Assignment_5_Code_906213559

November 21, 2020

1 ECE-6524 / CS-6524 Deep Learning

2 Assignment 5 (110 pts)

In this assignment, **you will implement the following:** 1. GAN / LSGAN loss functions and training code for MNIST dataset. 2. Train GAN / LSGAN on CelebA dataset with DCGAN architecture.

Again, it may take hours to train on CelebA. So start early. (We don't want you to work during holidays)

2.1 Submission guideline for the coding part (Jupyter Notebook)

1. Click the Save button at the top of the Jupyter Notebook
2. Please make sure to have entered your Virginia Tech PID below
3. Once you've completed everything (make sure output for all cells are visible), select File -> Download as -> PDF via LaTeX
4. Look at the PDF file and make sure all your solutions are displayed correctly there
5. Zip this notebook (Please don't include the data). Name it as Assignment_3_Code_[YOUR PID NUMBER].zip
6. Name your PDF file as Assignment_4_NB_[YOUR PID NUMBER].pdf
7. **Submit your zipped file and the PDF SEPARATELY**

Note: if facing issues with step 3 refer: https://pypi.org/project/notebook-as-pdf/

2.2 Submission guideline for the coding part (Google Colab)

1. Click the Save button at the top of the Notebook
2. Please make sure to have entered your Virginia Tech PID below
3. Follow last two cells in this notebook for guidelines to download pdf file of this notebook
4. Look at the PDF file and make sure all your solutions are displayed correctly there
5. Zip this notebook (Please don't include the data). Name it as Assignment_2_Code_[YOUR PID NUMBER].zip
6. Name your PDF file as Assignment_4_NB_[YOUR PID NUMBER].pdf
7. **Submit your zipped file and the PDF SEPARATELY**

While you are encouraged to discuss with your peers, all work submitted is expected to be your own. If you use any information from other resources (e.g. online materials), you are required to cite it below you VT PID. Any violation will result in a 0 mark for the assignment.

2.2.1 Please Write Your VT PID Here: 906213559

2.2.2 Reference (if any):

https://github.com/Zeleni9/pytorch-wgan

https://machinelearningmastery.com/how-to-evaluate-generative-adversarial-networks/

https://arxiv.org/abs/1802.03446(Pros and Cons of GAN Evaluation Measures)

2.2.3 Colab Setup:

- Below are some basic steps for colab setup.
- Make changes based on requirements.
- Comment out in case of ARC or your local device with powerful GPU.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
import sys
# modify "path_to_homework", path of folder in drive, where you uploaded your␣
 ↪homework files
path_to_homework = "/content/drive/My Drive/DL_Fall_2020/Assignment_5/"
sys.path.append(path_to_homework)
```

3 Section 1. Generative Adversarial Networks on MNIST Dataset [54 pts]

In this section, you will need to: 1. Implement two different types of loss functions (GAN / LSGAN) for generative adversarial networks. 2. Build the Discriminator and Generator. 3. Implement training codes for your GAN models. 3. Train your model on MNIST dataset and visualize the generated images.

Now, let's get started!

[ ]:
```
import torch
import torch.nn as nn
from torchvision import datasets
from torchvision import transforms
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder

import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

```
%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
```

[ ]:
```
def show_images(images, color=False):
    if color:
        sqrtimg = int(np.ceil(np.sqrt(images.shape[2]*images.shape[3])))
    else:
        images = np.reshape(images, [images.shape[0], -1])  # images reshape to␣
 ↪(batch_size, D)
        sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))


    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        if color:
            plt.imshow(np.swapaxes(np.swapaxes(img, 0, 1), 1, 2))
        else:
            plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

def preprocess_img(x):
    return 2 * x - 1.0

def deprocess_img(x):
    return (x + 1.0) / 2.0

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def sample_noise(batch_size, dim):
    """
    Generate a PyTorch Tensor of uniform random noise.
```

```
    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    to_return = torch.randn((batch_size, dim))
    return to_return/torch.max(to_return)

class Flatten(nn.Module):
  def forward(self, x):
    N, C, H, W = x.size() # read in N, C, H, W
    return x.view(N, -1)
```

[ ]:
```
# set your device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

3.1 Section 1.1. Vanilla GAN loss functions

3.1.1 Section 1.1.1 GAN Loss [10 pts]

We start from implementing the vanilla GAN loss from the original GAN paper. Specifically, you need to complete the `generator_loss` and `discriminator_loss` in the cell below.

Recalled from the class, the generator loss is written as:

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

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 equations could be silghtly different from what we have seen before. This is because that in Pytorch, our optimize will be *minimizing* the loss functions. As a result, we negate the formulas to match pytorch's behavior.

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.

```
[ ]: def discriminator_loss(logits_real, logits_fake):
        """
        Computes the discriminator loss.

        You should use the stable torch.nn.functional.
     ↪binary_cross_entropy_with_logits
        loss rather than using a separate softmax function followed by the binary␣
     ↪cross
        entropy loss.

        Inputs:
        - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
        - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

        Returns:
        - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
        """

        loss = None

        ###################################
        #           YOUR CODE HERE        #
        ###################################
        real = torch.ones_like(logits_real)
        fake = torch.zeros_like(logits_fake)
        loss = torch.nn.functional.binary_cross_entropy_with_logits(logits_real,␣
     ↪real, reduction='mean') + torch.nn.functional.
     ↪binary_cross_entropy_with_logits(logits_fake, fake, reduction='mean')
        ##########       END      ##########
        return loss

    def generator_loss(logits_fake):
        """
        Computes the generator loss.

        You should use the stable torch.nn.functional.
     ↪binary_cross_entropy_with_logits
        loss rather than using a separate softmax function followed by the binary␣
     ↪cross
        entropy loss.

        Inputs:
        - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

        Returns:
        - loss: PyTorch Tensor containing the (scalar) loss for the generator.
        """
```

```
    loss = None

    ##################################
    #           YOUR CODE HERE       #
    ##################################
    fake = torch.ones_like(logits_fake)
    loss = torch.nn.functional.binary_cross_entropy_with_logits(logits_fake,␣
↪fake, reduction='mean')
    ########       END      ##########
    return loss
```

3.1.2 Section 1.1.2 Build simple model [10 pts]

Build your simple model using below layers:

Discriminator:

- Flatten input (flatten the C x H x W into a single vector per image)
- linear (784, 256)
- Leaky ReLU ()
- Linear(256, 256)
- Leaky ReLU()
- Linear(256, 1)

Generator: - Linear(noise_dim, 1024) - ReLU() - Linear(1024, 1024) - ReLU() - linear(1024, 784) - Tanh()

[ ]:
```
NOISE_DIM = 100

def discriminator():
    """
    Initialize and return a simple discriminator model.
    """
    # Your code here:
    model = torch.nn.Sequential(Flatten(),torch.nn.Linear(784, 256),torch.nn.
↪LeakyReLU(),torch.nn.Linear(256, 256),torch.nn.LeakyReLU(),torch.nn.
↪Linear(256, 1))
    return model

def generator(noise_dim=NOISE_DIM):
    """
    Initialize and return a simple generator model.
    """
    # Your code here:
    model = nn.Sequential(torch.nn.Linear(noise_dim, 1024),torch.nn.ReLU(),torch.
↪nn.Linear(1024, 1024),torch.nn.ReLU(),torch.nn.Linear(1024, 784),torch.nn.
↪Tanh())
```

```
    return model
```

3.1.3 Section 1.1.3 Training code [10 pts]

You can't train a model without a proper training code. Implement the GAN training procedure here following the original GAN paper and the course slides in the cell below. Note that this code would be reused in the subsequent section, so make sure that it is correctly implemented.

[ ]:
```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

def train(D, G, D_solver, G_solver, discriminator_loss, generator_loss,␣
↪show_every=250,
          ckpt_dir = path_to_homework + '/ckpts/Vanilla_gan_MINST/',␣
↪cont_train=False,
          batch_size=128, noise_size=100, num_epochs=10, train_loader=None,␣
↪device=None):
    """
    Train loop for GAN.

    The loop will consist of two steps: a discriminator step and a generator␣
↪step.

    (1) In the discriminator step, you should zero gradients in the discriminator
    and sample noise to generate a fake data batch using the generator. Flatten␣
↪real images
    to 784 (28 * 28). Calculate the discriminator output for real and fake data,
    and use the output to compute discriminator loss. Call backward() on the␣
↪loss
    output and take an optimizer step for the discriminator.

    (2) For the generator step, you should once again zero gradients in the␣
↪generator
    and sample noise to generate a fake data batch. Get the discriminator output
    for the fake data batch and use this to compute the generator loss. Once␣
↪again
    call backward() on the loss and take an optimizer step.

    You will need to reshape the fake image tensor outputted by the generator to
    be dimensions (batch_size x input_channels x img_size x img_size).

    Use the sample_noise function to sample random noise, and the␣
↪discriminator_loss
    and generator_loss functions for their respective loss computations

    Inputs:
```

```
    - D, G: PyTorch models for the discriminator and generator
    - D_solver, G_solver: torch.optim Optimizers to use for training the
      discriminator and generator.
    - discriminator_loss, generator_loss: Functions to use for computing the␣
↪generator and
      discriminator loss, respectively.
    - show_every: Show samples after every show_every iterations.
    - batch_size: Batch size to use for training.
    - noise_size: Dimension of the noise to use as input to the generator.
    - num_epochs: Number of epochs over the training dataset to use for training.
    - train_loader: image dataloader
    - device: PyTorch device
    """
    if os.path.exists(os.path.join(ckpt_dir, 'checkpoint.pth')) and cont_train:
        ckpt = torch.load(os.path.join(ckpt_dir, 'checkpoint.pth'))
        start_epoch = ckpt['epoch'] + 1
        iter_count = ckpt['iter_count'] + 1
        G.load_state_dict(ckpt['G'])
        D.load_state_dict(ckpt['D'])
        print('Start from a checkpoint: {}, epoch:{}, iter:{}'.format(os.path.
↪join(ckpt_dir, 'checkpoint.pth'),
                                                      str(start_epoch),␣
↪str(iter_count)))
    else:
        start_epoch = 0
        iter_count = 0

    for epoch in range(start_epoch, num_epochs):
        print('EPOCH: ', (epoch+1))
        for x, _ in train_loader:
            _, input_channels, img_size, _ = x.shape

            real_images = preprocess_img(x).to(device)  # normalize

            # Store discriminator loss output, generator loss output, and fake␣
↪image output
            # in these variables for logging and visualization below
            d_error = None
            g_error = None
            fake_images = None


            ####################################
            #         Discriminator step       #
            #           YOUR CODE HERE         #
            ####################################
            D_solver.zero_grad()
```

```
            Discriminator = sample_noise(batch_size, noise_size).to(device)
            real_logits = D.forward(real_images)
            fake_logits = D.forward(G.forward(Discriminator).
↪reshape_as(real_images))
            Discriminator_loss = discriminator_loss(real_logits, fake_logits)
            Discriminator_loss.backward()
            D_solver.step()
            d_error = Discriminator_loss.view(-1)
            #########       END      ############

            ####################################
            #          Generator step          #
            #          YOUR CODE HERE          #
            ####################################
            G_solver.zero_grad()
            Generator = sample_noise(batch_size, noise_size).to(device)
            fake_logits = D.forward(G.forward(Generator).reshape_as(real_images))
            Generator_loss = generator_loss(fake_logits)
            Generator_loss.backward()
            G_solver.step()
            g_error = Generator_loss.view(-1)
            fake_images = G.forward(Generator).reshape_as(real_images)
            ##########       END      ##########
            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,d_error.
↪item(),g_error.item()))
                disp_fake_images = deprocess_img(fake_images.data)  # denormalize
                imgs_numpy = (disp_fake_images).cpu().numpy()
                show_images(imgs_numpy[0:16], color=input_channels!=1)
                plt.show()
                print()
            iter_count += 1

        # save checkpoints
        os.makedirs(ckpt_dir, exist_ok=True)
        print('Saving the model as a checkpoint...')
        torch.save({'epoch': epoch,
                    'iter_count': iter_count,
                    'G': G.state_dict(),
                    'D': D.state_dict()},
                   os.path.join(ckpt_dir, 'checkpoint.pth'))
```

MNIST is a simple dataset that contains one hand-written digit in each image. It is usually used for sanity check. So, let's test our loss functions and training code on it!

[ ]:
```
batch_size = 128
```

```
mnist = datasets.MNIST('./MNIST_data', train=True, download=True,
                           transform=transforms.ToTensor())
loader_train = DataLoader(mnist, batch_size=batch_size, drop_last=True)


imgs = loader_train.__iter__().next()[0].view(batch_size, 784).numpy().squeeze()
show_images(imgs)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./MNIST_data/MNIST/raw/train-images-idx3-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting ./MNIST_data/MNIST/raw/train-images-idx3-ubyte.gz to
./MNIST_data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting ./MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz to
./MNIST_data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

