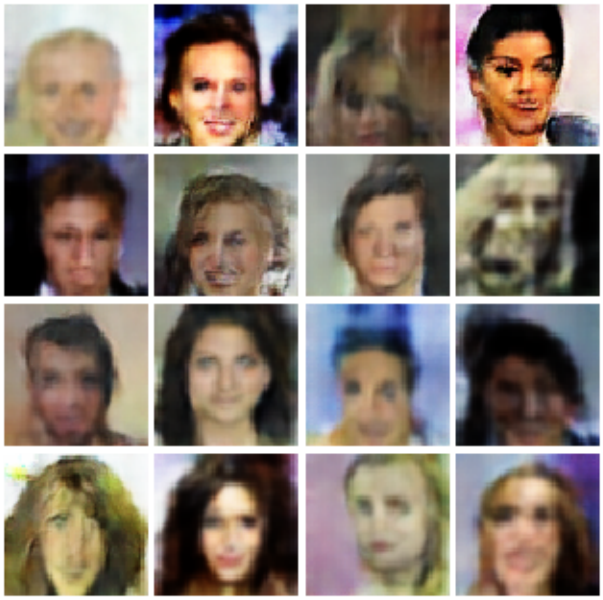
```
EPOCH:  18
Iter: 6500, D: 0.1008, G:0.4626
```



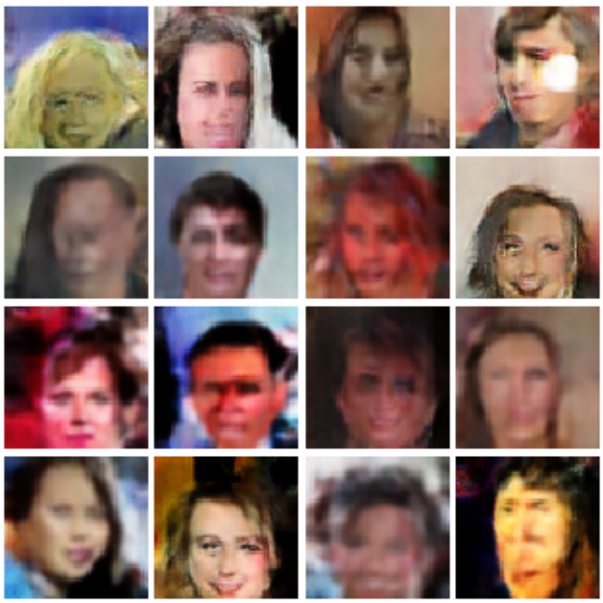
```
EPOCH:  19
EPOCH:  20
Iter: 7000, D: 0.02863, G:0.4605
```



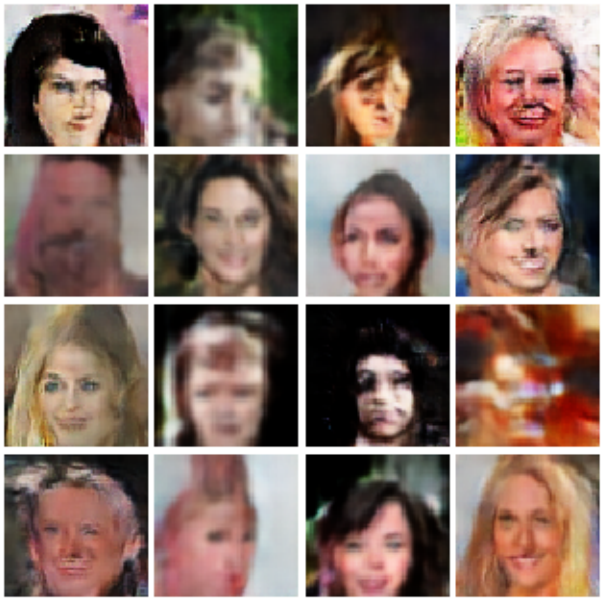
```
EPOCH:  21
Iter: 7500, D: 0.139, G:0.4785
```



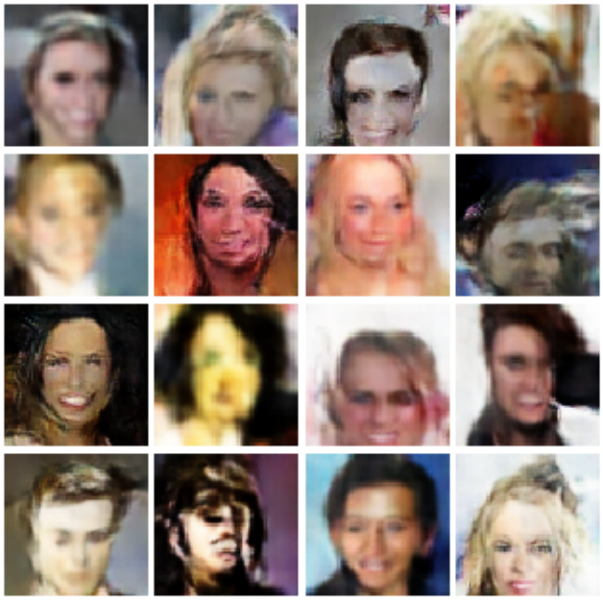
```
EPOCH:  22
Iter: 8000, D: 0.02935, G:0.379
```



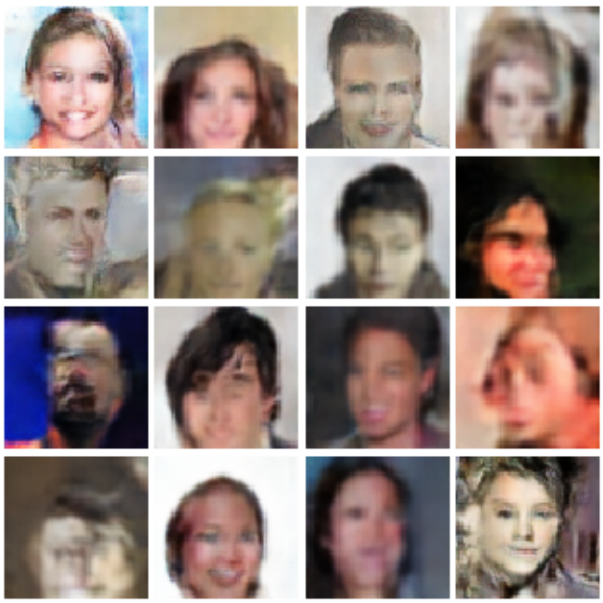
```
EPOCH:  23
EPOCH:  24
Iter: 8500, D: 0.01558, G:0.5125
```



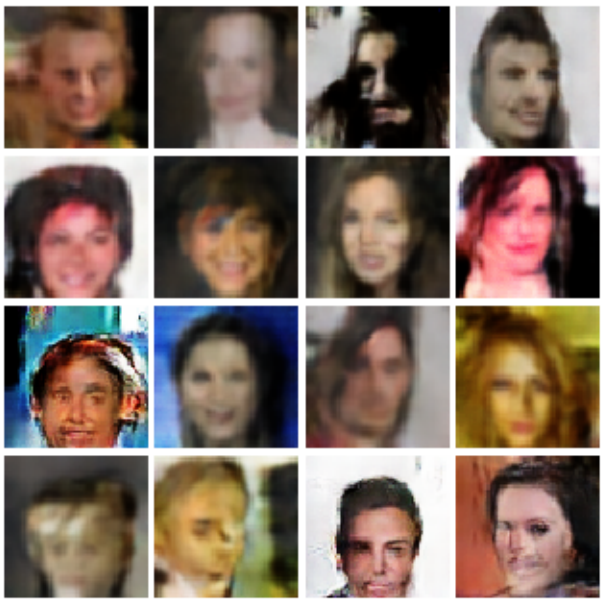
```
EPOCH:  25
Iter: 9000, D: 0.05985, G:0.4245
```



```
EPOCH:  26
Iter: 9500, D: 0.04011, G:0.4385
```

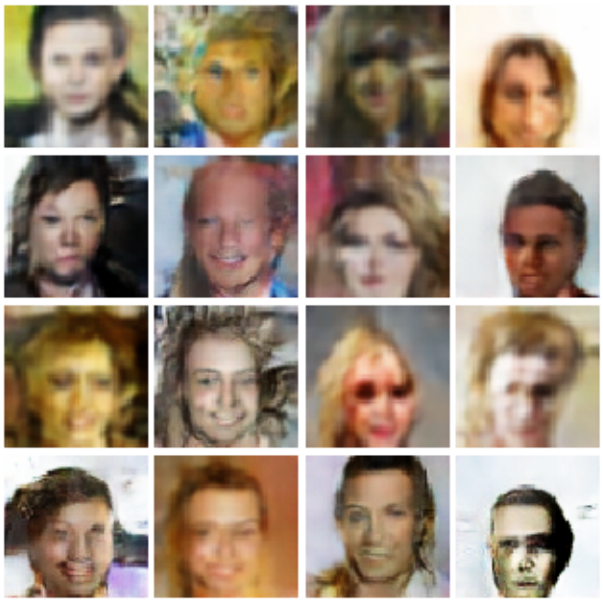


```
EPOCH:  27
EPOCH:  28
Iter: 10000, D: 0.05926, G:0.3332
```