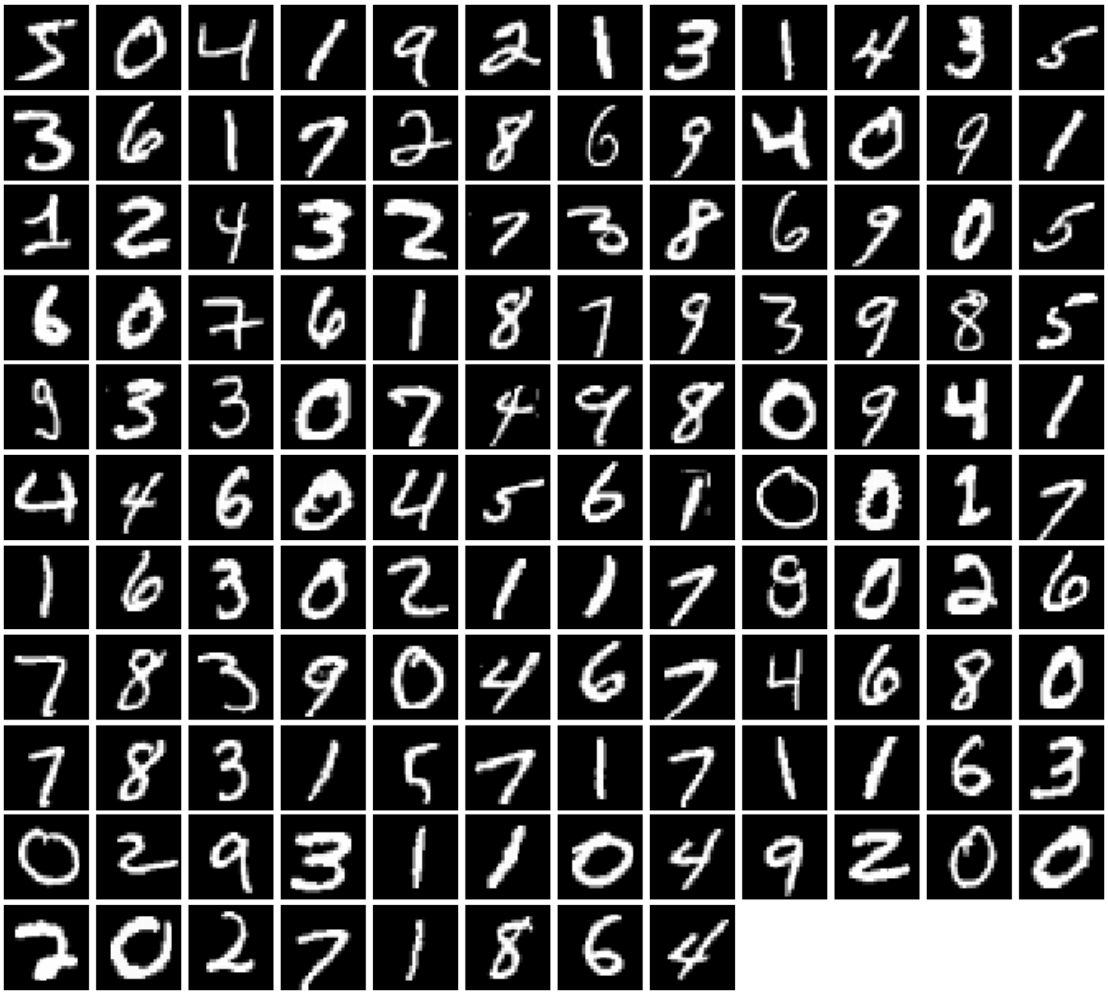
```
Extracting ./MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz to
./MNIST_data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting ./MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
./MNIST_data/MNIST/raw
Processing...



Done!
```

```
/usr/local/lib/python3.6/dist-packages/torchvision/datasets/mnist.py:480:
UserWarning: The given NumPy array is not writeable, and PyTorch does not
support non-writeable tensors. This means you can write to the underlying
(supposedly non-writeable) NumPy array using the tensor. You may want to copy
the array to protect its data or make it writeable before converting it to a
tensor. This type of warning will be suppressed for the rest of this program.
```

```
(Triggered internally at  /pytorch/torch/csrc/utils/tensor_numpy.cpp:141.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```



3.1.4 Section 1.1.4 Train your model [7 pts]

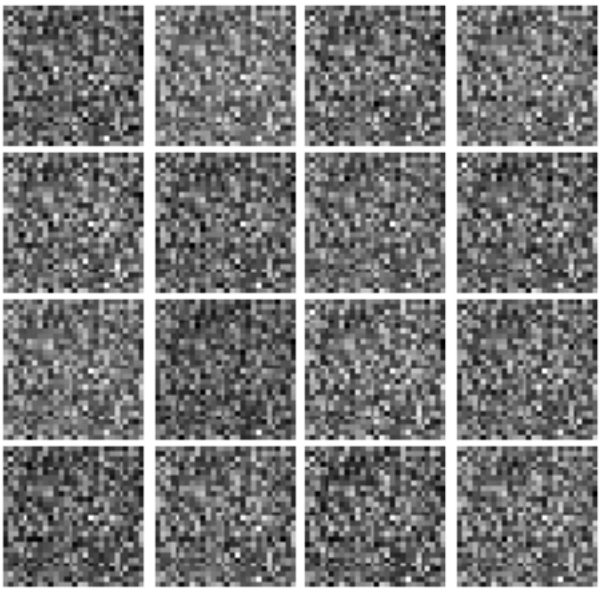
- Call Discriminator and Generator for training.
- Call optimizers for both discriminator and generator for training. (Use Adam with lr=1e-3, betas = (0.5, 0.999))
- Call train function to train.
- Train for 10 epochs.

After training your GAN model, you should expect results that resemble the following if your loss function and training loop implementations are correct:

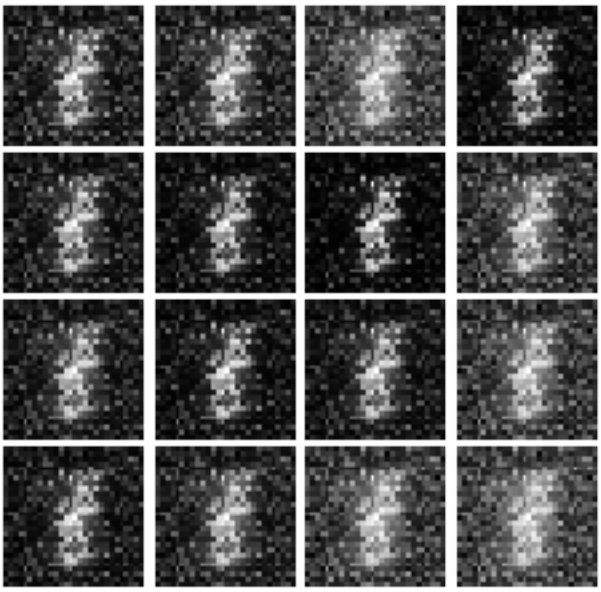
Refer mnist.jpg from gan_samples folder.

[ ]:
```
# Add training code here:
# original GAN
Discriminator = discriminator().to(device)
Generator = generator().to(device)
Discriminator_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=1e-3,␣
 ↪betas = (0.5, 0.999))
Generator_optimizer = torch.optim.Adam(Generator.parameters(), lr=1e-3, betas =␣
 ↪(0.5, 0.999))
train(Discriminator, Generator, Discriminator_optimizer, Generator_optimizer,␣
 ↪discriminator_loss, generator_loss, train_loader=loader_train, device=device)
```

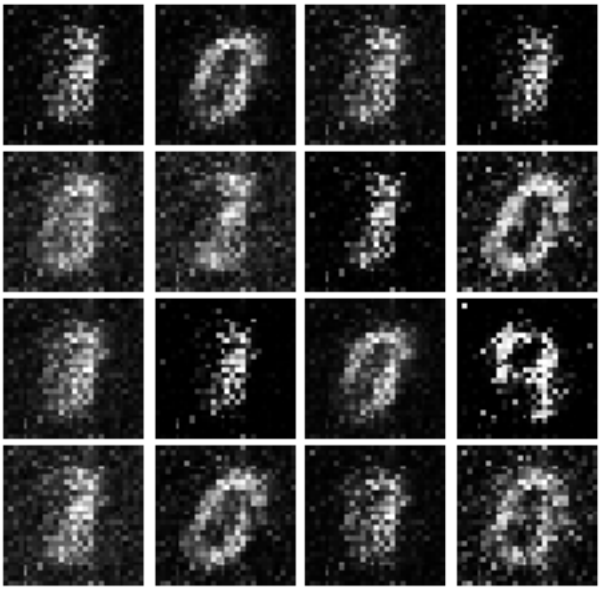
```
EPOCH:  1
Iter: 0, D: 1.365, G:0.6973
```



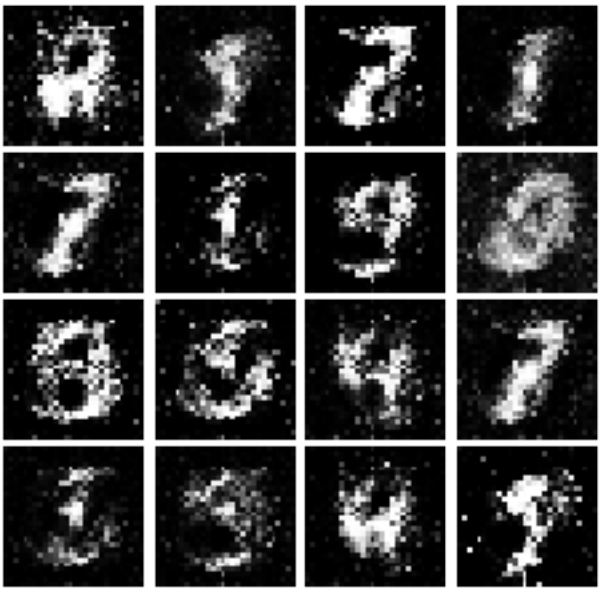
```
Iter: 250, D: 0.8937, G:0.9792
```



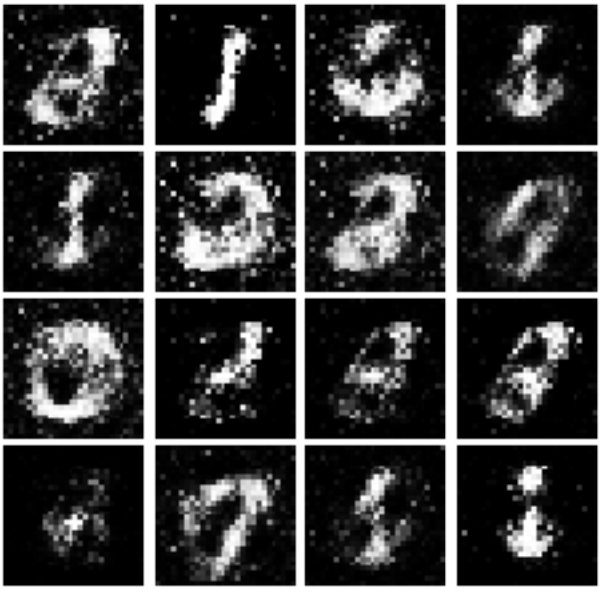
```
Saving the model as a checkpoint...
EPOCH:  2
Iter: 500, D: 1.257, G:1.247
```



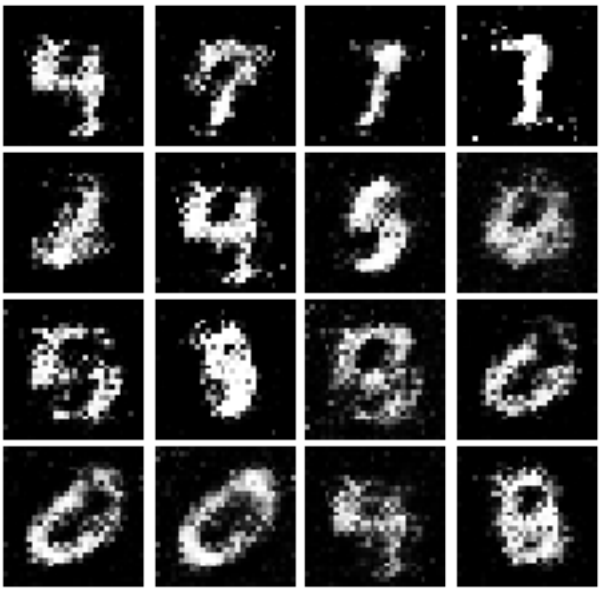
```
Iter: 750, D: 0.8385, G:1.785
```



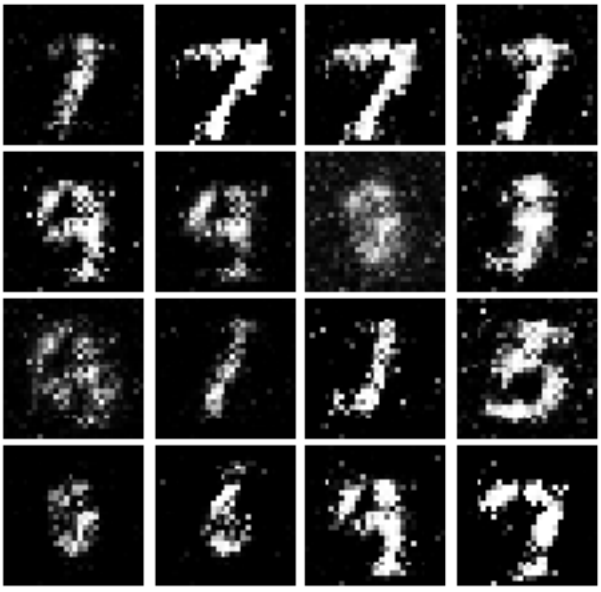
```
Saving the model as a checkpoint...
EPOCH:  3
Iter: 1000, D: 1.146, G:1.066
```



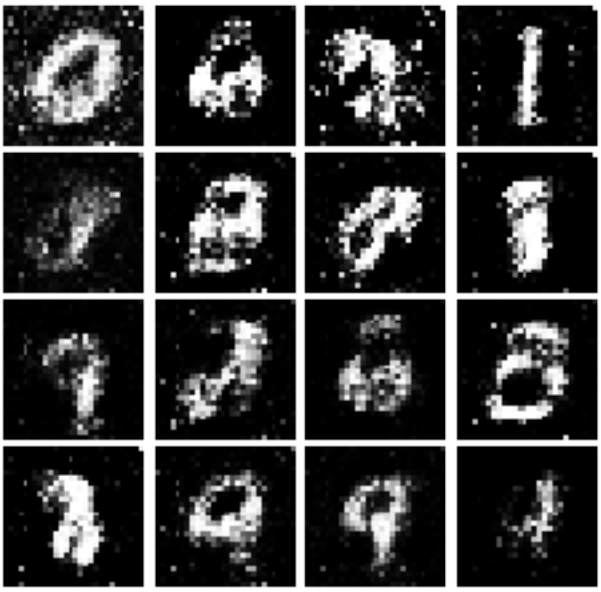
```
Iter: 1250, D: 1.138, G:0.9237
```



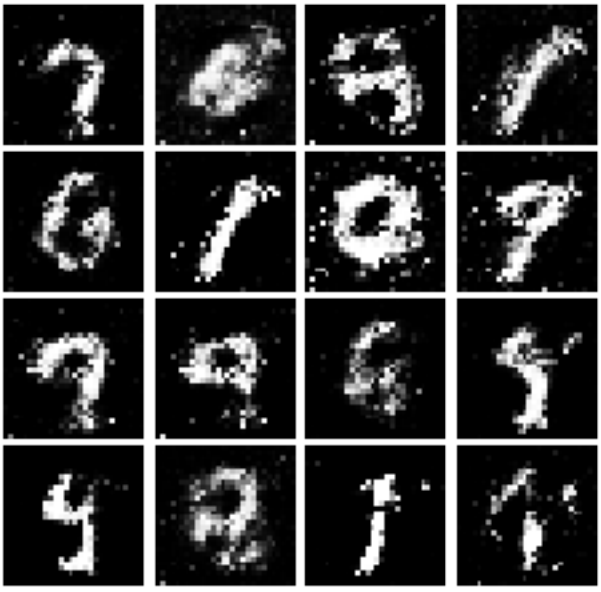
```
Saving the model as a checkpoint...
EPOCH:  4
Iter: 1500, D: 1.166, G:1.18
```



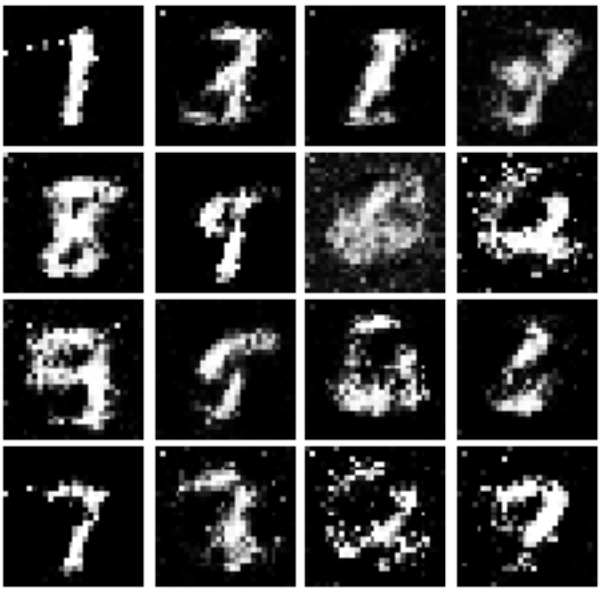
```
Iter: 1750, D: 1.073, G:0.981
```



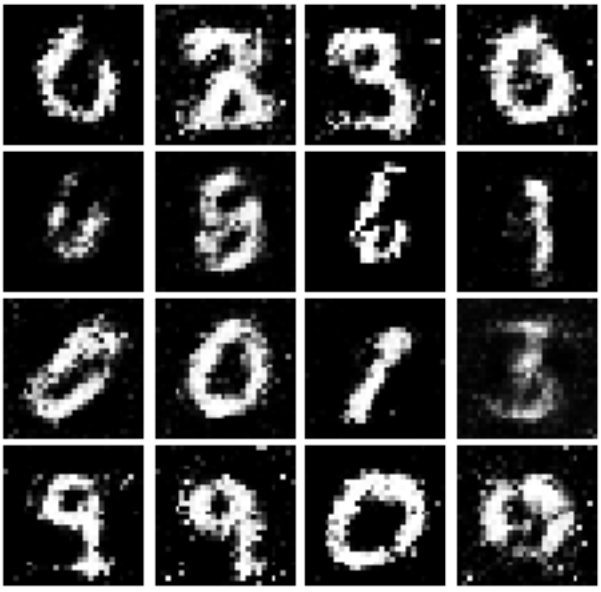
```
Saving the model as a checkpoint...
EPOCH:  5
Iter: 2000, D: 1.236, G:0.8874
```



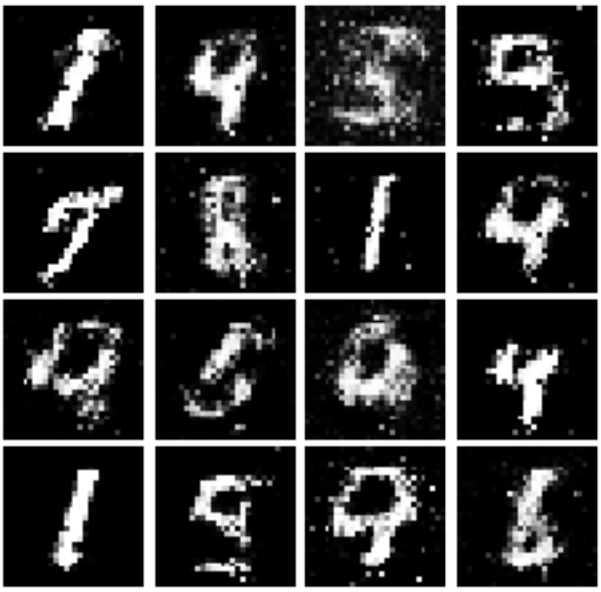
```
Iter: 2250, D: 1.232, G:0.8178
```



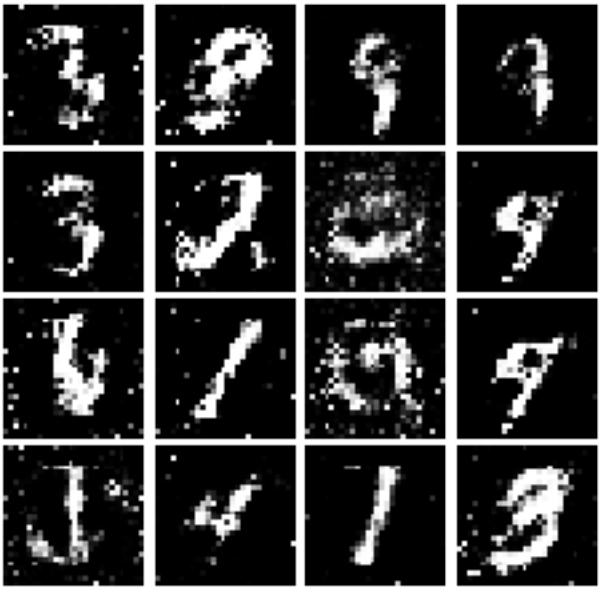
```
Saving the model as a checkpoint...
EPOCH:  6
Iter: 2500, D: 1.258, G:0.7426
```



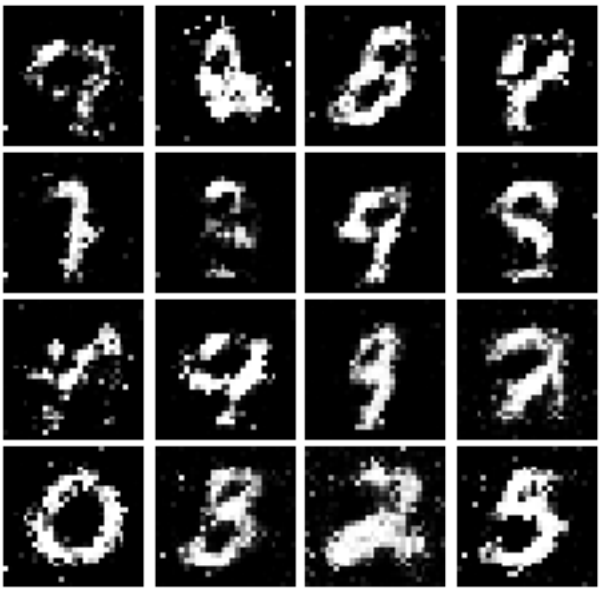
```
Iter: 2750, D: 1.325, G:0.9565
```



```
Saving the model as a checkpoint...
EPOCH:  7
Iter: 3000, D: 1.328, G:0.7802
```



```
Iter: 3250, D: 1.295, G:1.034
```

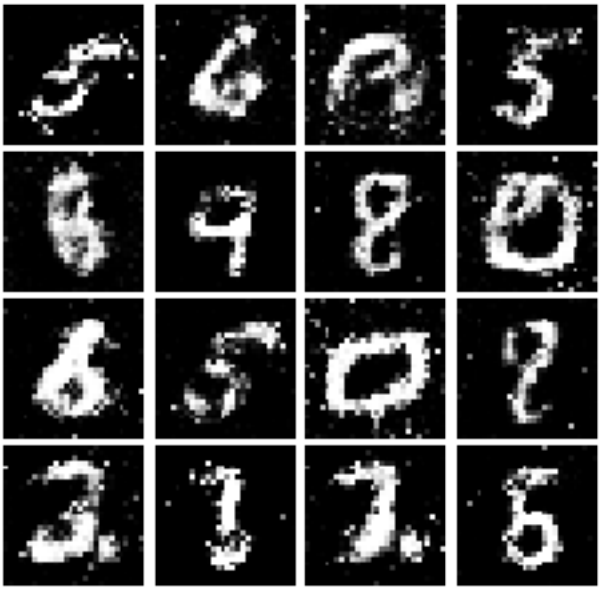


```
Saving the model as a checkpoint...
EPOCH:  8
Iter: 3500, D: 1.412, G:0.7897
```

```
Saving the model as a checkpoint...
```

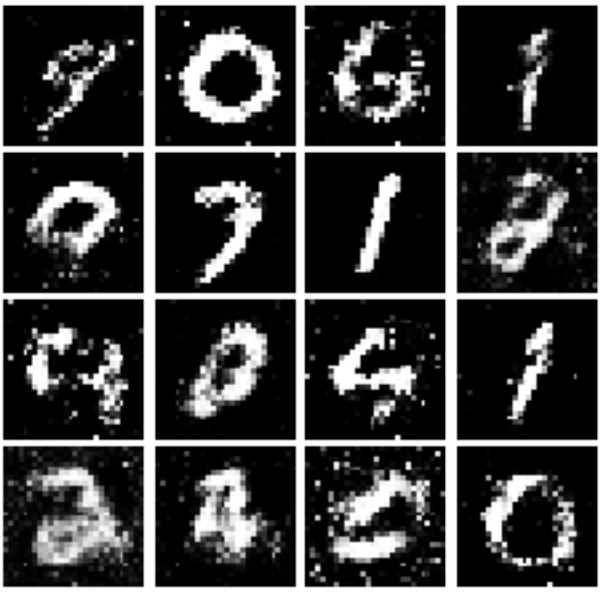
```
EPOCH:  9
Iter: 3750, D: 1.301, G:0.7817
```

```
Iter: 4000, D: 1.335, G:0.7885
```



```
Saving the model as a checkpoint...
```

```
EPOCH:  10
Iter: 4250, D: 1.31, G:0.7612
```



```
Iter: 4500, D: 1.29, G:0.8414
```

```
Saving the model as a checkpoint...
```

3.2 Section 1.2 Least-square GAN Loss

3.2.1 Section 1.2.1 LSGAN Loss [10 pts]

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:

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

and the discriminator loss:

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

Fill in the `ls_discriminator_loss` and `ls_generator_loss` in the cell below.

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`).

[ ]:
```
def ls_discriminator_loss(logits_real, logits_fake):
    """
    Compute the Least-Squares GAN loss for the discriminator.

    Inputs:
    - scores_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Outputs:
    - loss: A PyTorch Tensor containing the loss.
    """

    loss = None

    ##################################
    #           YOUR CODE HERE       #
    ##################################
    loss = torch.mean(0.5*(logits_real-1)**2+0.5*logits_fake**2,dim=0)
    ##########       END      ##########
    return loss

def ls_generator_loss(logits_fake):
    """
    Computes the Least-Squares GAN loss for the generator.

    Inputs:
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Outputs:
```

```
    - loss: A PyTorch Tensor containing the loss.
    """

    loss = None

    ######################################
    #           YOUR CODE HERE           #
    ######################################
    loss = torch.mean(0.5*(logits_fake-1)**2,dim=0)
    ##########       END      ##########
    return loss
```

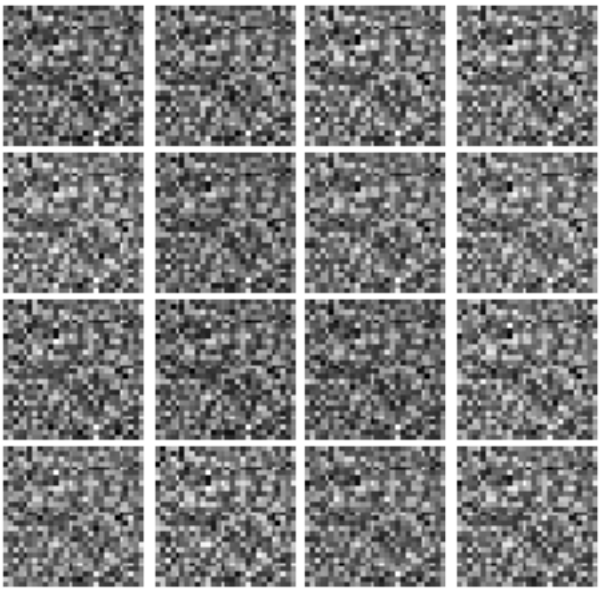
3.2.2 Section 1.2.2 Train model with LSGAN loss[7 pts]

- Call Discriminator and Generator for training.
- Call optimizers for both discriminator and generator for training. (Use Adam with lr=1e-3, betas = (0.5, 0.999))
- Call train function to train.
- Train for 10 epochs.

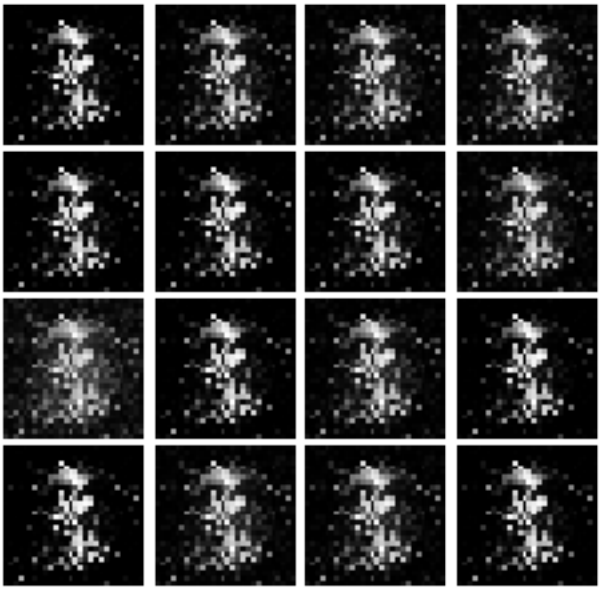
Similarly, train your LSGAN on MNIST dataset. You should expect results that resemble the following if your loss function and training loop implementations are correct:

[ ]:
```
# YOUR TRAINING CODE HERE
Discriminator = discriminator().to(device)
Generator = generator().to(device)
Discriminator_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=1e-3,␣
 ↪betas = (0.5, 0.999))
Generator_optimizer = torch.optim.Adam(Generator.parameters(), lr=1e-3, betas =␣
 ↪(0.5, 0.999))
train(Discriminator, Generator, Discriminator_optimizer, Generator_optimizer,␣
 ↪ls_discriminator_loss, ls_generator_loss, train_loader=loader_train,␣
 ↪device=device)
```

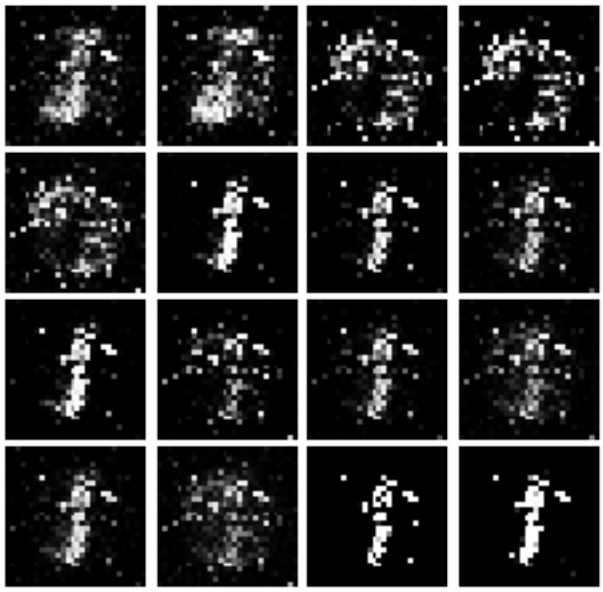
```
EPOCH:  1
Iter: 0, D: 0.5492, G:0.4561
```



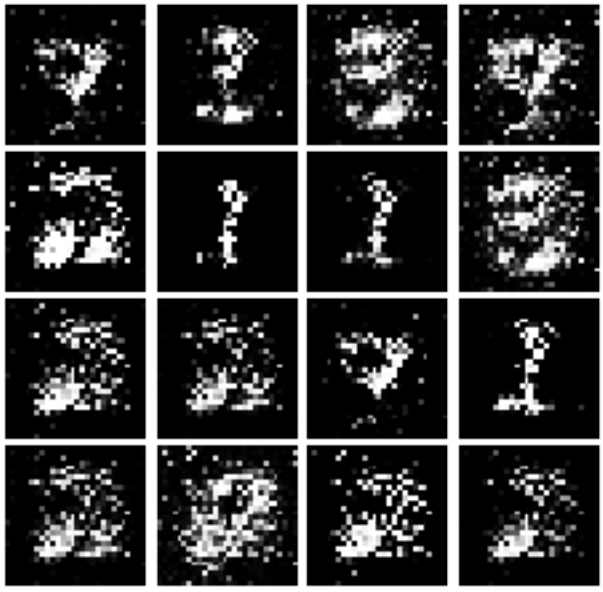
```
Iter: 250, D: 0.1203, G:0.6983
```



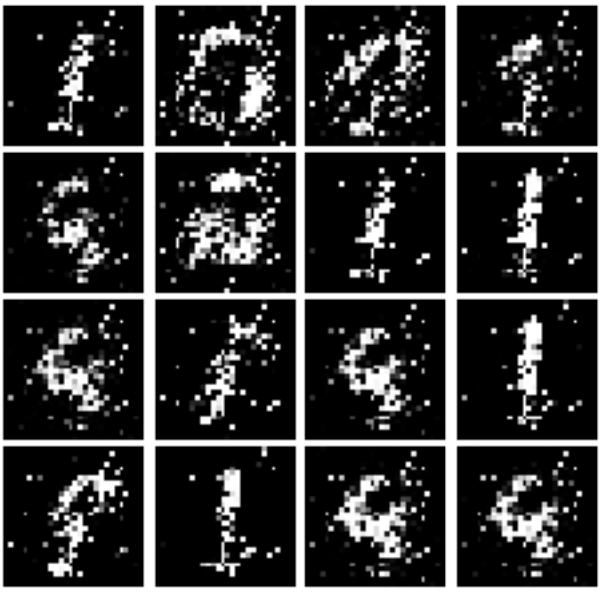
```
Saving the model as a checkpoint...
EPOCH:  2
Iter: 500, D: 0.2079, G:0.3382
```



```
Iter: 750, D: 0.191, G:0.5072
```

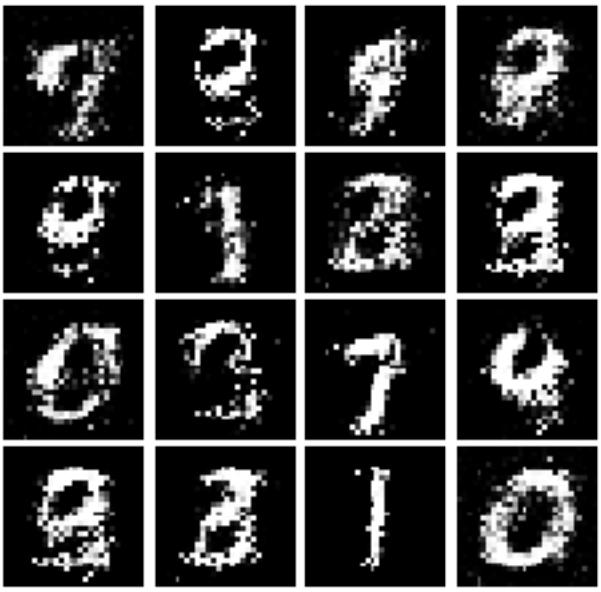


```
Saving the model as a checkpoint...
EPOCH:  3
Iter: 1000, D: 0.1374, G:0.4243
```



```
Iter: 1250, D: 0.1803, G:0.2617
```

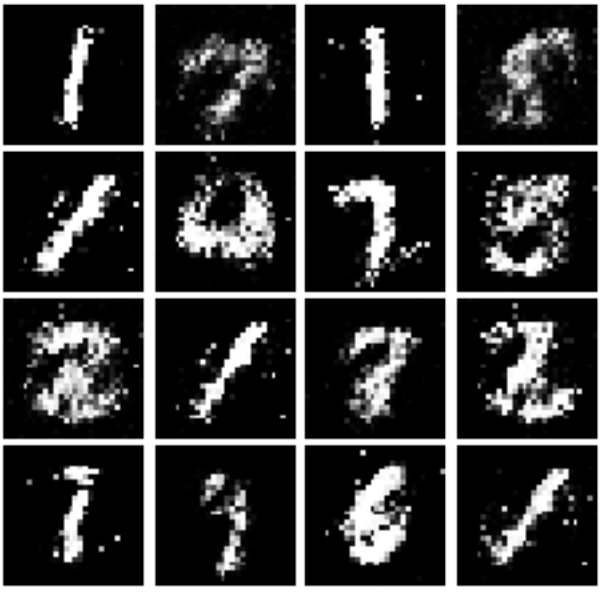
```
Saving the model as a checkpoint...
EPOCH:  4
Iter: 1500, D: 0.1575, G:0.2341
```



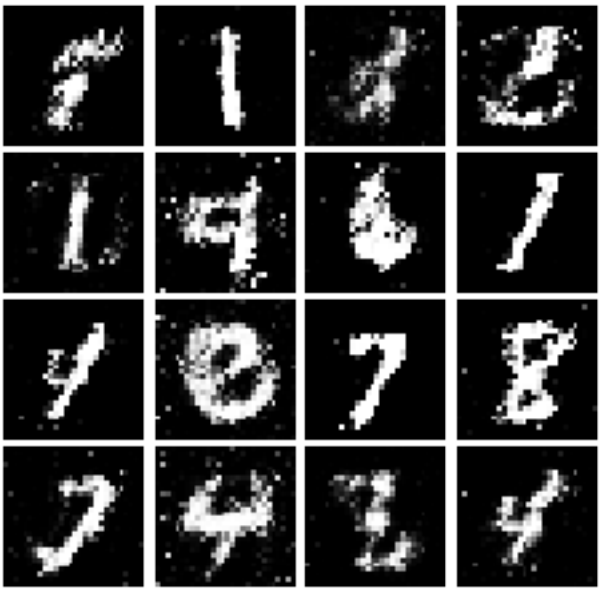
```
Iter: 1750, D: 0.2054, G:0.1936
```

```
Saving the model as a checkpoint...
EPOCH:  5
Iter: 2000, D: 0.2016, G:0.2079
```

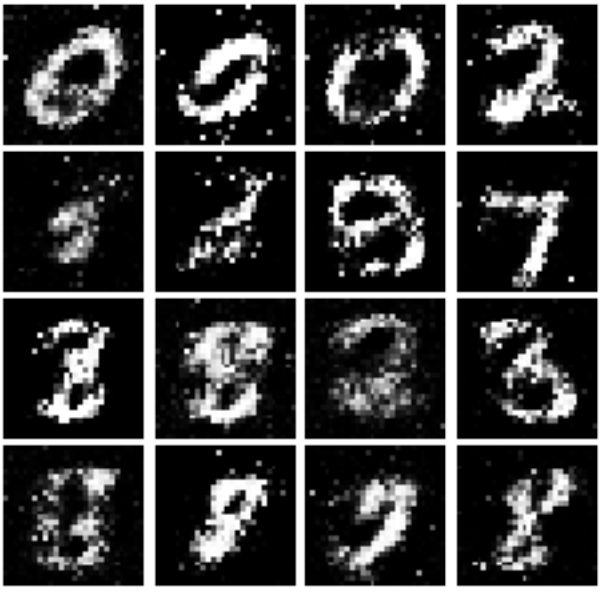
```
Iter: 2250, D: 0.206, G:0.1949
```



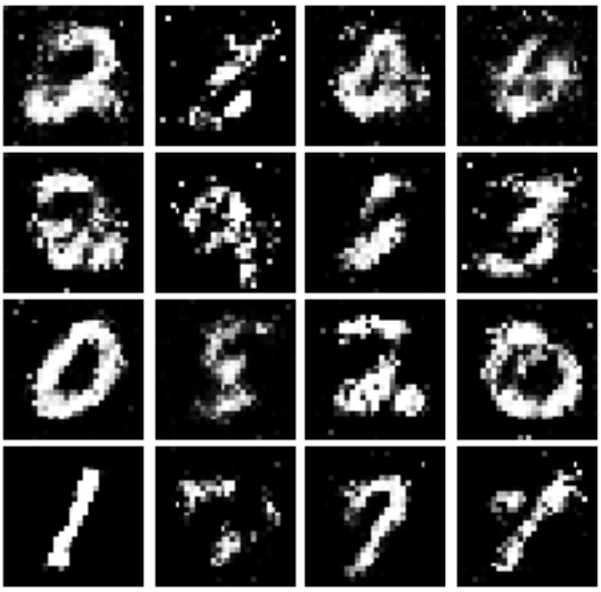
```
Saving the model as a checkpoint...
EPOCH:  6
Iter: 2500, D: 0.2268, G:0.1631
```



```
Iter: 2750, D: 0.2355, G:0.1782
```



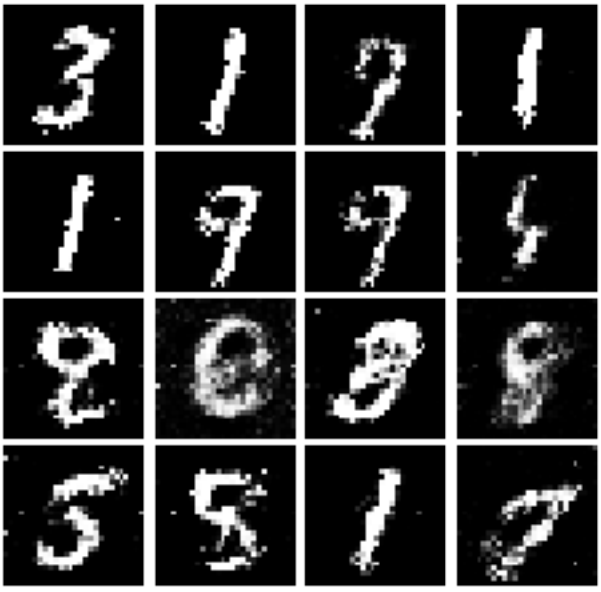
```
Saving the model as a checkpoint...
EPOCH:  7
Iter: 3000, D: 0.2451, G:0.2237
```



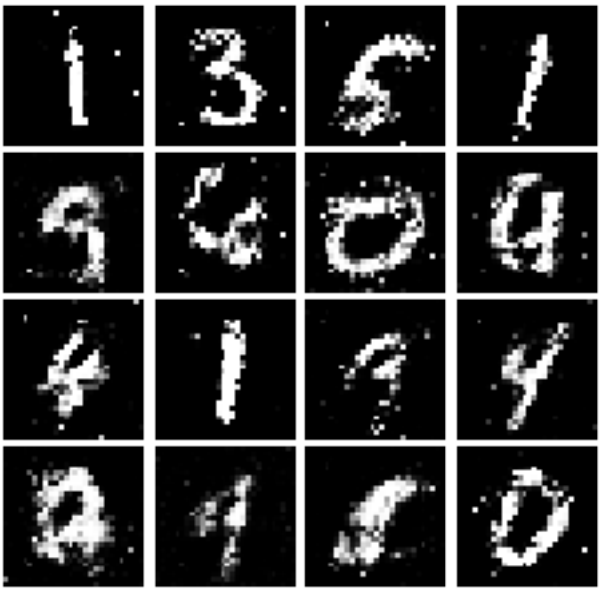
```
Iter: 3250, D: 0.2123, G:0.1828
```

```
Saving the model as a checkpoint...
EPOCH:  8
Iter: 3500, D: 0.2365, G:0.1676
```

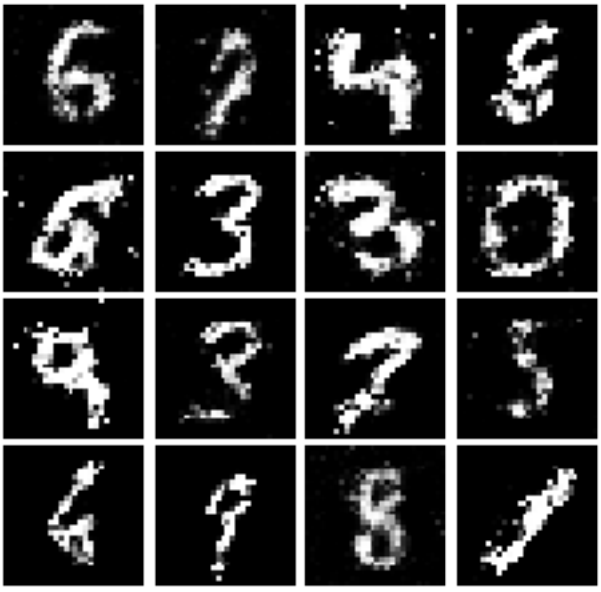
```
Saving the model as a checkpoint...
EPOCH:  9
Iter: 3750, D: 0.2311, G:0.1876
```



```
Iter: 4000, D: 0.2295, G:0.1646
```



```
Saving the model as a checkpoint...
EPOCH:  10
Iter: 4250, D: 0.2027, G:0.1629
```



```
Iter: 4500, D: 0.2325, G:0.1671
```

```
Saving the model as a checkpoint...
```

4 Section 2. Generative Adversarial Networks on CelebA Dataset [56 pts]

In this section, you will need to: 1. Implement DCGAN architecture 2. Train it on CelebA dataset.

We are done with the simple, not-so-challenging MNIST dataset. Now, you need to implement a specific model architecture called DCGAN, and train your model to generate human faces!

4.0.1 Section 2.1. GAN model architecture [20 pts]

Implement your generator and discriminator for generating faces. 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 use 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).

[ ]:
```
# YOUR GENERATOR/DISCRIMINATOR HERE
class Reshape(nn.Module):
  def __init__(self, *args):
    super(Reshape, self).__init__()
    self.shape = args
  def forward(self, x):
    batch_size = x.shape[0]
    noise_dim = x.shape[1]
    return x.view((batch_size,noise_dim,1,1))

def Discriminator():
  model = nn.Sequential(
    torch.nn.
 ↪Conv2d(in_channels=3,out_channels=128,kernel_size=4,stride=2,padding=1),
    torch.nn.LeakyReLU(negative_slope=0.2),
    torch.nn.
 ↪Conv2d(in_channels=128,out_channels=256,kernel_size=4,stride=2,padding=1),
    torch.nn.LeakyReLU(negative_slope=0.2),
    torch.nn.BatchNorm2d(256),
    torch.nn.
 ↪Conv2d(in_channels=256,out_channels=512,kernel_size=4,stride=2,padding=1),
    torch.nn.LeakyReLU(negative_slope=0.2),
    torch.nn.BatchNorm2d(512),
    torch.nn.
 ↪Conv2d(in_channels=512,out_channels=1024,kernel_size=4,stride=2,padding=1),
```

```
    torch.nn.LeakyReLU(negative_slope=0.2),
    torch.nn.BatchNorm2d(1024),
    torch.nn.
 ↪Conv2d(in_channels=1024,out_channels=1,kernel_size=4,stride=1,padding=0),
    torch.nn.LeakyReLU(negative_slope=0.2))
  return model



def Generator(noise_dim=NOISE_DIM):
  model = nn.Sequential(Reshape(),
    torch.nn.
 ↪ConvTranspose2d(in_channels=NOISE_DIM,out_channels=1024,kernel_size=4,stride=1,padding=0),
    torch.nn.ReLU(),
    torch.nn.BatchNorm2d(1024),
    torch.nn.
 ↪ConvTranspose2d(in_channels=1024,out_channels=512,kernel_size=4,stride=2,padding=1),
    torch.nn.ReLU(),
    torch.nn.BatchNorm2d(512),
    torch.nn.
 ↪ConvTranspose2d(in_channels=512,out_channels=256,kernel_size=4,stride=2,padding=1),
    torch.nn.ReLU(),
    torch.nn.BatchNorm2d(256),
    torch.nn.
 ↪ConvTranspose2d(in_channels=256,out_channels=128,kernel_size=4,stride=2,padding=1),
    torch.nn.ReLU(),
    torch.nn.BatchNorm2d(128),
    torch.nn.
 ↪ConvTranspose2d(in_channels=128,out_channels=3,kernel_size=4,stride=2,padding=1),
    torch.nn.Tanh())
  return model
```

4.0.2 Section 2.2 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.

Run download_celeba.sh to get dataset.

[ ]:
```
!wget https://uofi.box.com/shared/static/q4pf89jtkvjndi4f8ip7wofuulhhphjj.zip
!mkdir celeba_data
!unzip q4pf89jtkvjndi4f8ip7wofuulhhphjj.zip -d celeba_data
!rm q4pf89jtkvjndi4f8ip7wofuulhhphjj.zip
!cp -r /content/celeba_data /content/drive/My\ Drive/DL_Fall_2020/Assignment_5/
```

[ ]:
```
import sys
```

```
sys.path, dirs, files = next(os.walk("/content/drive/My Drive/DL_Fall_2020/
 ↪Assignment_5/celeba_data/celeba_train_128res"))
file_count = len(files)
print(file_count)
```

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

celeba_root = 'celeba_data'
```

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

# You can change the num_workers to speed up loading
celeba_loader_train = DataLoader(celeba_train, batch_size=batch_size,␣
 ↪drop_last=True)
```

Visualize dataset

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

4.0.3 Section 2.3 Train a Vanilla GAN on CelebA [13 pts]

- Call discriminator and generator for training.
- Call optimizers for both discriminator and generator for training. (Use Adam with betas = (0.5, 0.999))
- Call train function to train.
- Train for 30 epochs.

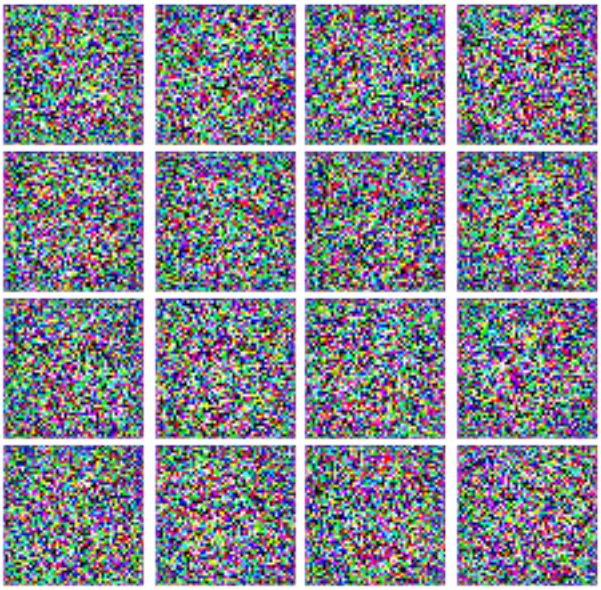
Now, train your GAN model with vanilla GAN loss. If you models are implemented correctly, you should see something like this:

Now, train your model. **Observe the visualized result of your model, and describe what you see.**

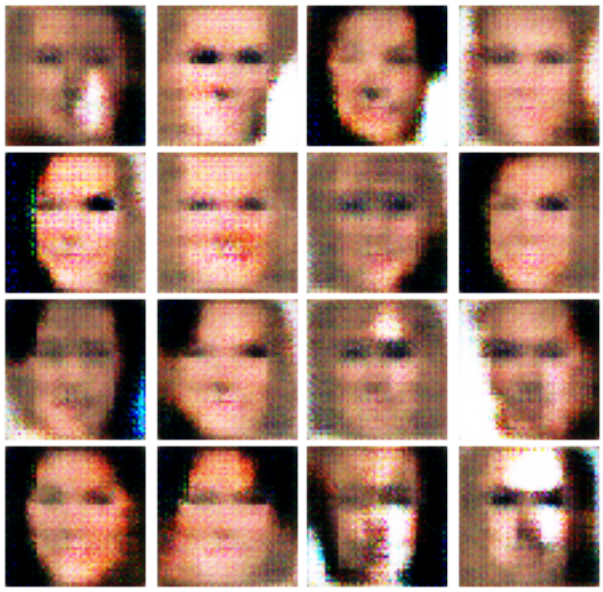
[ ]:
```
NOISE_DIM = 100
NUM_EPOCHS = 30
learning_rate = 0.0002
```

[ ]:
```
# original GAN
# Add code here:
Discriminator = Discriminator().to(device)
Generator = Generator(noise_dim=NOISE_DIM).to(device)

Discriminator_optimizer = torch.optim.Adam(Discriminator.parameters(),␣
 ↪lr=learning_rate, betas = (0.5, 0.999))
Generator_optimizer = torch.optim.Adam(Generator.parameters(), lr=learning_rate,␣
 ↪betas = (0.5, 0.999))

train(Discriminator, Generator, Discriminator_optimizer, Generator_optimizer,␣
 ↪discriminator_loss, generator_loss, num_epochs=NUM_EPOCHS,␣
 ↪train_loader=celeba_loader_train, device=device)
```

```
EPOCH:  1
Iter: 0, D: 1.438, G:1.058
```



```
Iter: 250, D: 0.7416, G:0.933
```



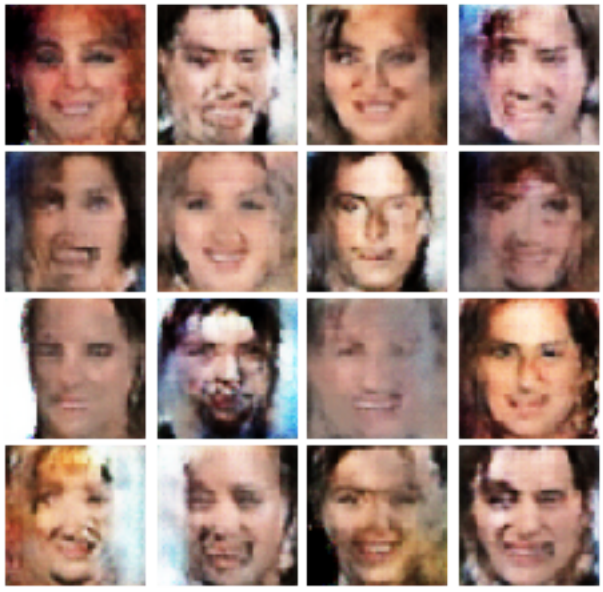
```
Iter: 500, D: 1.197, G:2.466
```

```
Iter: 750, D: 1.218, G:2.011
```

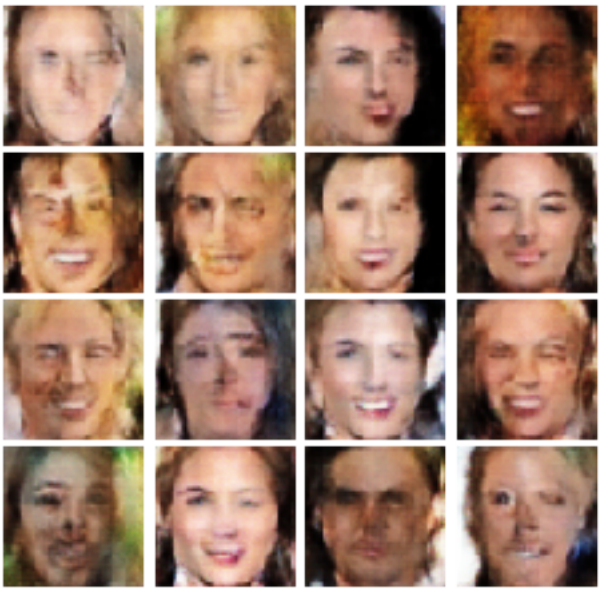
```
Saving the model as a checkpoint...
EPOCH:  2
Iter: 1000, D: 1.706, G:1.731
```