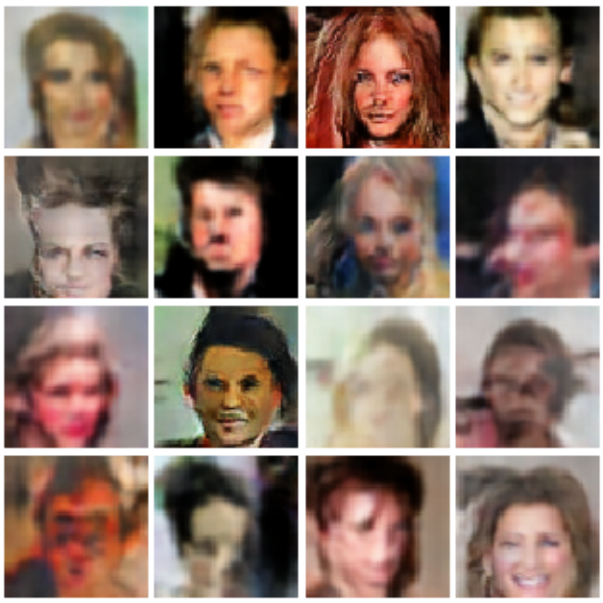


```
EPOCH:  29
```

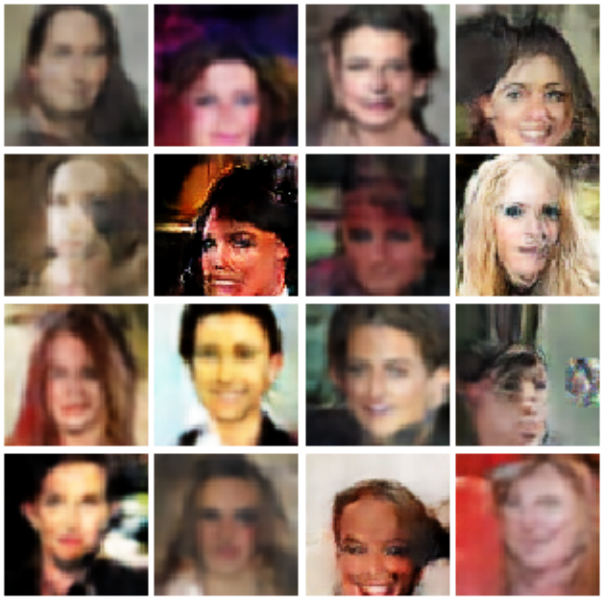
```
Iter: 10500, D: 0.02504, G:0.4754
```



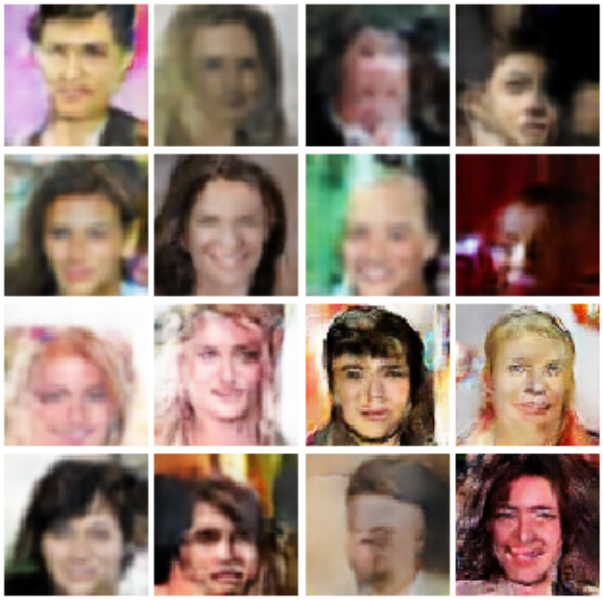
```
EPOCH:  30
EPOCH:  31
Iter: 11000, D: 0.02017, G:0.4829
```