```
Iter: 1250, D: 1.266, G:1.305
```

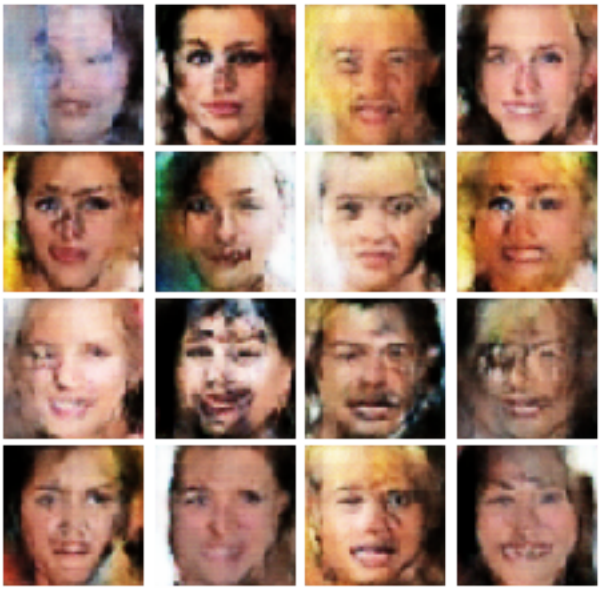
```
Iter: 1500, D: 1.156, G:0.8481
```



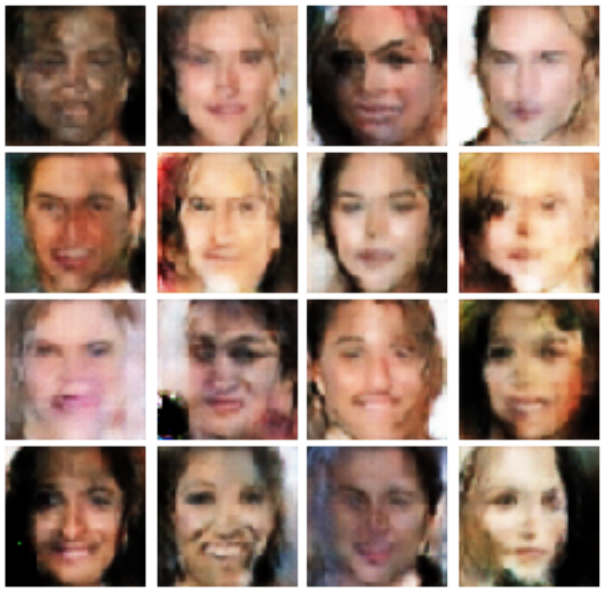
```
Iter: 1750, D: 1.149, G:0.9253
```



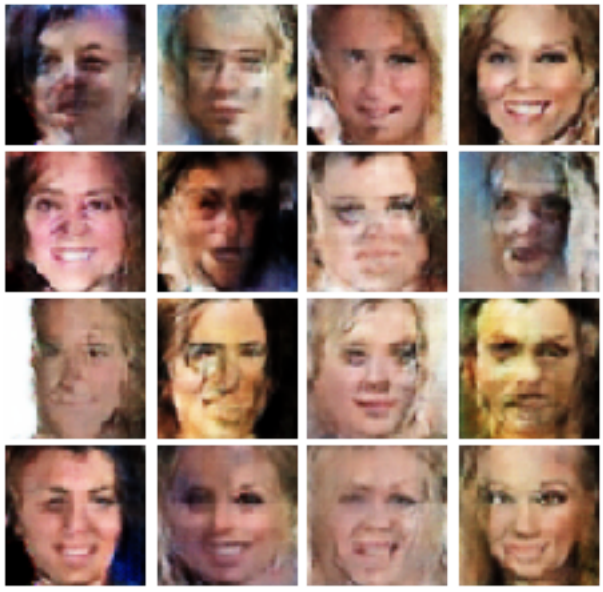
```
Saving the model as a checkpoint...
EPOCH:  3
Iter: 2000, D: 1.414, G:0.9775
```



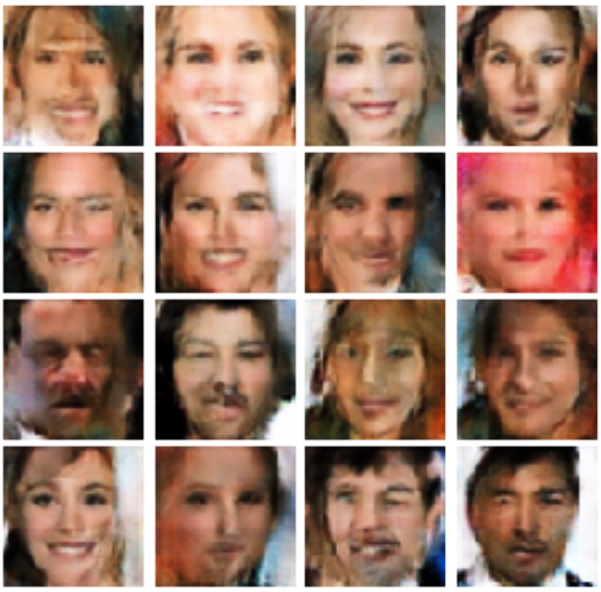
```
Iter: 2250, D: 0.9734, G:0.8645
```



Iter: 2500, D: 1.247, G:1.016

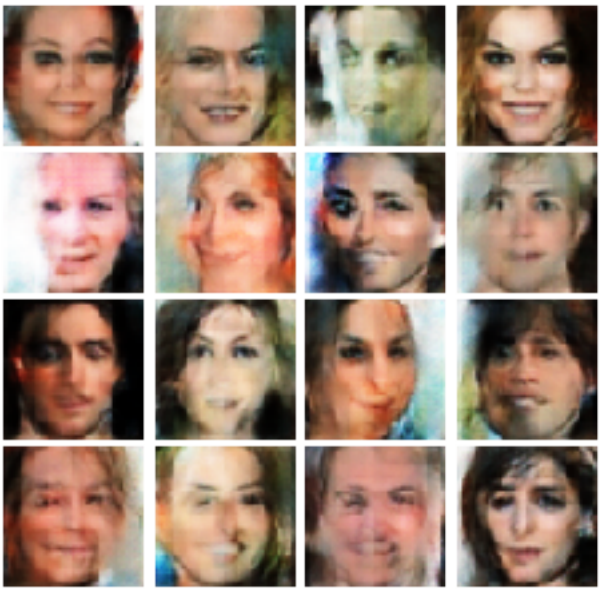


Iter: 2750, D: 1.058, G:1.077

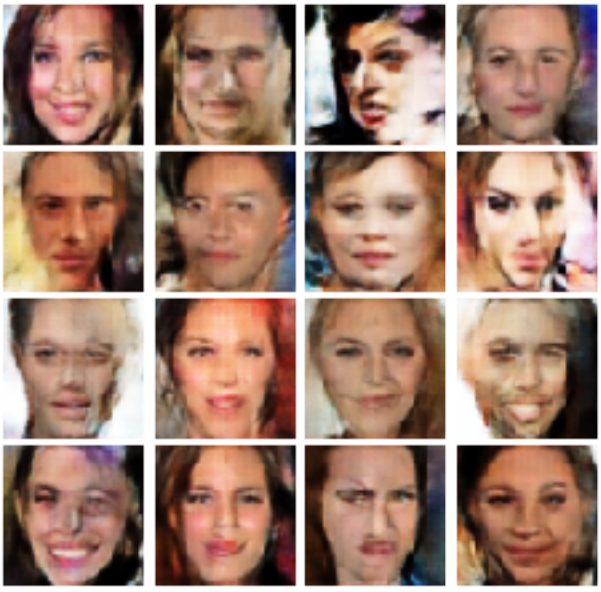


```
Saving the model as a checkpoint...
EPOCH:  4
Iter: 3000, D: 1.517, G:0.9261
```

```
Iter: 3250, D: 1.386, G:0.8513
```

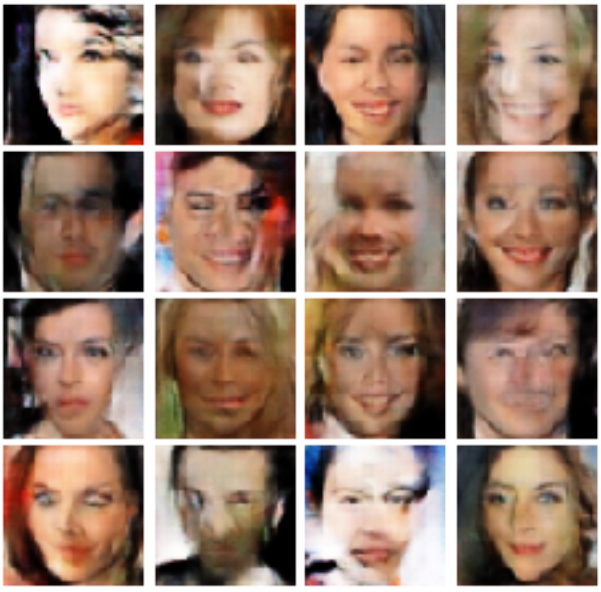


Iter: 3500, D: 1.495, G:1.468



Iter: 3750, D: 1.251, G:1.161

```
Saving the model as a checkpoint...
EPOCH:  5
Iter: 4000, D: 1.39, G:0.9363
```



```
Iter: 4250, D: 1.445, G:0.82
```

Iter: 4500, D: 1.227, G:1.336

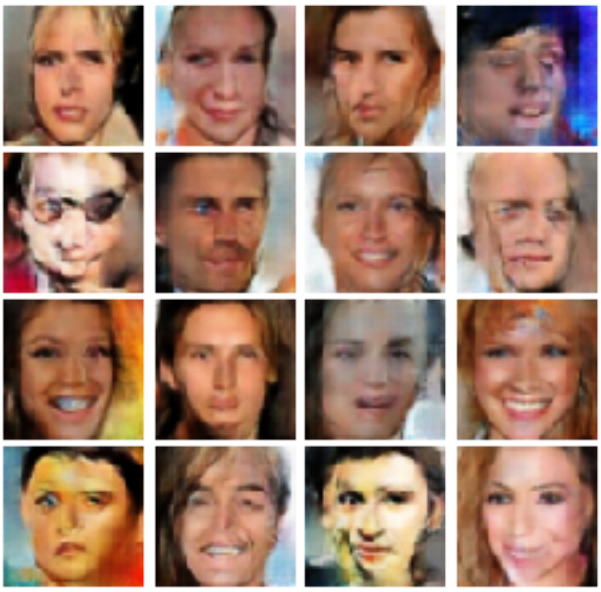
Iter: 4750, D: 1.289, G:1.154

```
Saving the model as a checkpoint...
EPOCH:  6
Iter: 5000, D: 1.446, G:1.214
```

```
Iter: 5250, D: 1.46, G:0.9585
```

Iter: 5500, D: 1.572, G:1.309

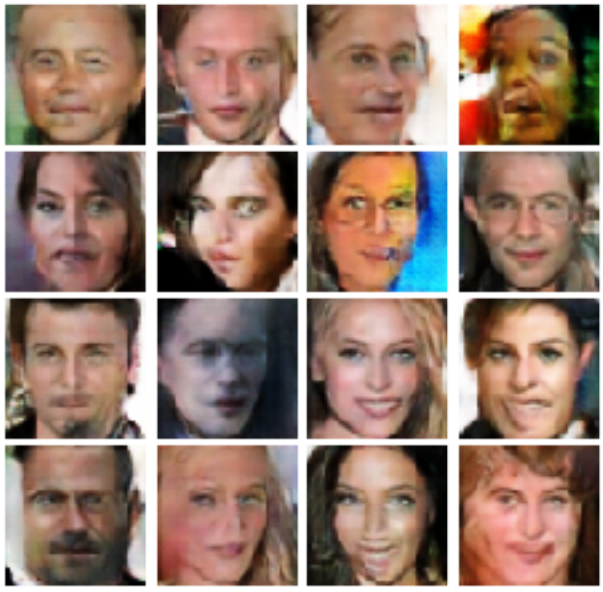
Iter: 5750, D: 1.429, G:1.159



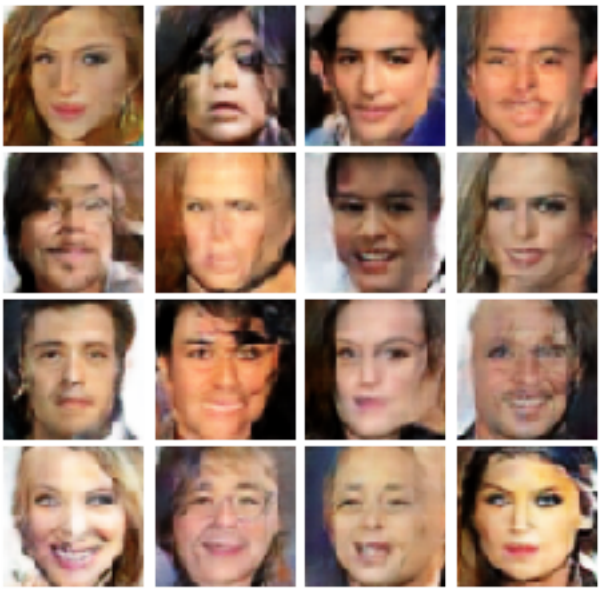
```
Saving the model as a checkpoint...
EPOCH:  7
Iter: 6000, D: 1.712, G:2.163
```

```
Iter: 6250, D: 0.9849, G:0.7809
```

Iter: 6500, D: 1.062, G:0.8662



Iter: 6750, D: 1.388, G:0.8857

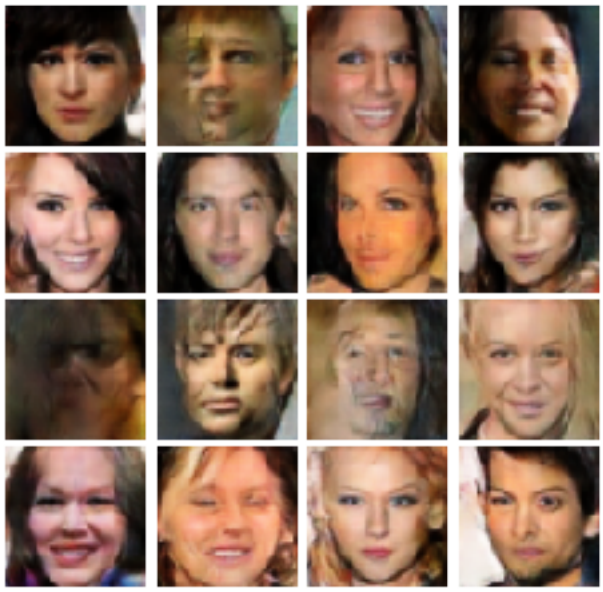


```
Saving the model as a checkpoint...
EPOCH:  8
Iter: 7000, D: 1.056, G:0.7804
```

```
Iter: 7250, D: 0.9631, G:1.183
```

Iter: 7500, D: 1.031, G:0.7642

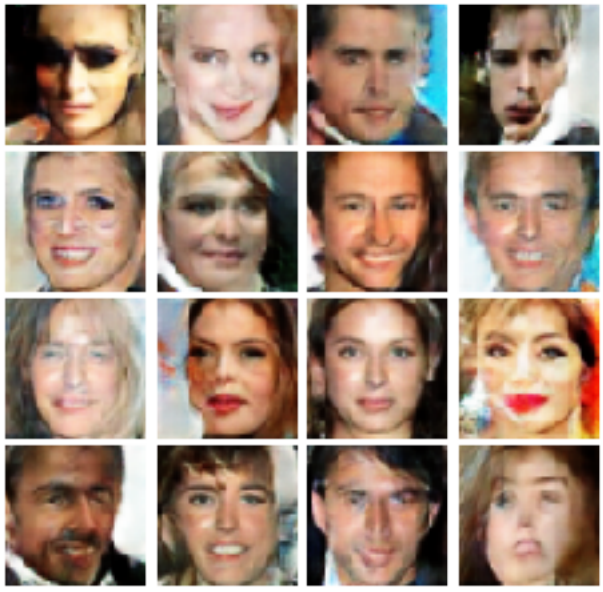
Iter: 7750, D: 1.138, G:0.6318



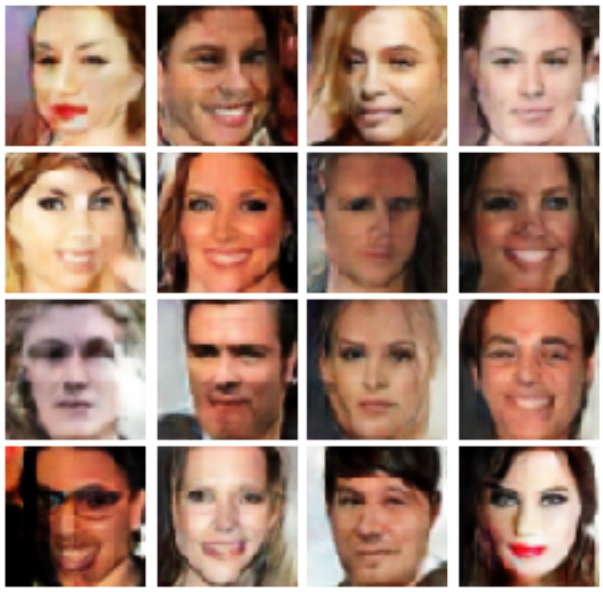
```
Saving the model as a checkpoint...
EPOCH:  9
Iter: 8000, D: 1.162, G:1.012
```

```
Iter: 8250, D: 1.024, G:0.8713
```

Iter: 8500, D: 1.417, G:1.381



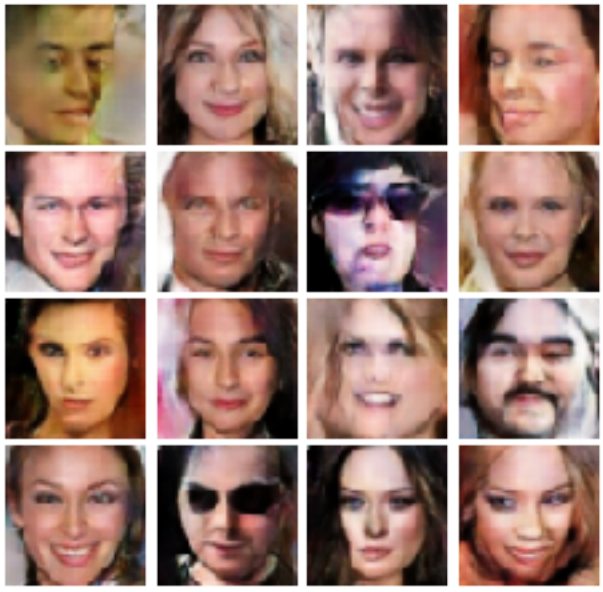
Iter: 8750, D: 0.9345, G:0.9742



```
Saving the model as a checkpoint...
EPOCH:  10
Iter: 9000, D: 1.233, G:1.648
```

```
Iter: 9250, D: 0.801, G:1.101
```

Iter: 9500, D: 1.258, G:1.537

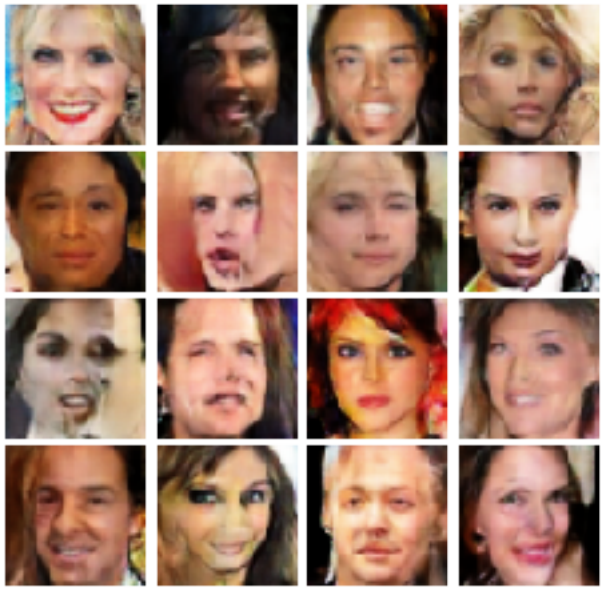


Iter: 9750, D: 0.9569, G:0.6229

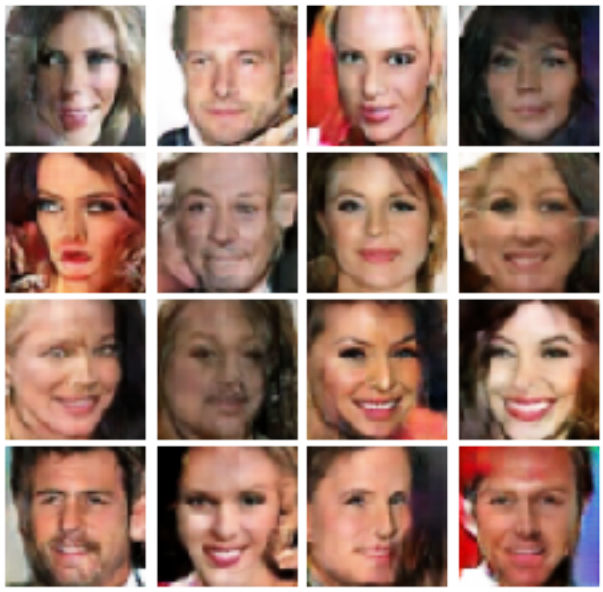
```
Saving the model as a checkpoint...
EPOCH:  11
Iter: 10000, D: 0.9802, G:1.189
```

```
Iter: 10250, D: 1.071, G:1.895
```

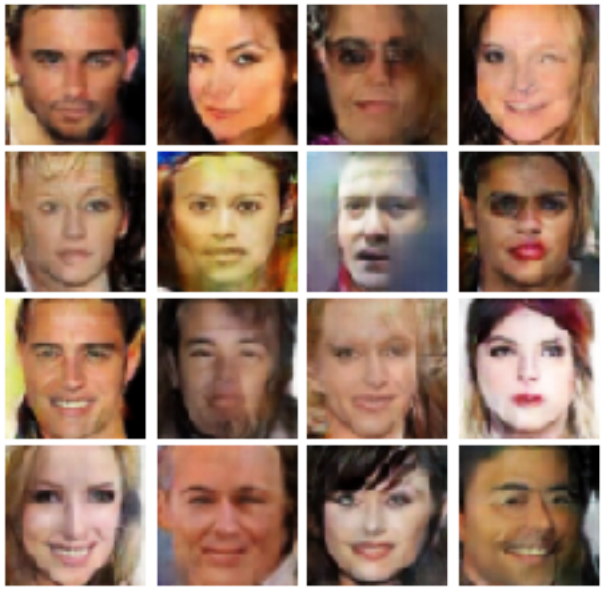
Iter: 10500, D: 0.7206, G:1.142



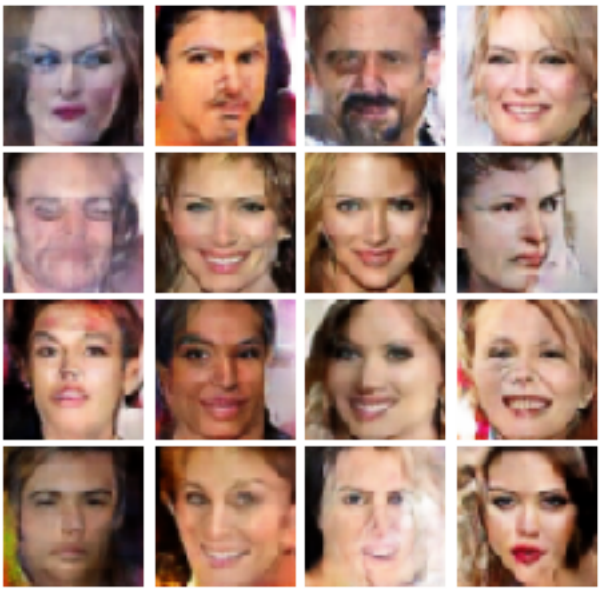
Iter: 10750, D: 0.6457, G:0.9265



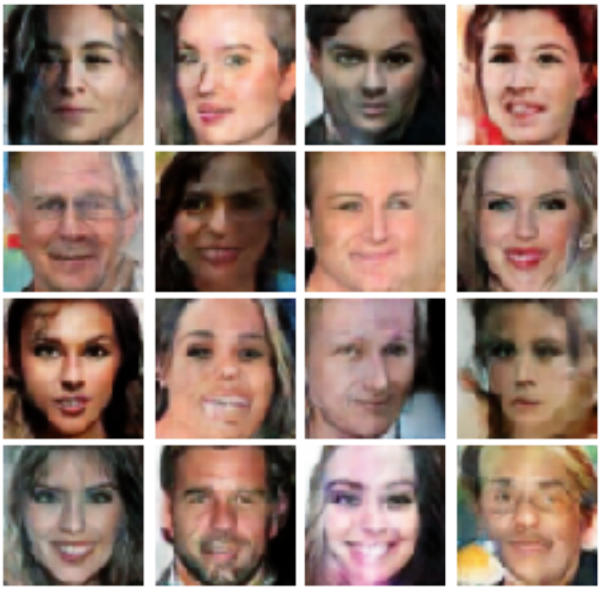
```
Saving the model as a checkpoint...
EPOCH:  12
Iter: 11000, D: 0.9569, G:0.9214
```



```
Iter: 11250, D: 1.012, G:0.92
```

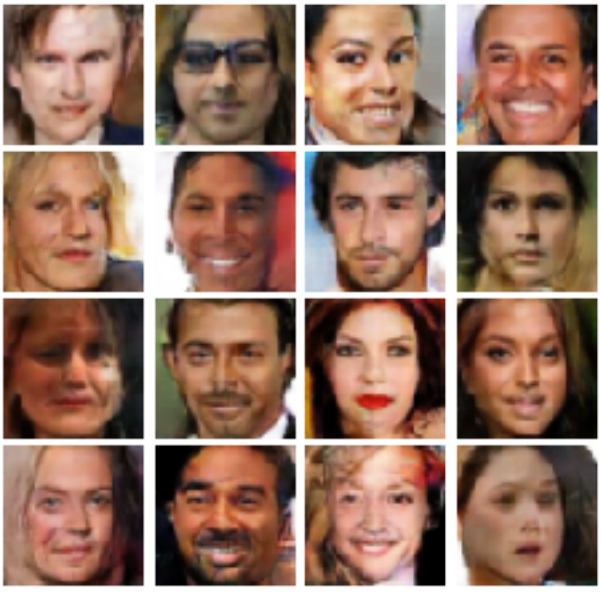


Iter: 11500, D: 0.5924, G:1.234



Iter: 11750, D: 0.406, G:1.574

```
Saving the model as a checkpoint...
EPOCH:  13
Iter: 12000, D: 0.5151, G:1.596
```

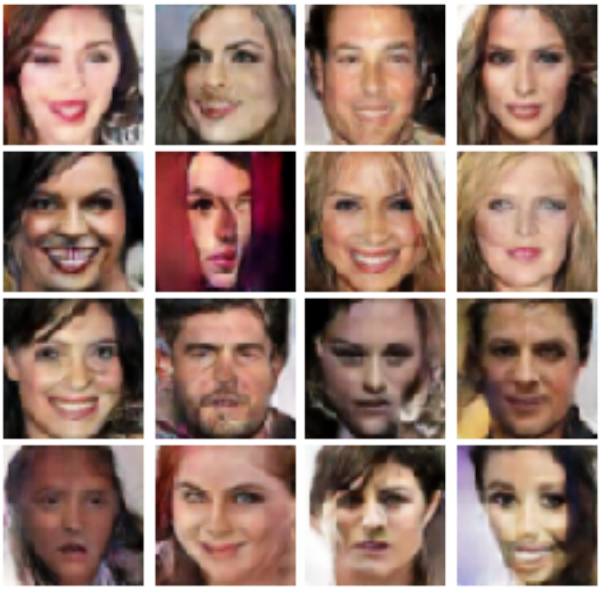


```
Iter: 12250, D: 1.186, G:3.003
```

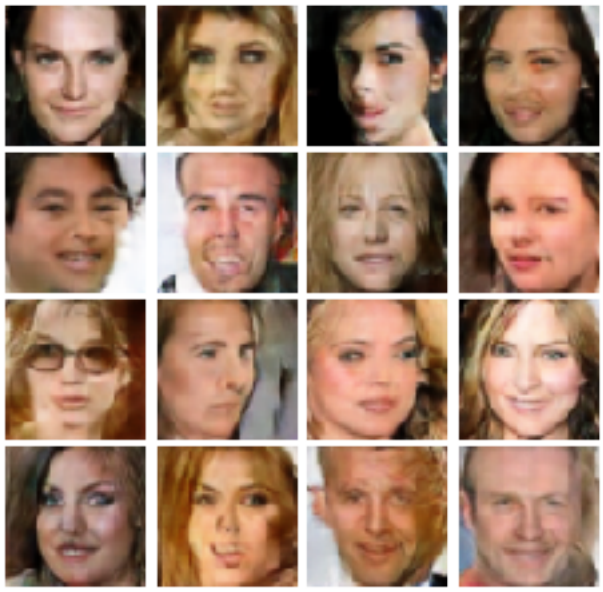
Iter: 12500, D: 0.4939, G:1.223

Iter: 12750, D: 0.5383, G:1.524

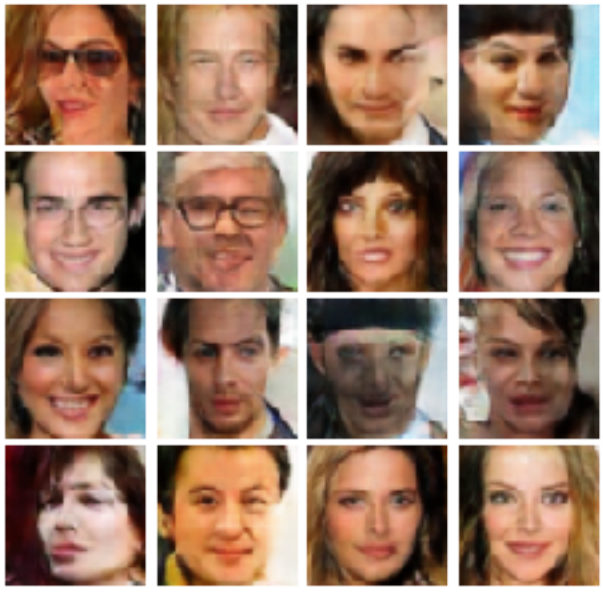
```
Saving the model as a checkpoint...
EPOCH:  14
Iter: 13000, D: 0.3773, G:1.71
```



```
Iter: 13250, D: 0.5989, G:1.105
```

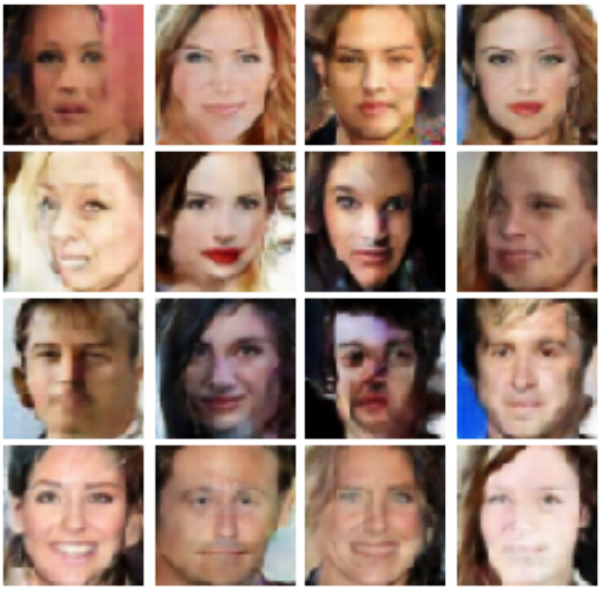


```
Iter: 13500, D: 0.5765, G:0.9919
```

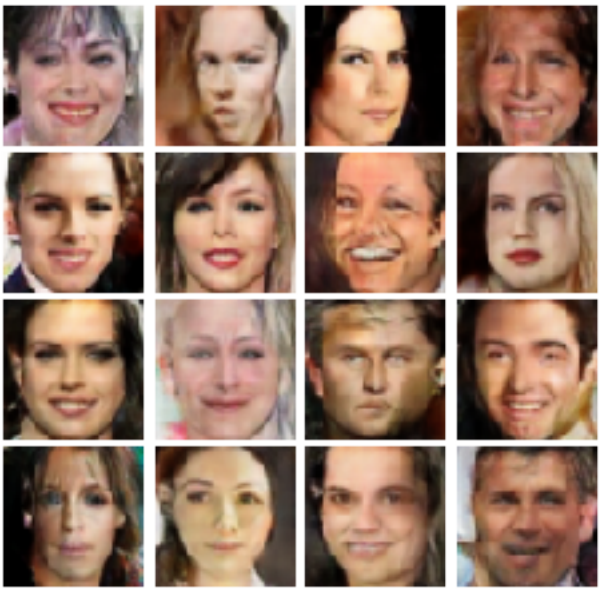


```
Iter: 13750, D: 0.2428, G:1.466
```

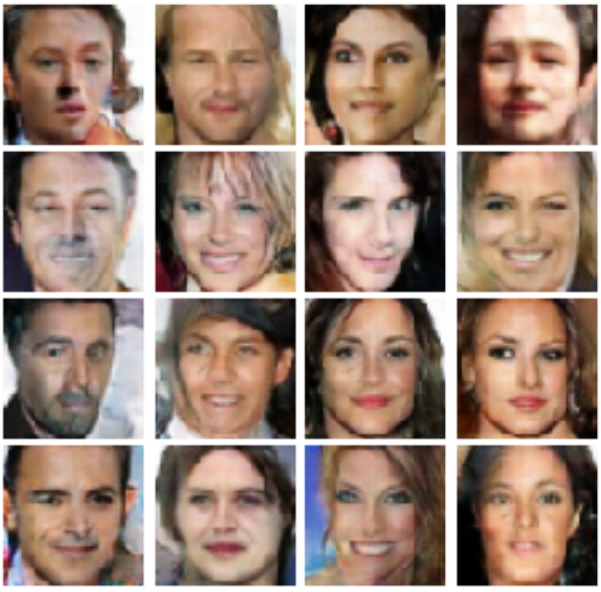
```
Saving the model as a checkpoint...
EPOCH:  15
Iter: 14000, D: 1.359, G:2.212
```



```
Iter: 14250, D: 0.259, G:1.784
```

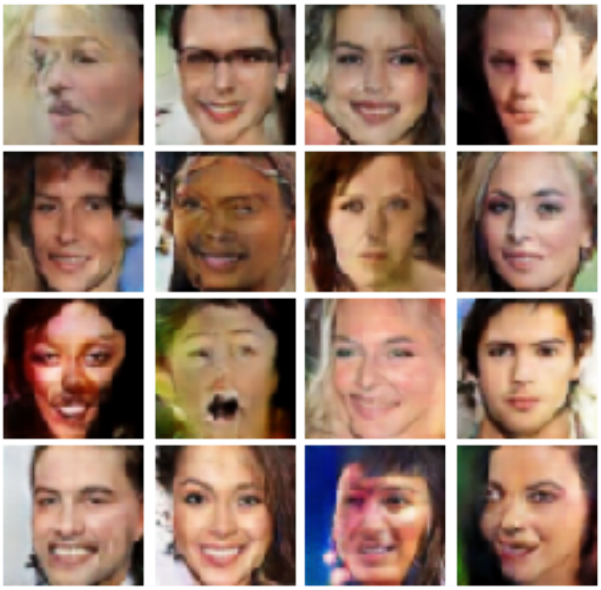


Iter: 14500, D: 0.6733, G:1.474



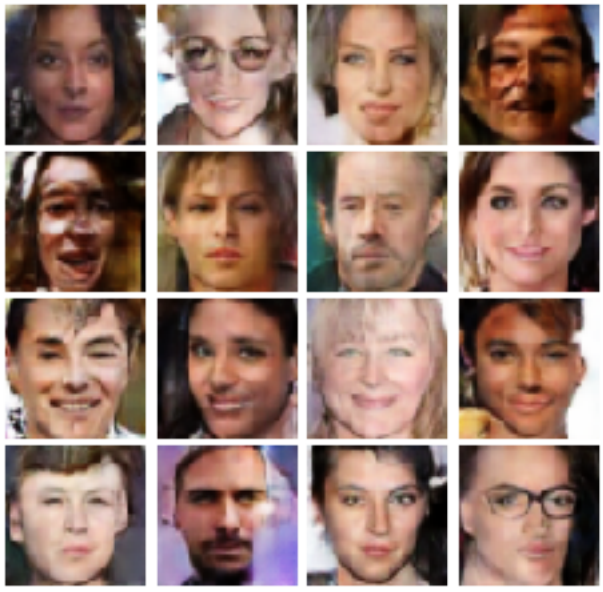
Iter: 14750, D: 0.5327, G:1.539

```
Saving the model as a checkpoint...
EPOCH:  16
Iter: 15000, D: 0.263, G:2.219
```

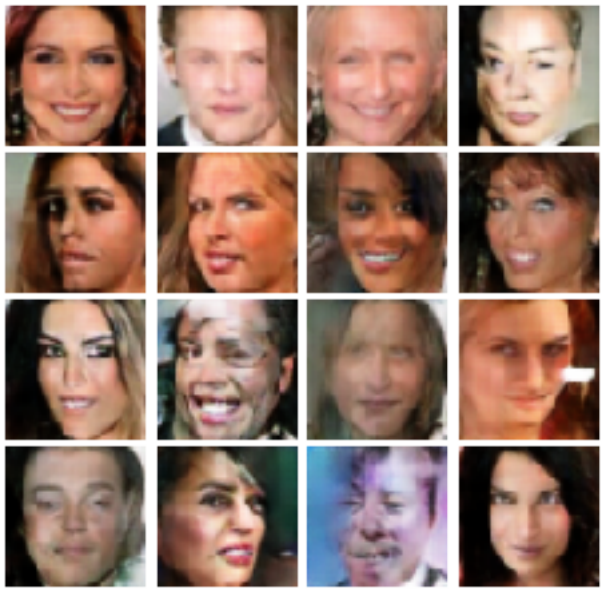


```
Iter: 15250, D: 0.3132, G:1.481
```

Iter: 15500, D: 0.2866, G:1.917



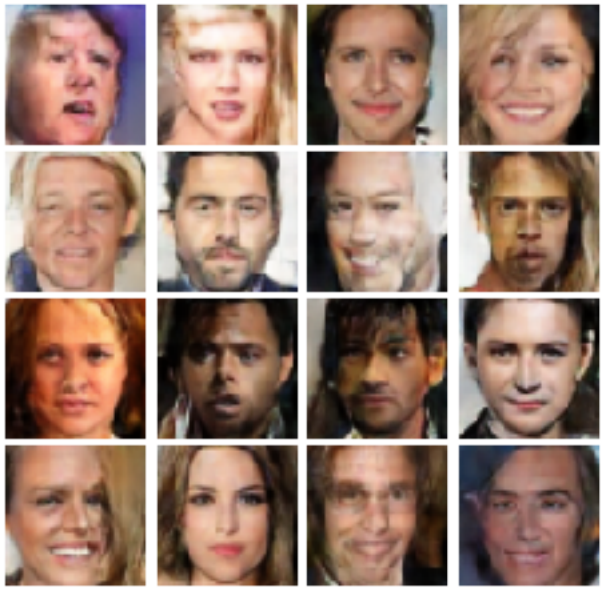
Iter: 15750, D: 0.9266, G:0.8389



```
Saving the model as a checkpoint...
EPOCH:  17
Iter: 16000, D: 0.2755, G:1.463
```

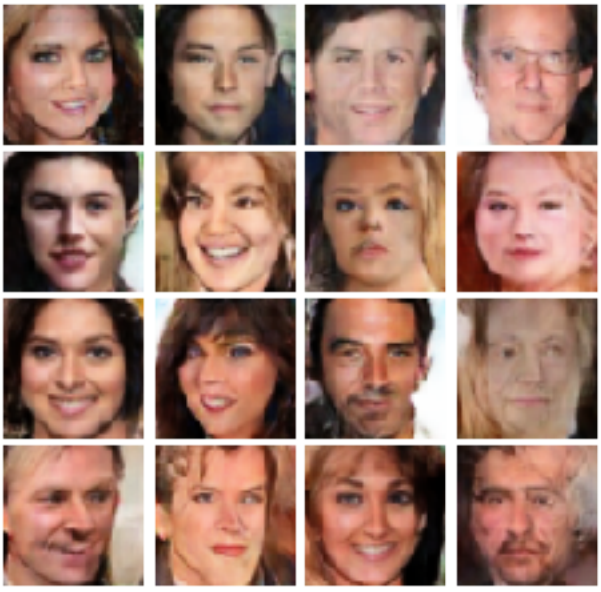
```
Iter: 16250, D: 2.11, G:3.539
```

Iter: 16500, D: 0.1331, G:2.292

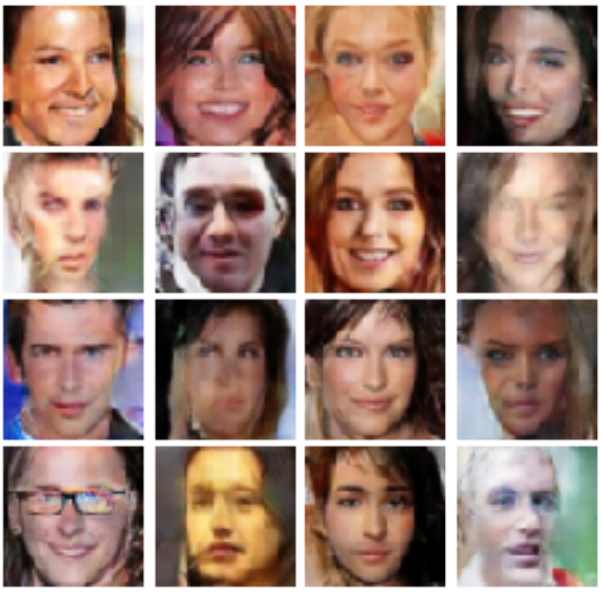


Iter: 16750, D: 0.4863, G:2.086

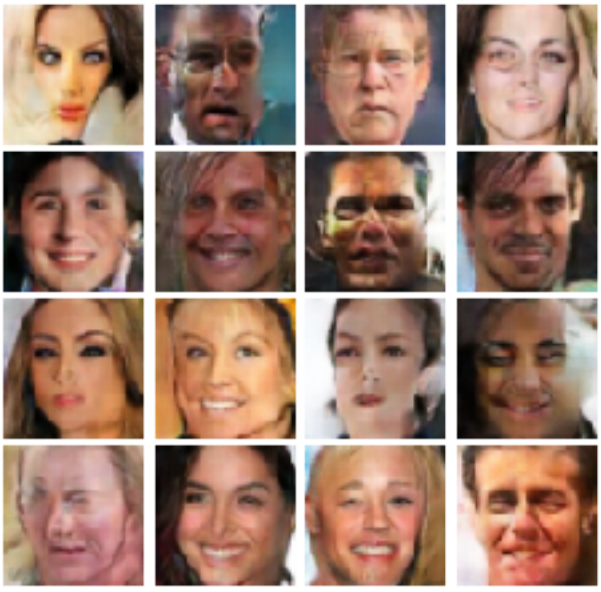
```
Saving the model as a checkpoint...
EPOCH:  18
Iter: 17000, D: 0.1523, G:2.005
```



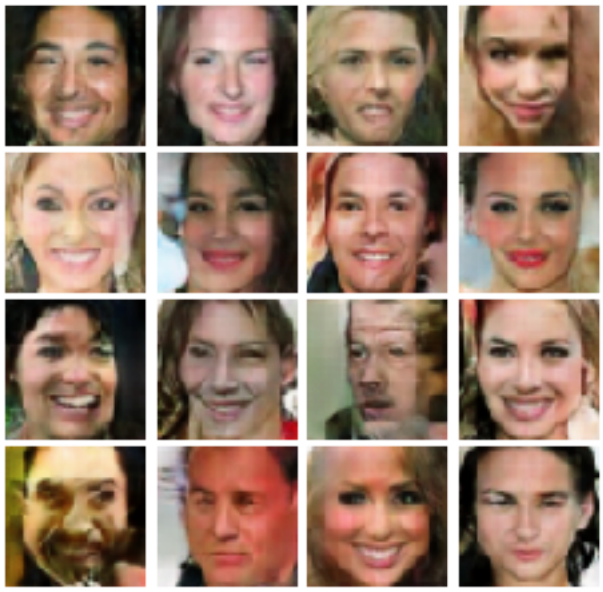
```
Iter: 17250, D: 0.3748, G:1.875
```



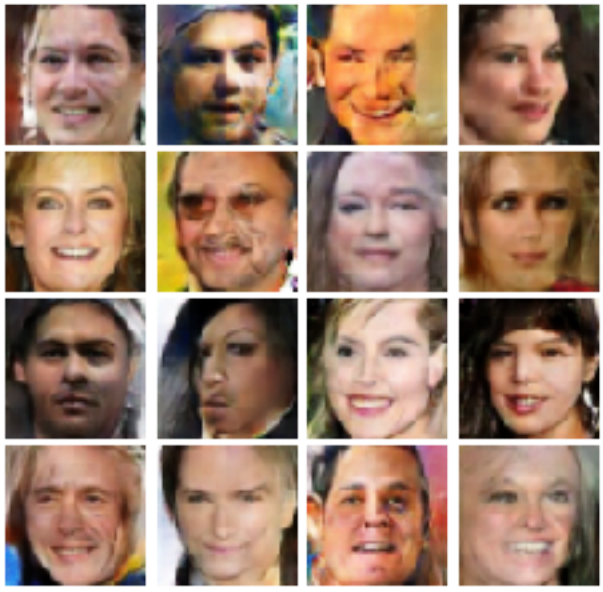
```
Iter: 17500, D: 0.7205, G:0.6492
```