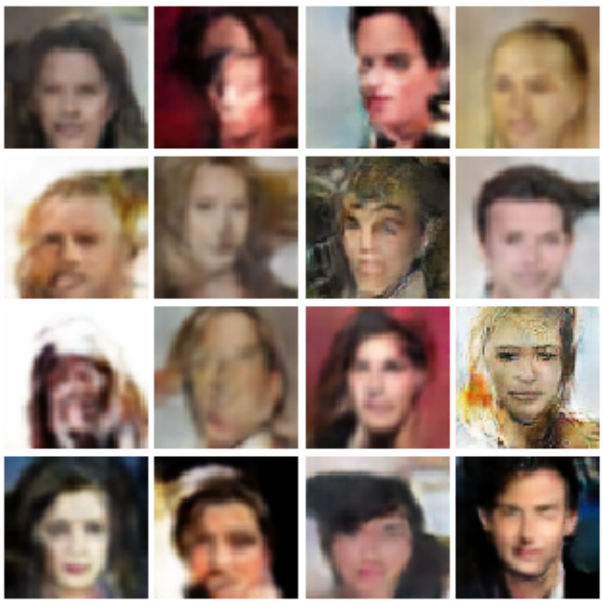
EPOCH: 32
Iter: 11500, D: 0.03373, G:0.5699



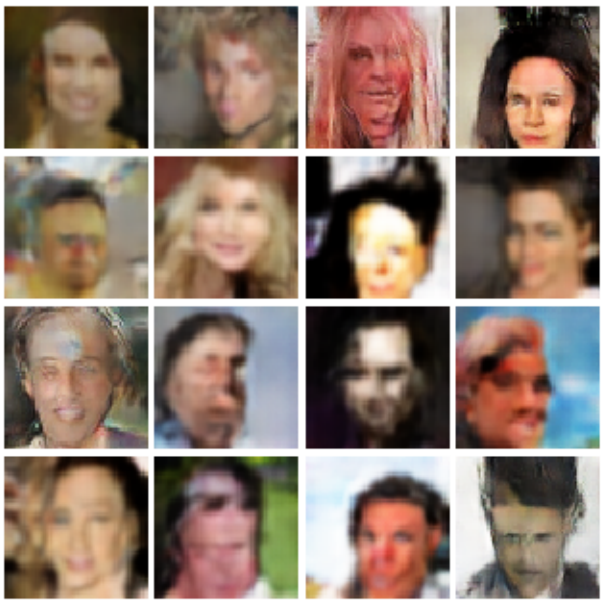
EPOCH: 33
Iter: 12000, D: 0.02253, G:0.5387



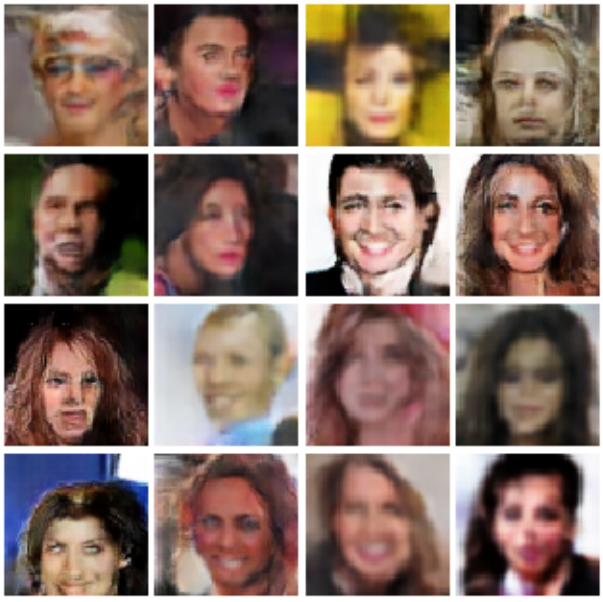
```
EPOCH:  34
EPOCH:  35
Iter: 12500, D: 0.01565, G:0.4914
```



```
EPOCH:  36
Iter: 13000, D: 0.02636, G:0.5897
```