```
Saving the model as a checkpoint...
EPOCH:  19
Iter: 17750, D: 0.207, G:2.199
```



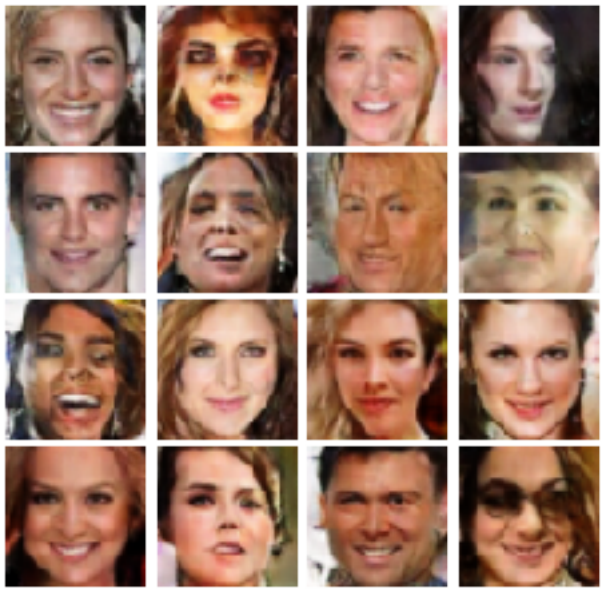
Iter: 18000, D: 0.1269, G:2.616



Iter: 18250, D: 0.3044, G:2.129

```
Iter: 18500, D: 0.7016, G:2.016
```

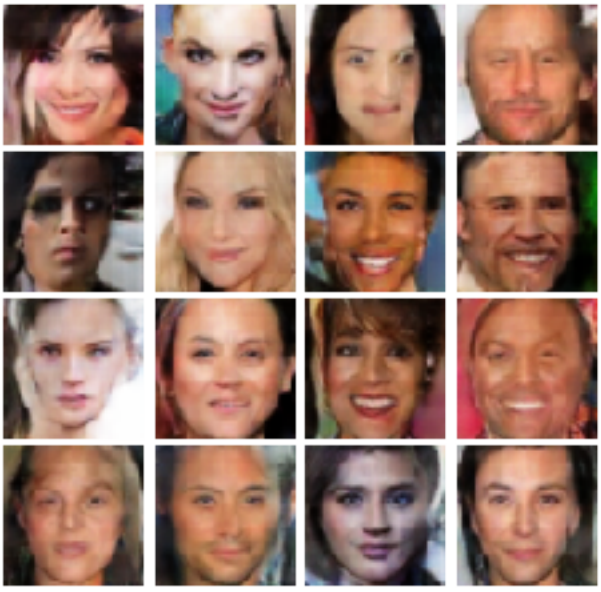
```
Saving the model as a checkpoint...
EPOCH:  20
Iter: 18750, D: 0.2275, G:2.011
```



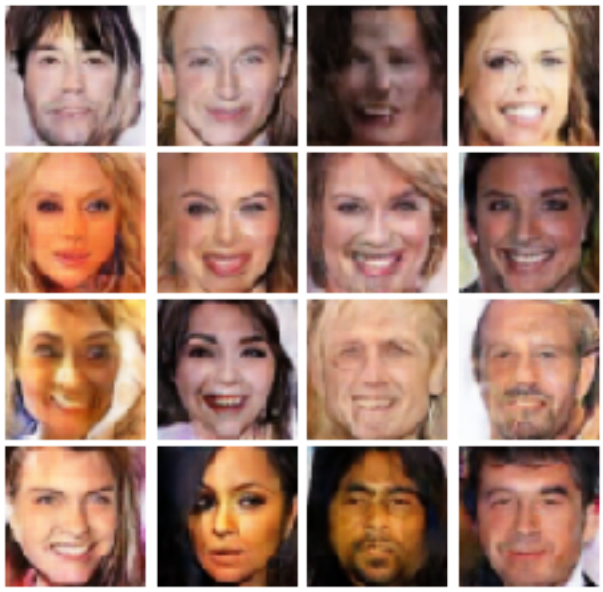
Iter: 19000, D: 0.167, G:2.663

Iter: 19250, D: 0.2132, G:1.849

```
Iter: 19500, D: 0.8389, G:0.7802
```

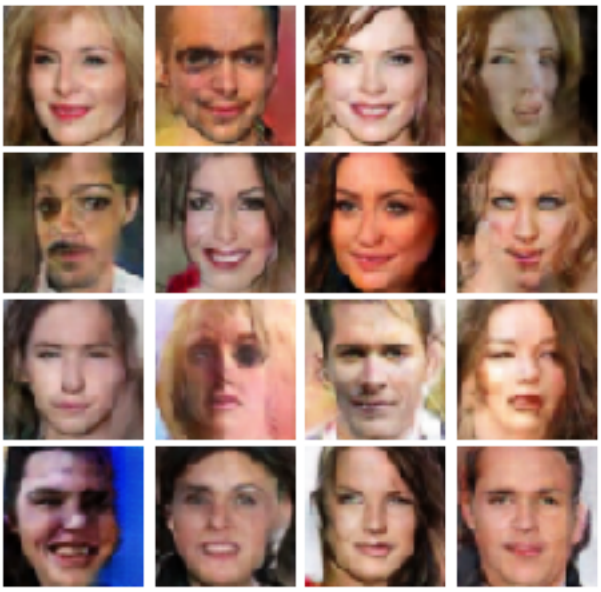


```
Saving the model as a checkpoint...
EPOCH:  21
Iter: 19750, D: 0.6588, G:1.82
```



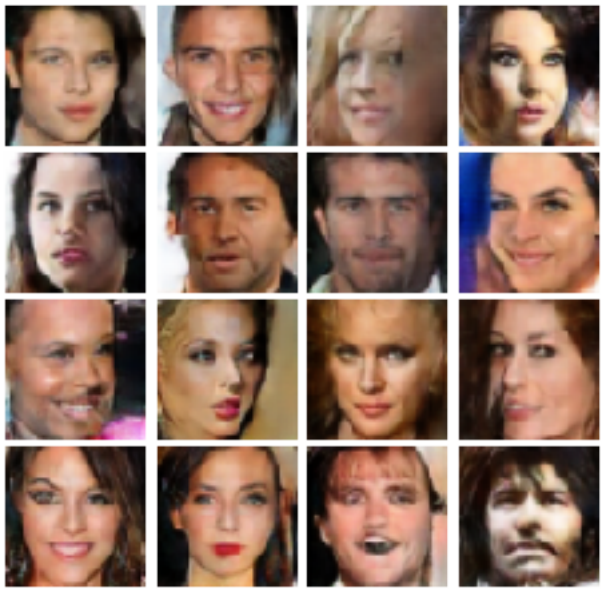
Iter: 20000, D: 0.08408, G:3.15

Iter: 20250, D: 0.2082, G:1.716

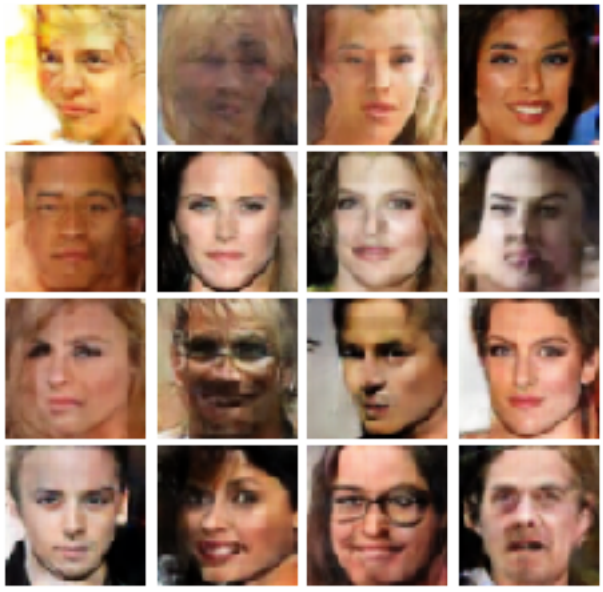


```
Iter: 20500, D: 0.1889, G:2.296
```

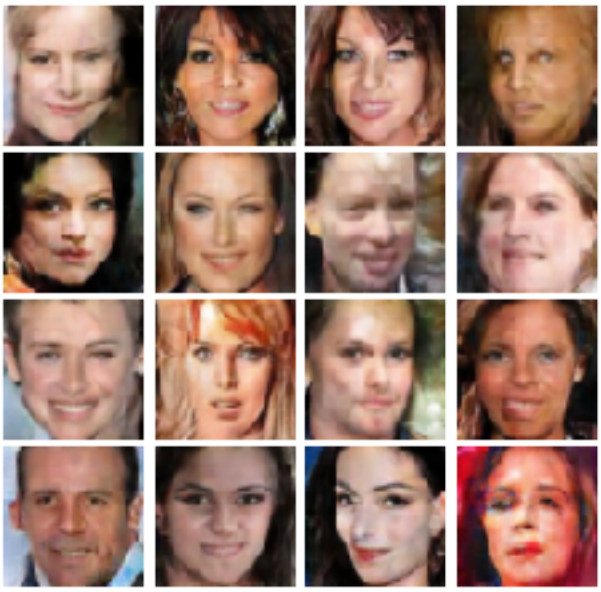
```
Saving the model as a checkpoint...
EPOCH:  22
Iter: 20750, D: 0.2209, G:2.381
```



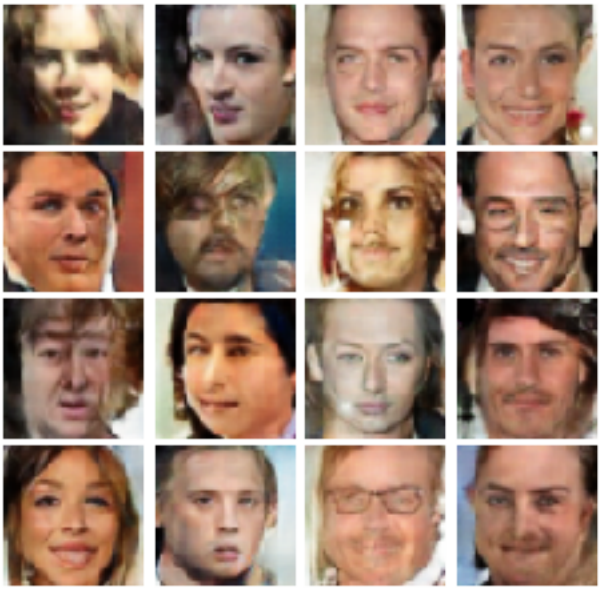
Iter: 21000, D: 0.5584, G:1.91



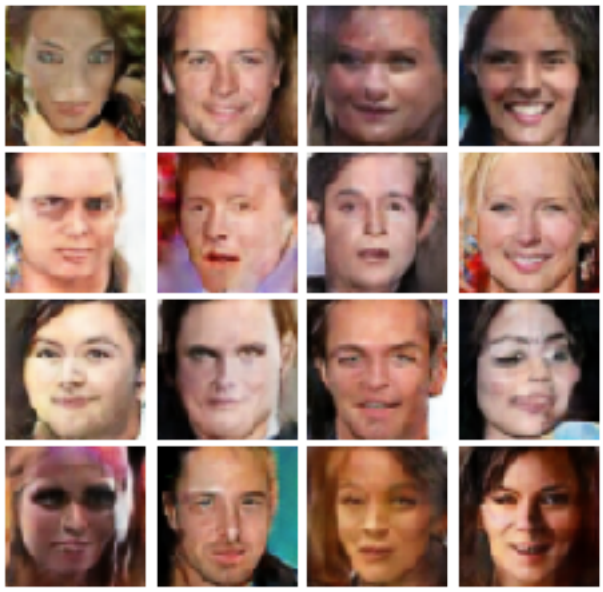
Iter: 21250, D: 0.2062, G:2.404



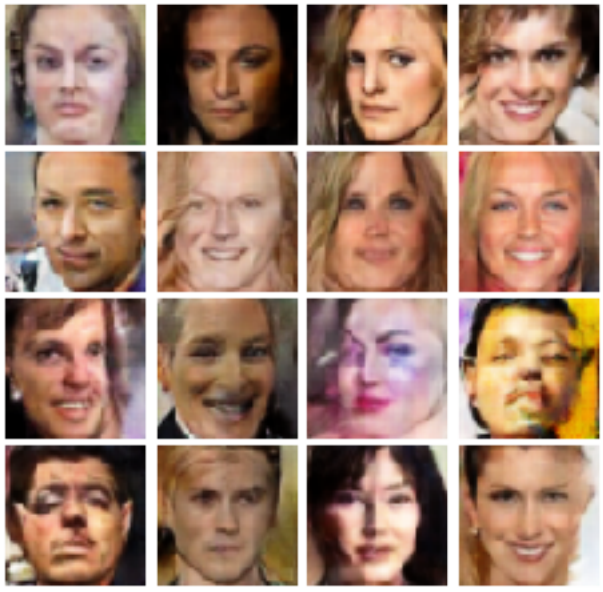
```
Iter: 21500, D: 0.2445, G:2.333
```



```
Saving the model as a checkpoint...
EPOCH:  23
Iter: 21750, D: 0.15, G:2.216
```



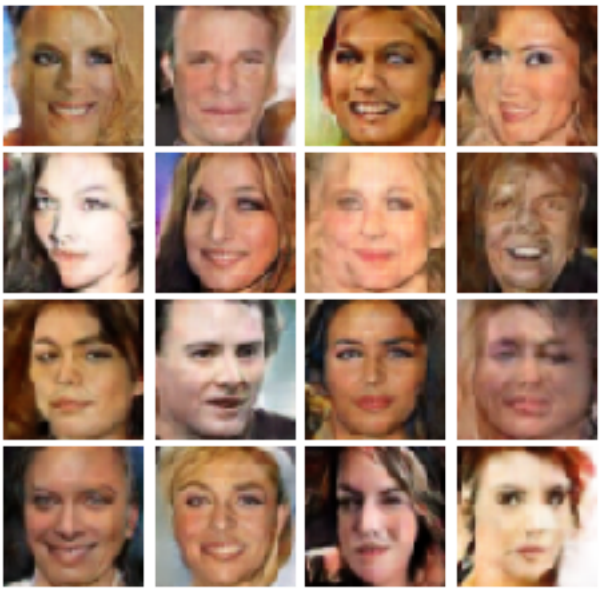
Iter: 22000, D: 0.296, G:2.576



Iter: 22250, D: 0.4117, G:2.505

```
Iter: 22500, D: 0.6837, G:0.5808
```

```
Saving the model as a checkpoint...
EPOCH:  24
Iter: 22750, D: 0.4137, G:2.356
```



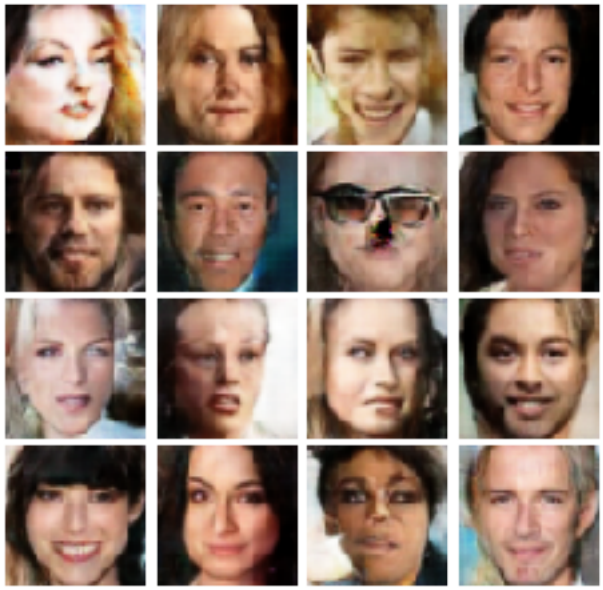
Iter: 23000, D: 0.1193, G:2.731

Iter: 23250, D: 1.676, G:2.122

```
Iter: 23500, D: 0.1137, G:3.327
```

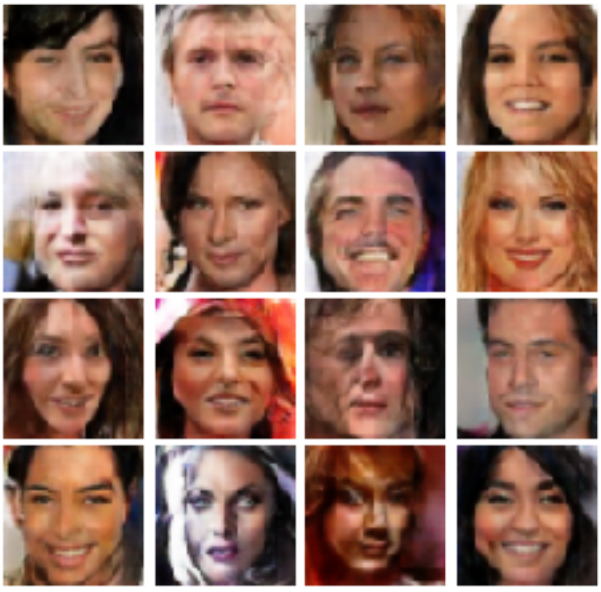
```
Saving the model as a checkpoint...
EPOCH:  25
Iter: 23750, D: 0.2707, G:2.544
```

Iter: 24000, D: 1.013, G:1.25

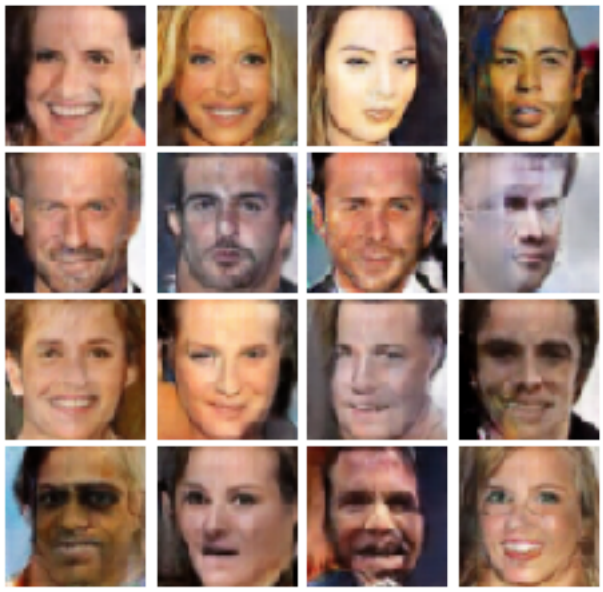


Iter: 24250, D: 0.1149, G:2.67

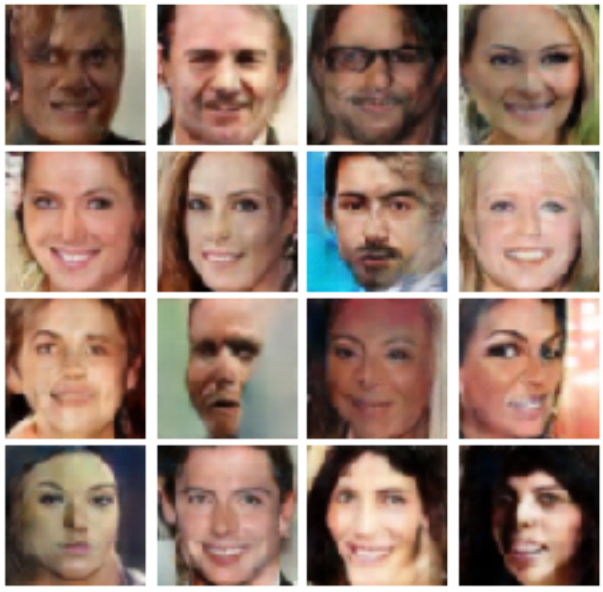
```
Iter: 24500, D: 0.245, G:2.074
```



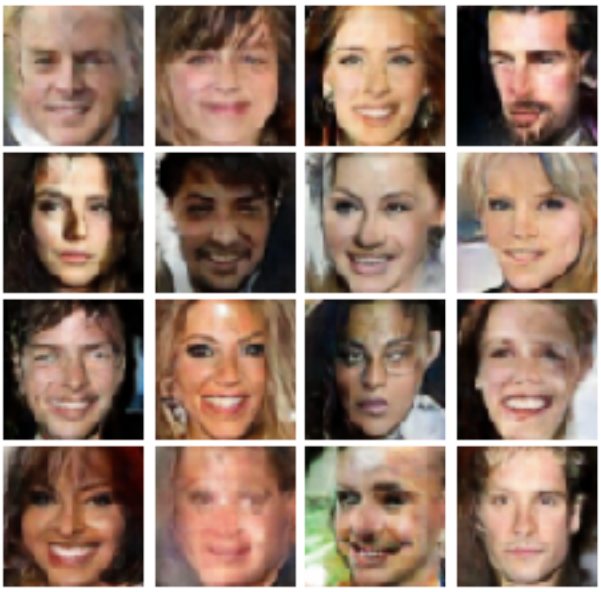
```
Saving the model as a checkpoint...
EPOCH:  26
Iter: 24750, D: 0.1378, G:2.561
```



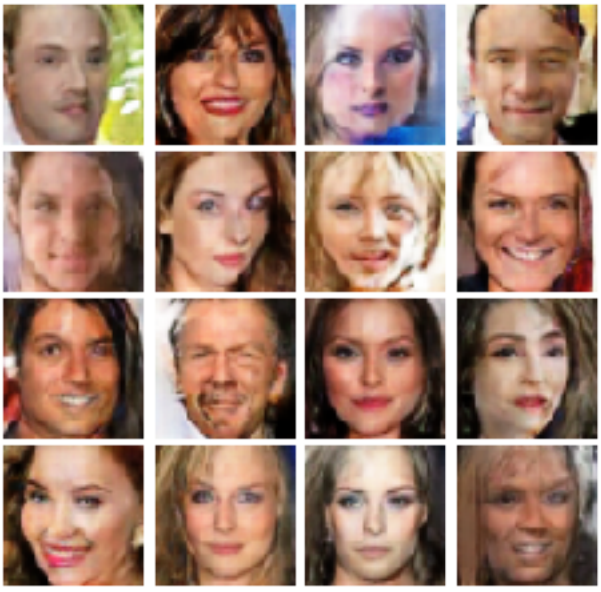
Iter: 25000, D: 0.08143, G:3.013



Iter: 25250, D: 0.1813, G:2.756



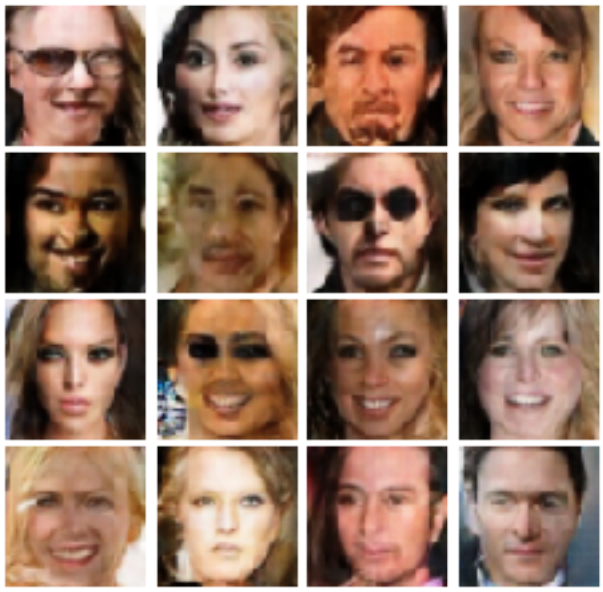
```
Iter: 25500, D: 0.2371, G:1.915
```



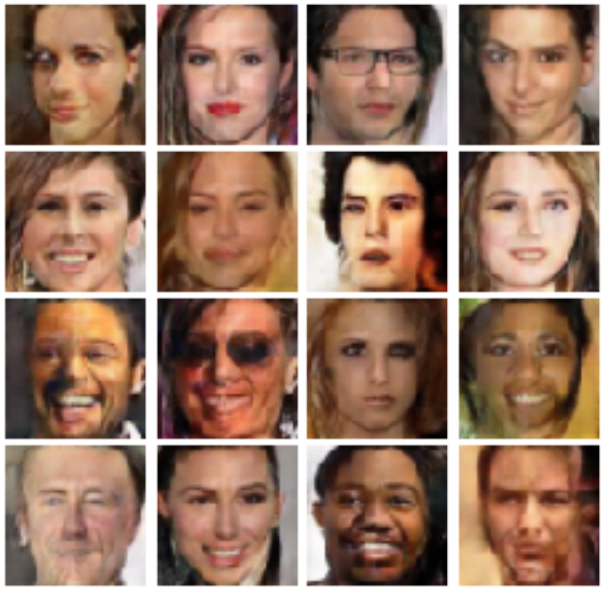
```
Saving the model as a checkpoint...
EPOCH:  27
Iter: 25750, D: 0.2815, G:2.083
```

Iter: 26000, D: 0.9508, G:1.187

Iter: 26250, D: 0.6689, G:2.835



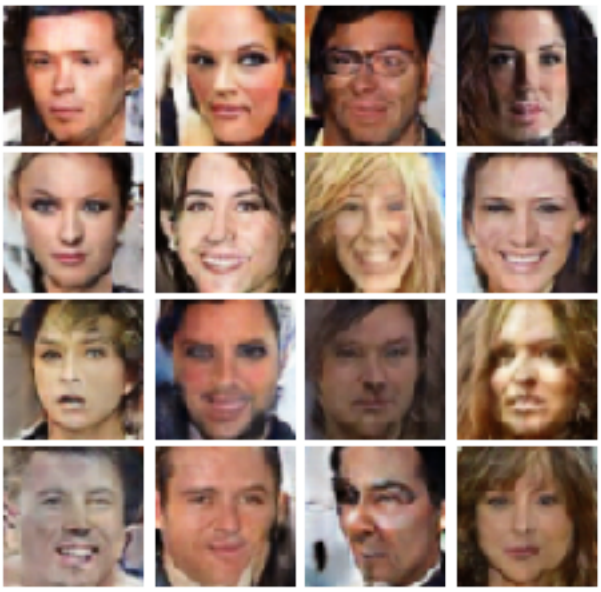
```
Iter: 26500, D: 0.2072, G:2.16
```



```
Saving the model as a checkpoint...
EPOCH:  28
Iter: 26750, D: 0.06536, G:3.777
```

Iter: 27000, D: 0.2835, G:2.045

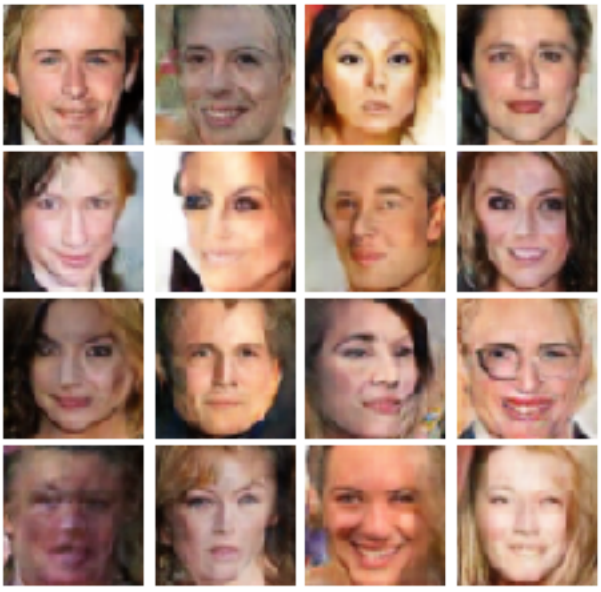
Iter: 27250, D: 0.1272, G:2.954



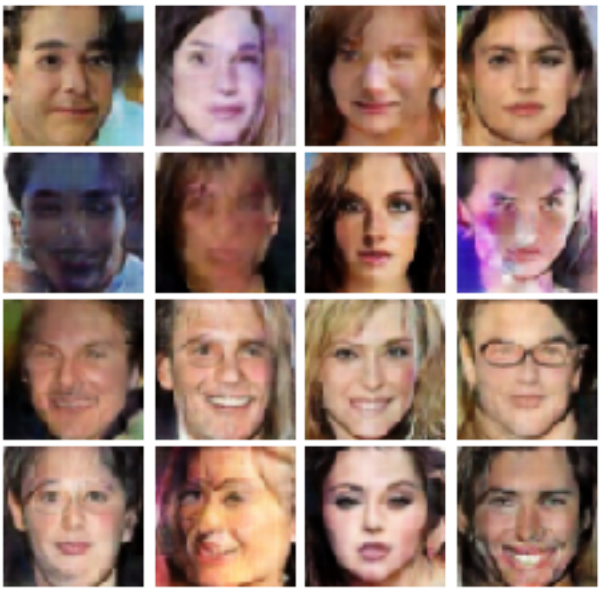
```
Iter: 27500, D: 0.2794, G:1.757
```

```
Saving the model as a checkpoint...
EPOCH:  29
Iter: 27750, D: 0.1335, G:2.786
```

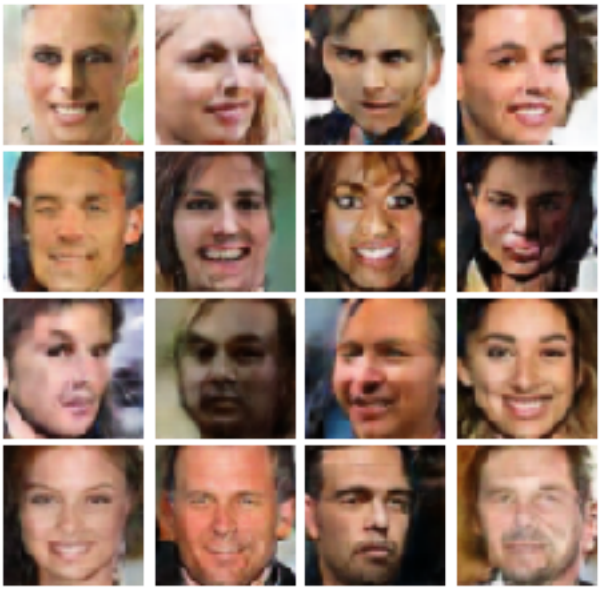
Iter: 28000, D: 0.0718, G:3.895



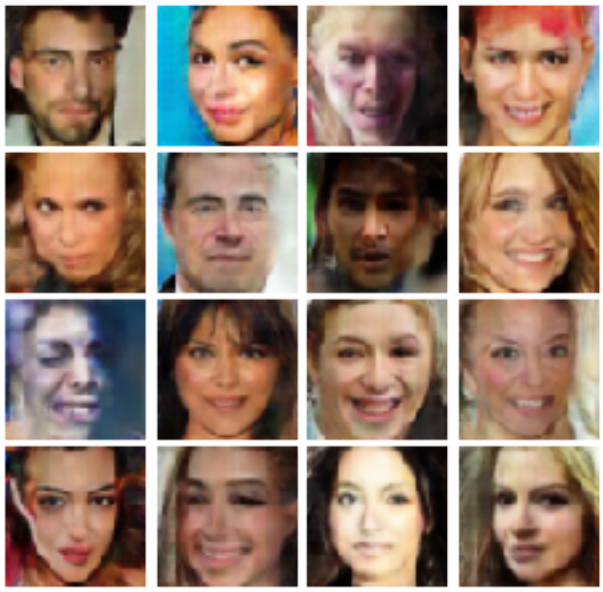
Iter: 28250, D: 0.2281, G:1.907



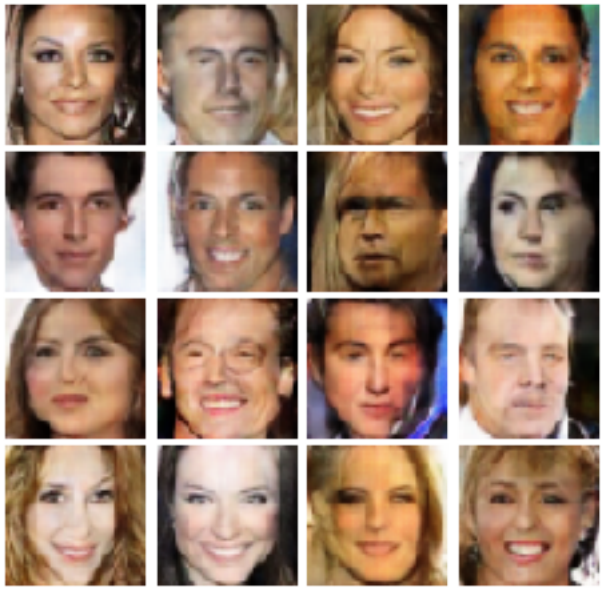
```
Iter: 28500, D: 0.1668, G:2.927
```



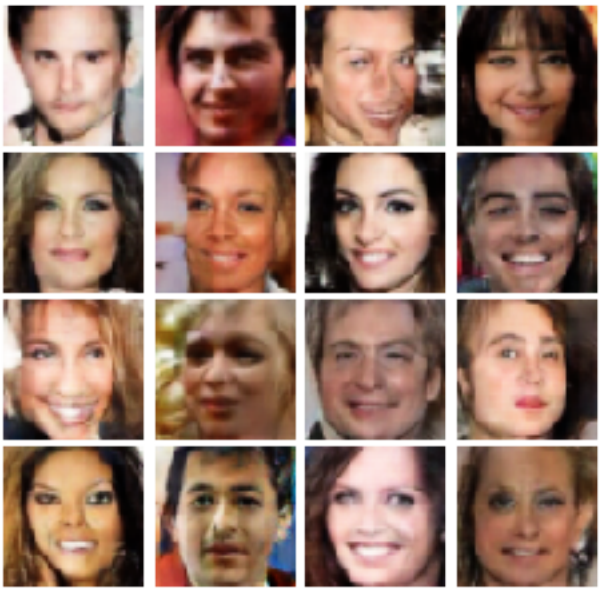
```
Saving the model as a checkpoint...
EPOCH:  30
Iter: 28750, D: 0.2712, G:2.028
```



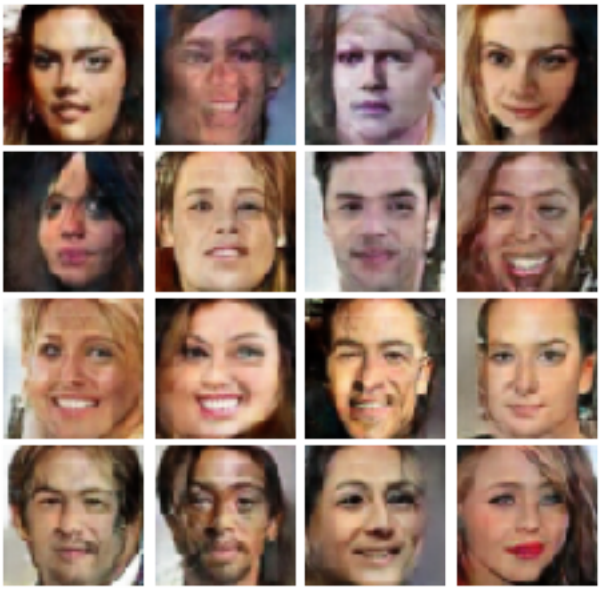
Iter: 29000, D: 0.08962, G:3.355



Iter: 29250, D: 0.5942, G:1.3



```
Iter: 29500, D: 0.08878, G:3.395
```



```
Saving the model as a checkpoint...
```

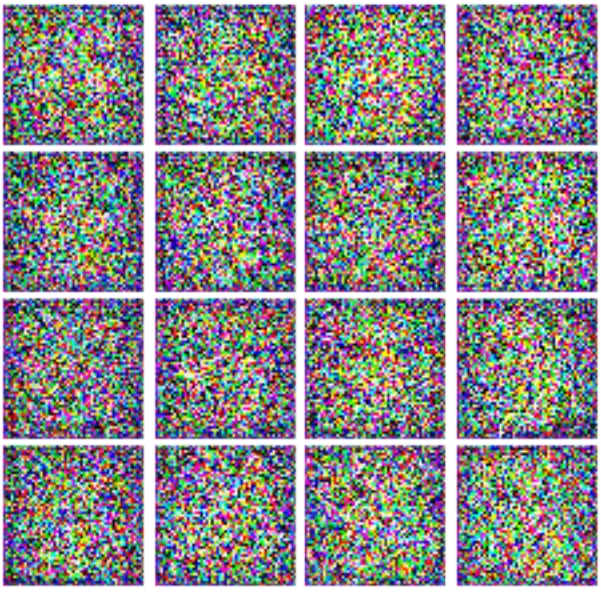
4.0.4 Section 2.4 Train a LSGAN on CelebA [13 pts]

- Call discriminator and generator for training.
- Call optimizers for both discriminator and generator for training. (Use Adam with betas = (0.5, 0.999))
- Call train function to train.
- Train for 30 epochs.

Now, train your GAN model with LSGAN loss. **Observe the visualized result of your model, and describe what you see.**

[ ]:
```
# LSGAN
# Add code here:
Discriminator = Discriminator().to(device)
Generator = Generator(noise_dim=NOISE_DIM).to(device)

Discriminator_optimizer = torch.optim.Adam(Discriminator.parameters(),␣
↪lr=learning_rate, betas = (0.5, 0.999))
Generator_optimizer = torch.optim.Adam(Generator.parameters(), lr=learning_rate,␣
↪betas = (0.5, 0.999))

train(Discriminator, Generator, Discriminator_optimizer, Generator_optimizer,␣
↪ls_discriminator_loss, ls_generator_loss, num_epochs=NUM_EPOCHS,␣
↪train_loader=celeba_loader_train, device=device)
```

```
EPOCH:  1
Iter: 0, D: 0.6173, G:0.8
```

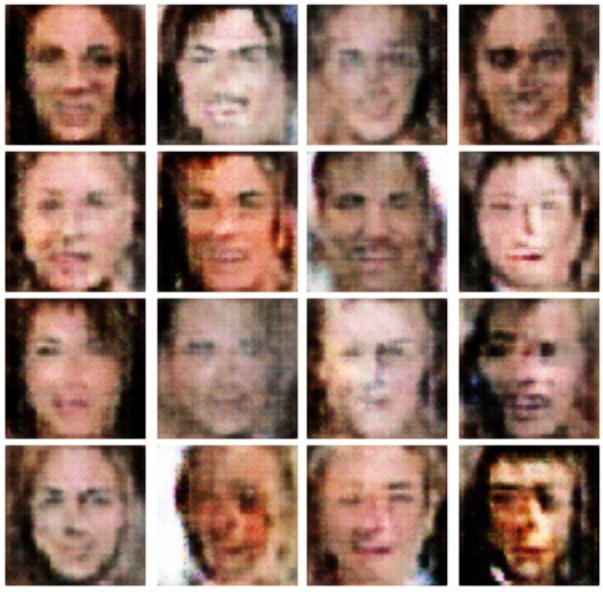


```
Iter: 200, D: 0.1789, G:0.6515
```

```
Iter: 400, D: 0.3168, G:0.3989
```

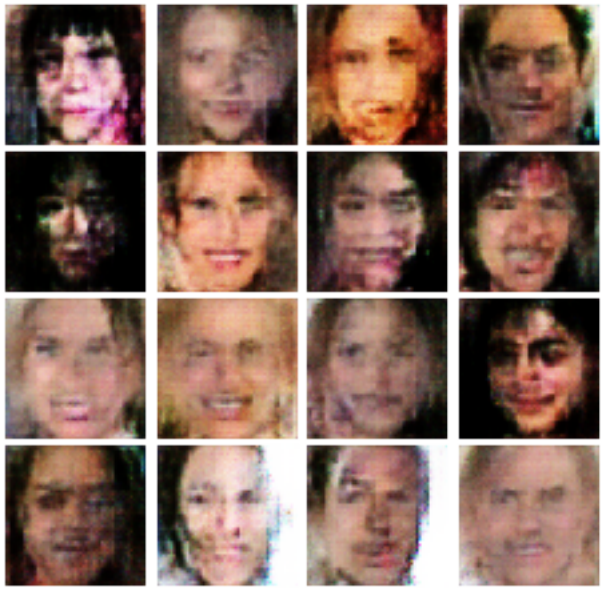
```
Iter: 600, D: 0.2022, G:0.2309
```

```
Iter: 800, D: 0.3394, G:0.3744
```

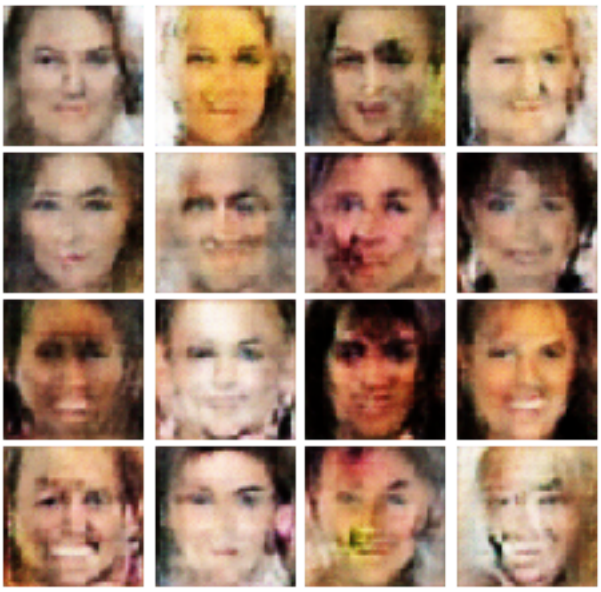


```
Saving the model as a checkpoint...
EPOCH:  2
Iter: 1000, D: 0.2386, G:0.332
```

```
Iter: 1200, D: 0.3897, G:0.2698
```

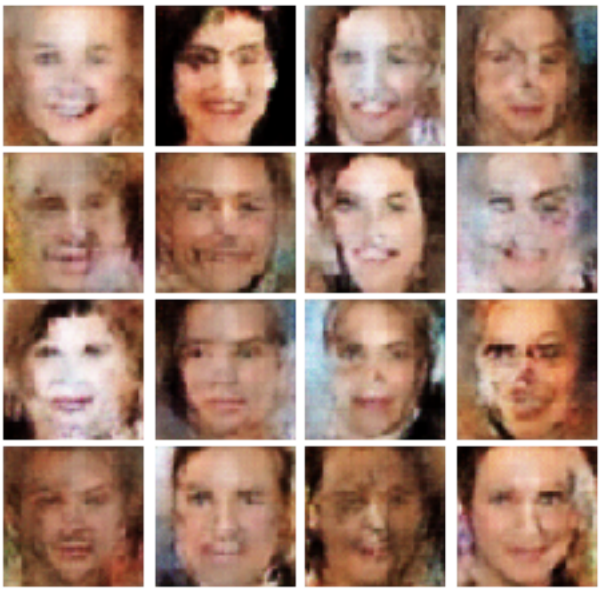


```
Iter: 1400, D: 0.17, G:0.3141
```



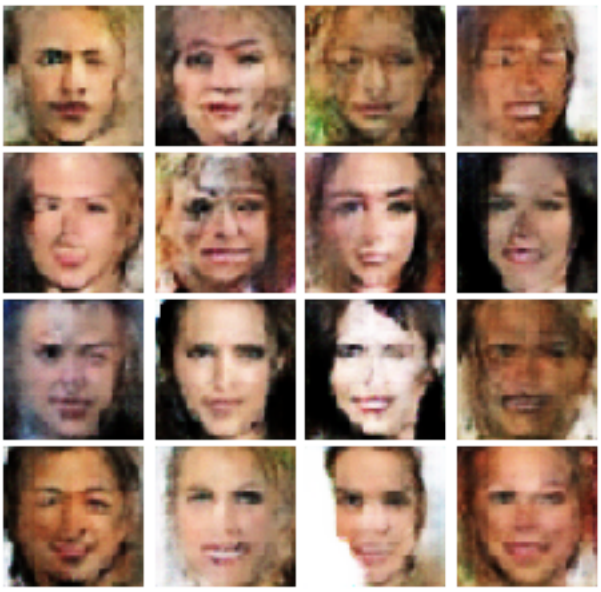
```
Iter: 1600, D: 0.2985, G:0.2513
```

```
Iter: 1800, D: 0.2072, G:0.1496
```

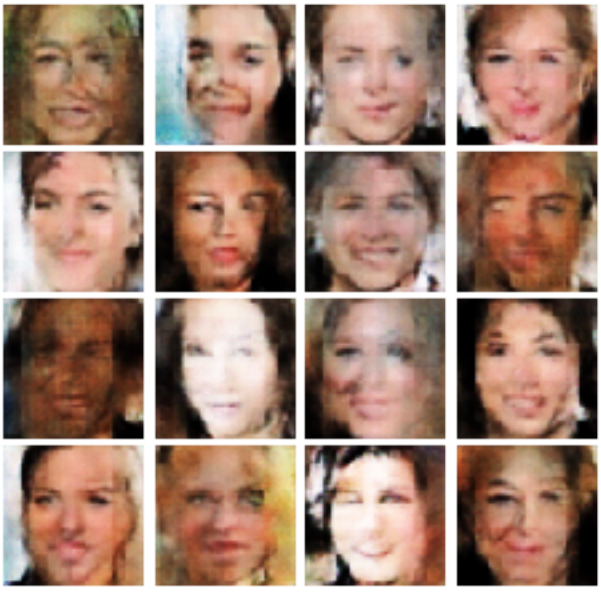


```
Saving the model as a checkpoint...
EPOCH:  3
Iter: 2000, D: 0.2835, G:0.3917
```

```
Iter: 2200, D: 0.2569, G:0.2652
```

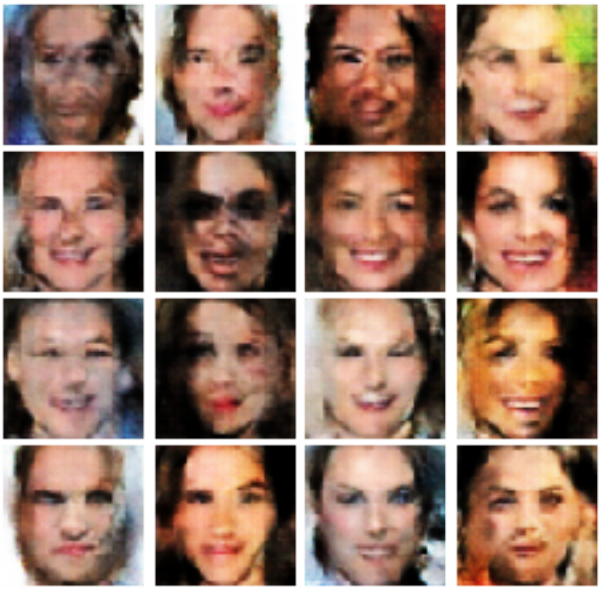