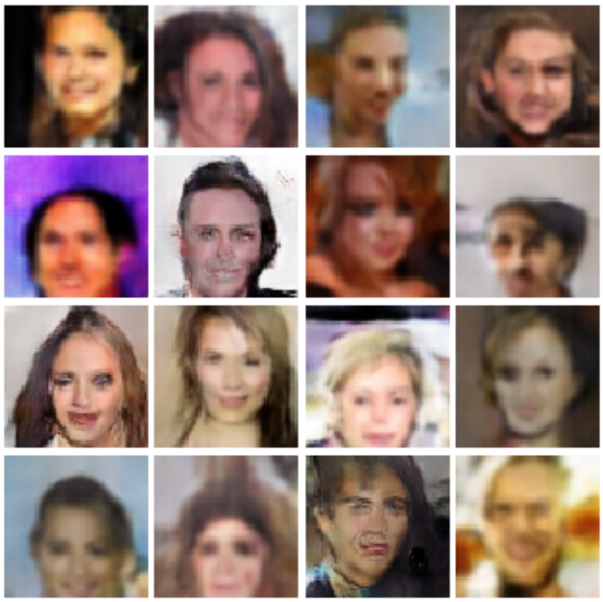


EPOCH: 37
Iter: 13500, D: 0.01392, G:0.4781

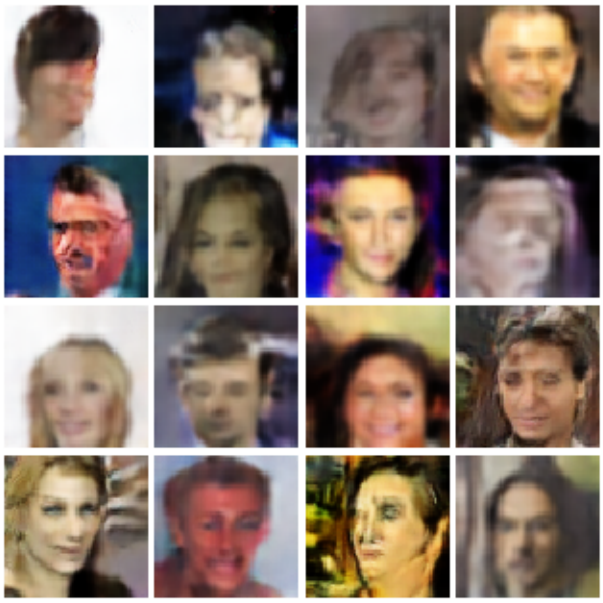


EPOCH: 38
EPOCH: 39

```
Iter: 14000, D: 0.04197, G:0.4795
```

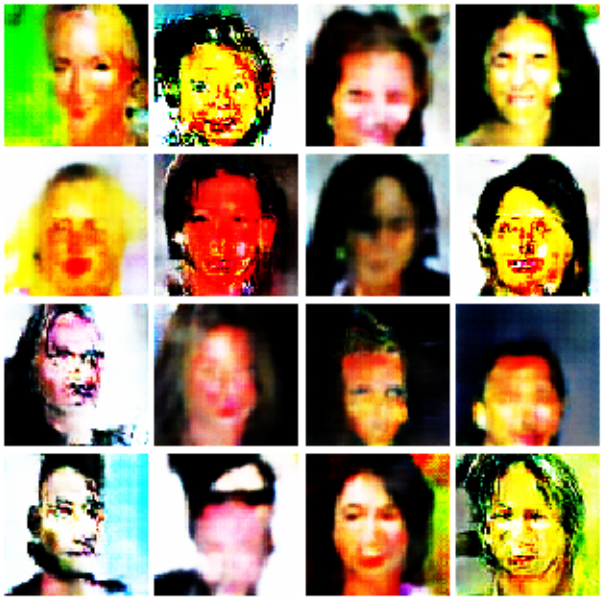


```
EPOCH:  40
Iter: 14500, D: 0.05925, G:0.4232
```

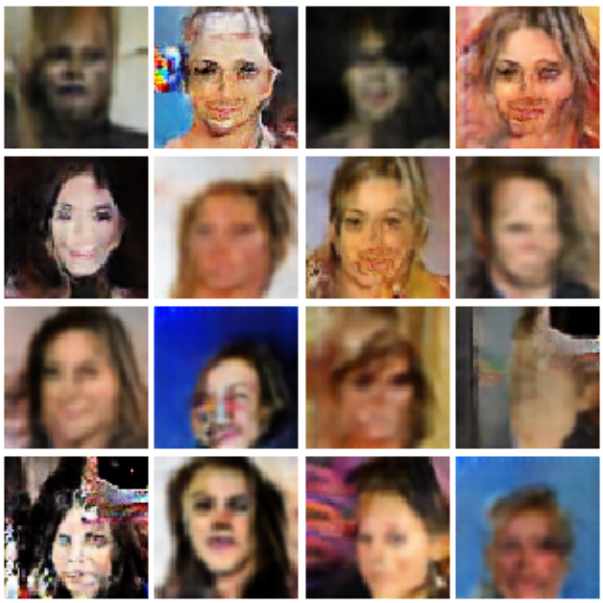


EPOCH: 41
Iter: 15000, D: 0.008526, G:0.4932

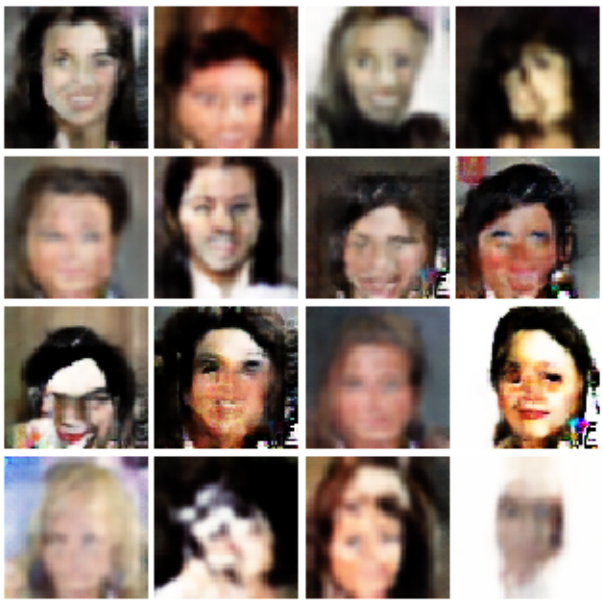
EPOCH: 42
EPOCH: 43
Iter: 15500, D: 0.00521, G:0.5452



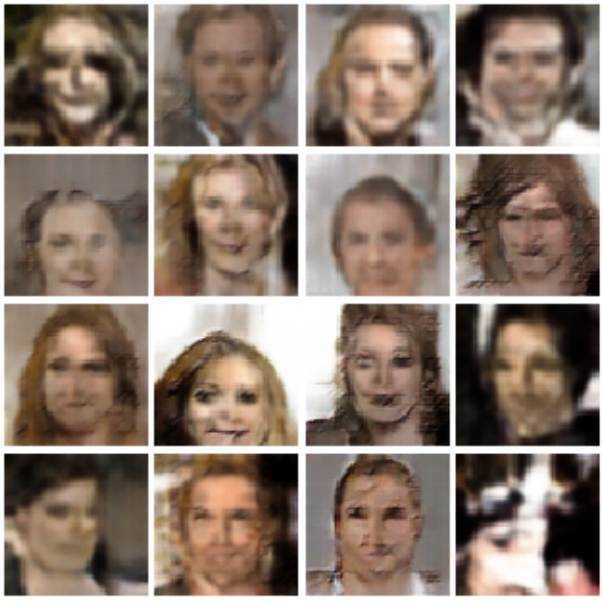
```
EPOCH:  44
Iter: 16000, D: 0.005171, G:0.5401
```



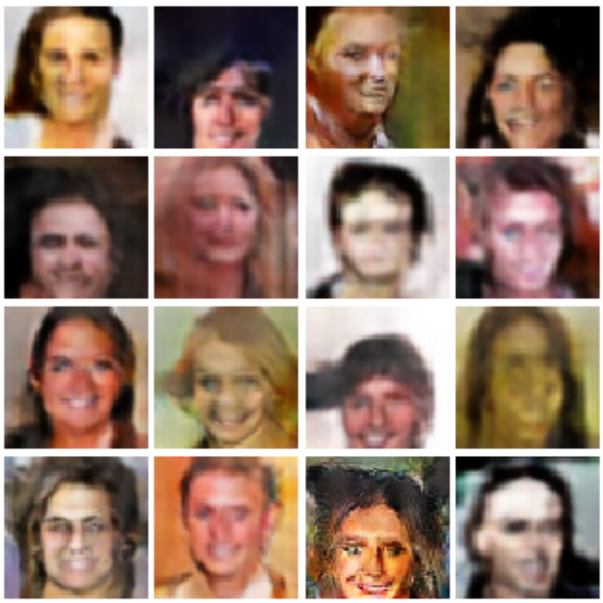
```
EPOCH:  45
EPOCH:  46
Iter: 16500, D: 0.001548, G:0.5003
```



EPOCH: 47
Iter: 17000, D: 0.03123, G:0.528



EPOCH: 48
Iter: 17500, D: 0.03637, G:0.5304



```
EPOCH:  49
EPOCH:  50
Iter: 18000, D: 0.01503, G:0.4588
```