```
Iter: 2400, D: 0.1718, G:0.4543
```

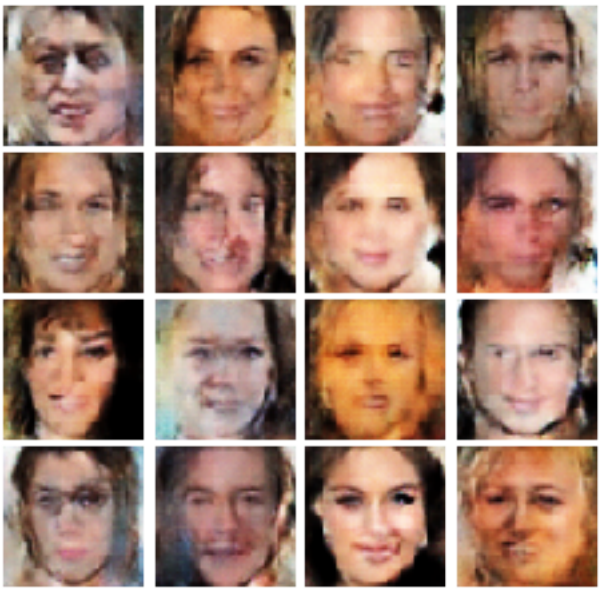


```
Iter: 2600, D: 0.297, G:0.4744
```

```
Iter: 2800, D: 0.2909, G:0.3571
```

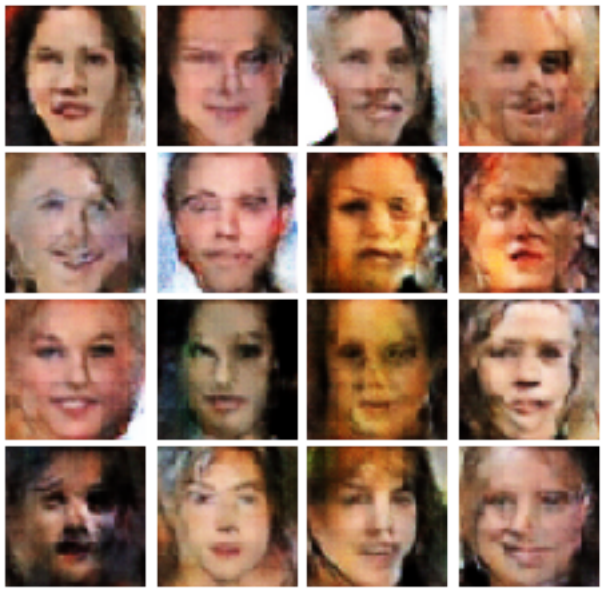


```
Saving the model as a checkpoint...
EPOCH:  4
Iter: 3000, D: 0.6384, G:0.6455
```

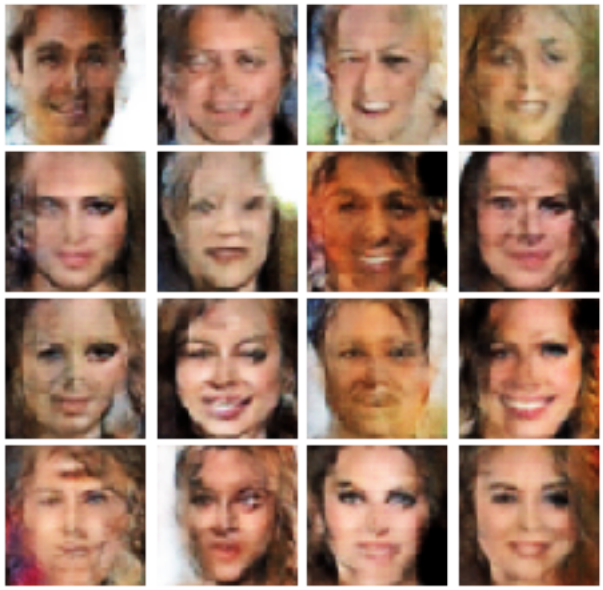


```
Iter: 3200, D: 0.8428, G:0.6925
```

```
Iter: 3400, D: 0.2726, G:0.3374
```



```
Iter: 3600, D: 0.1845, G:0.3391
```

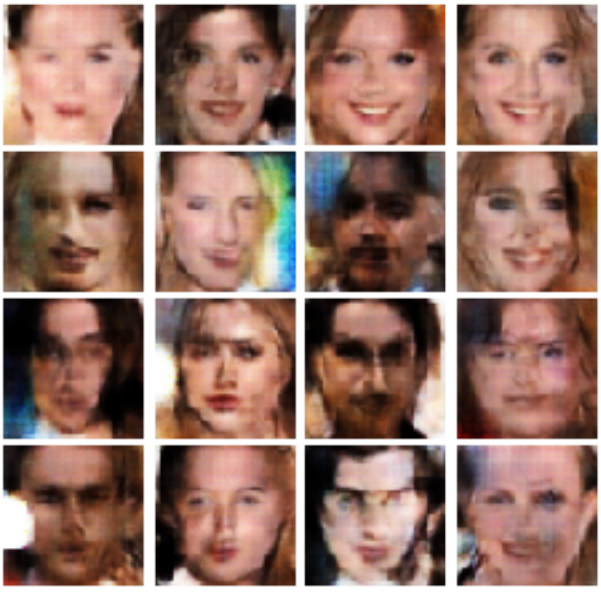


```
Iter: 3800, D: 0.2282, G:0.3251
```

```
Saving the model as a checkpoint...
EPOCH:  5
Iter: 4000, D: 0.2777, G:0.2091
```

```
Iter: 4200, D: 0.2756, G:0.1804
```

```
Iter: 4400, D: 0.4627, G:0.2294
```

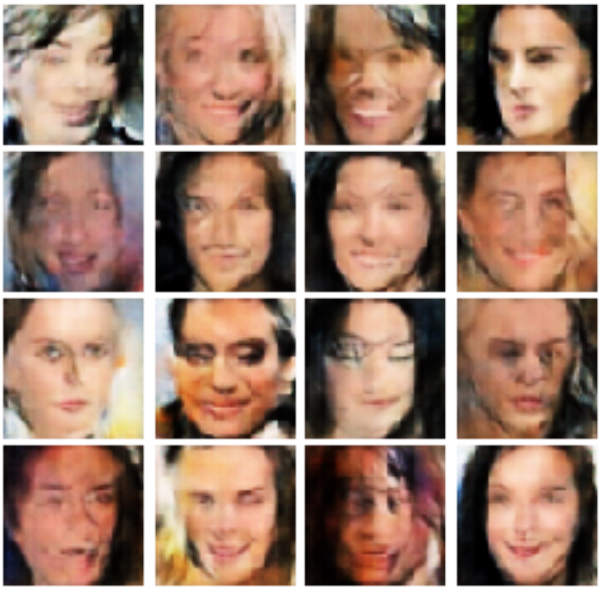


```
Iter: 4600, D: 0.3075, G:0.3127
```

```
Iter: 4800, D: 0.209, G:0.2595
```

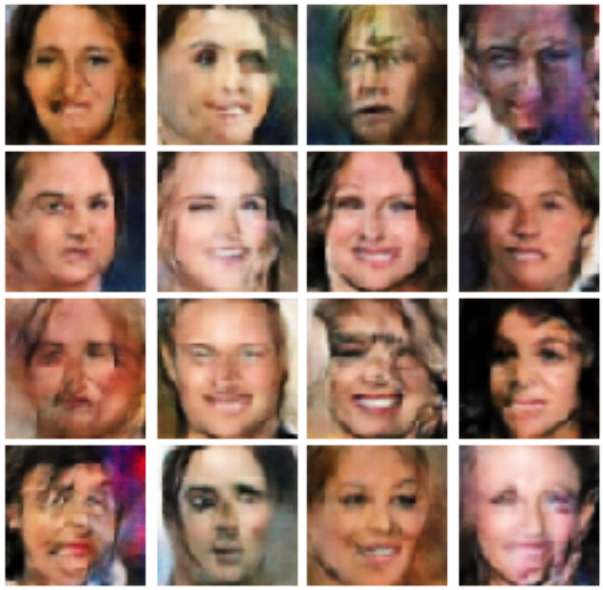
```
Saving the model as a checkpoint...
EPOCH:  6
Iter: 5000, D: 0.2472, G:0.3804
```

```
Iter: 5200, D: 0.3141, G:0.2208
```



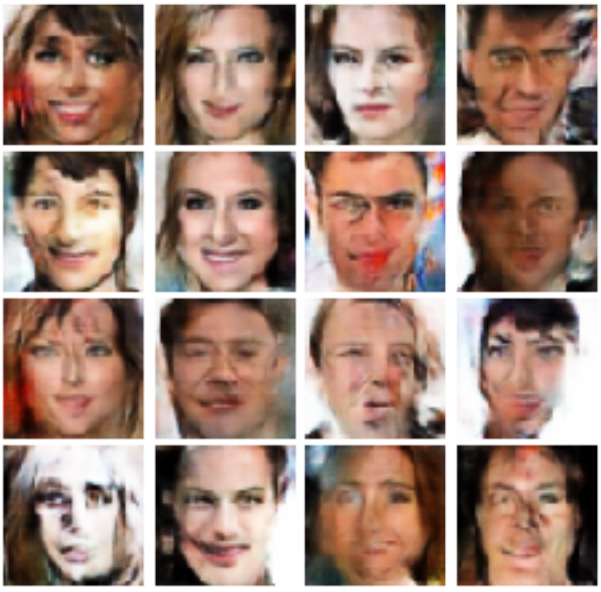
```
Iter: 5400, D: 0.2378, G:0.5414
```

Iter: 5600, D: 0.2329, G:0.3406

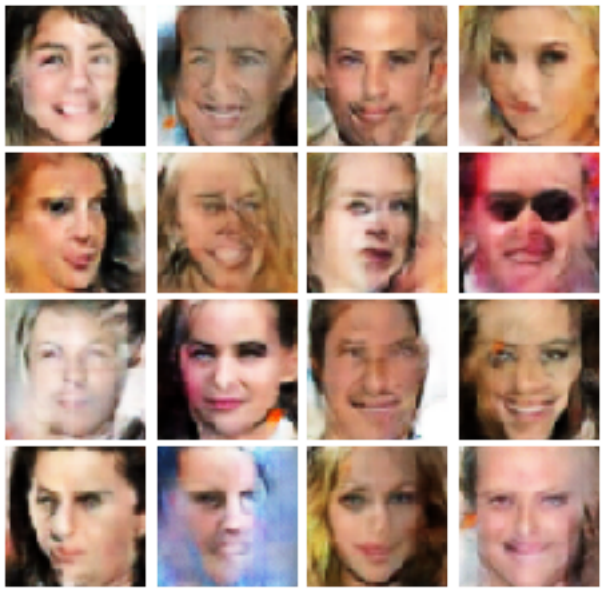


Iter: 5800, D: 0.2715, G:0.1634

```
Saving the model as a checkpoint...
EPOCH:  7
Iter: 6000, D: 0.2105, G:0.2738
```

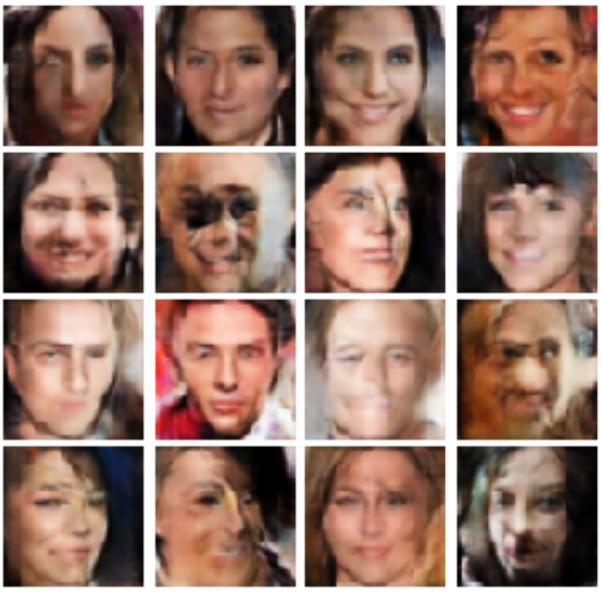


```
Iter: 6200, D: 0.2062, G:0.3741
```

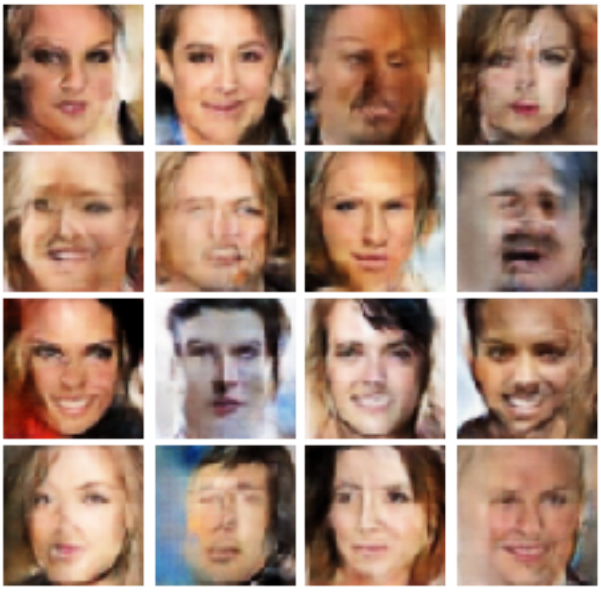


Iter: 6400, D: 0.1721, G:0.2557

Iter: 6600, D: 0.3084, G:0.389

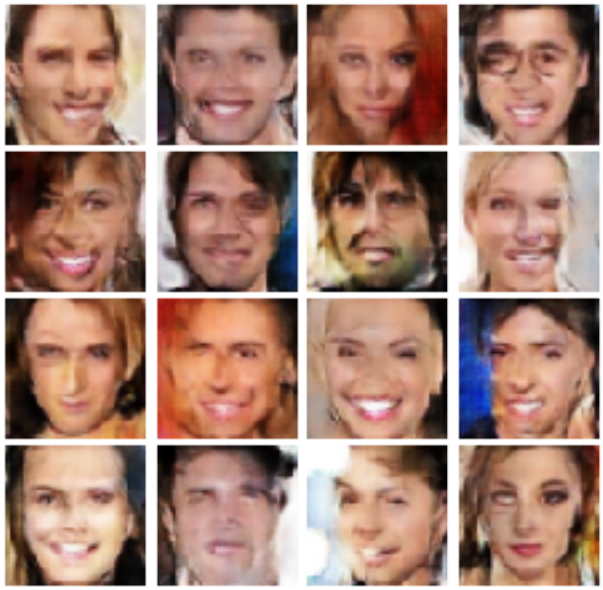


```
Iter: 6800, D: 0.1698, G:0.4296
```



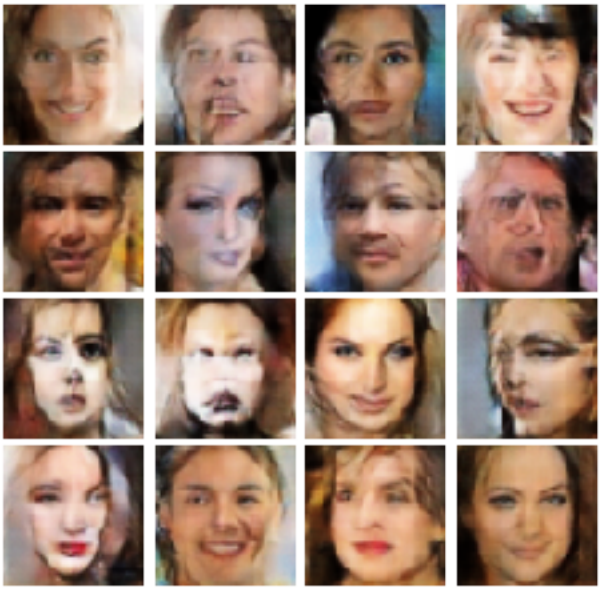
```
Saving the model as a checkpoint...
EPOCH:  8
Iter: 7000, D: 0.1234, G:0.3987
```

```
Iter: 7200, D: 0.24, G:0.2989
```



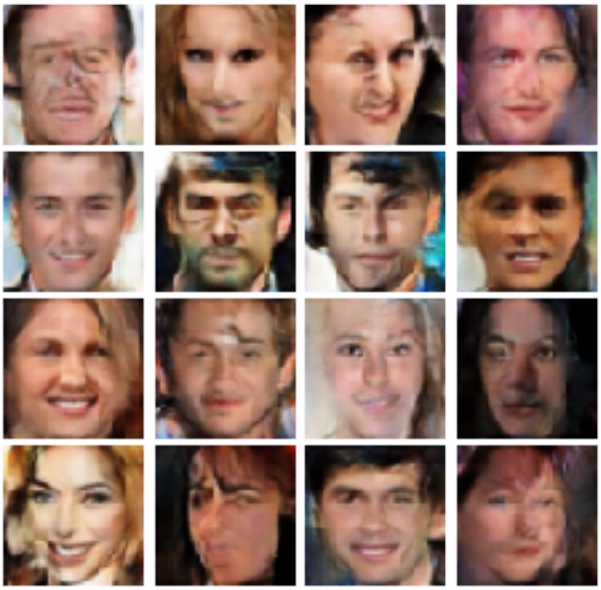
```
Iter: 7400, D: 0.2233, G:0.3457
```

```
Iter: 7600, D: 0.1863, G:0.268
```

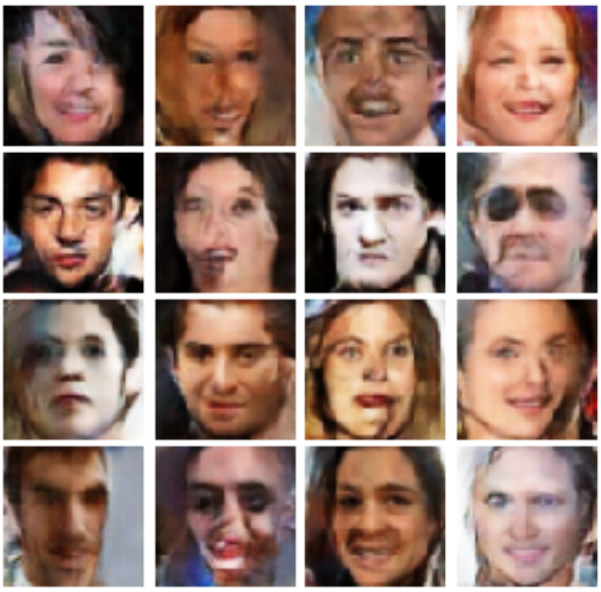


```
Iter: 7800, D: 0.1122, G:0.4481
```

```
Saving the model as a checkpoint...
EPOCH:  9
Iter: 8000, D: 0.2978, G:0.362
```

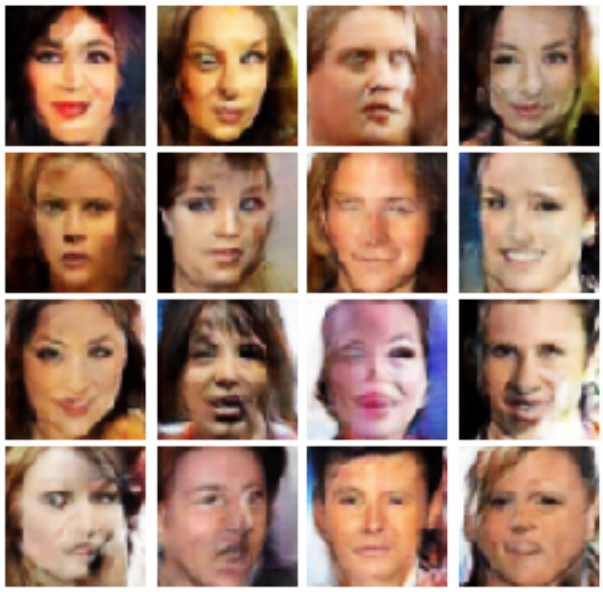


```
Iter: 8200, D: 0.176, G:0.4461
```



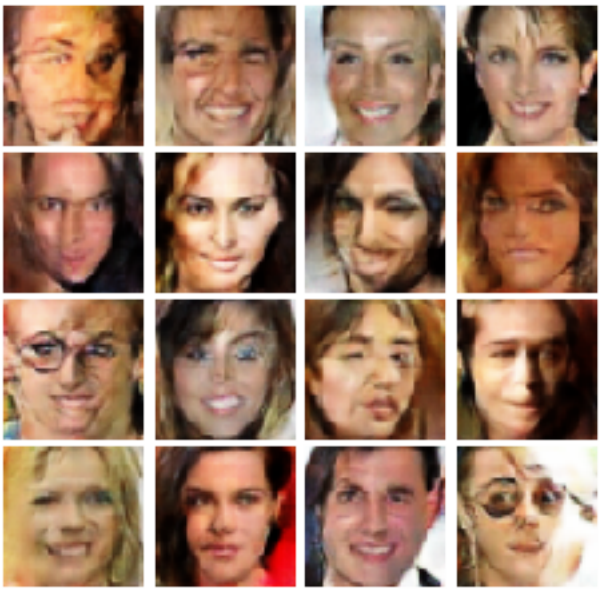
Iter: 8400, D: 0.2866, G:0.3153

Iter: 8600, D: 0.1465, G:0.481



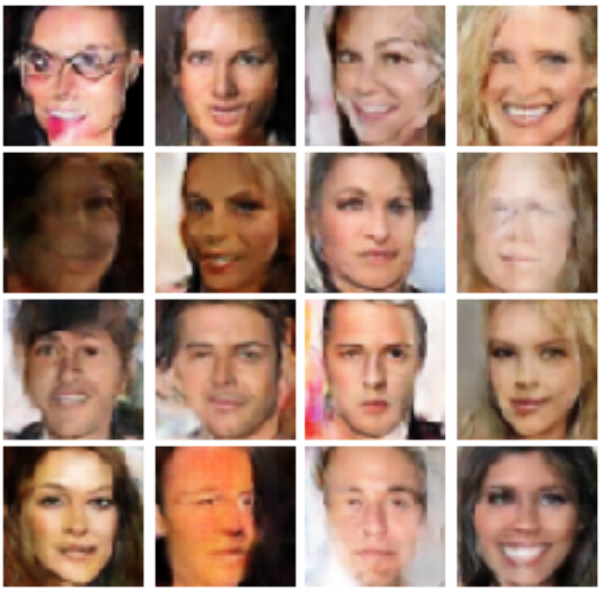
```
Iter: 8800, D: 0.1176, G:0.5146
```

```
Saving the model as a checkpoint...
EPOCH:  10
Iter: 9000, D: 0.2112, G:0.5374
```



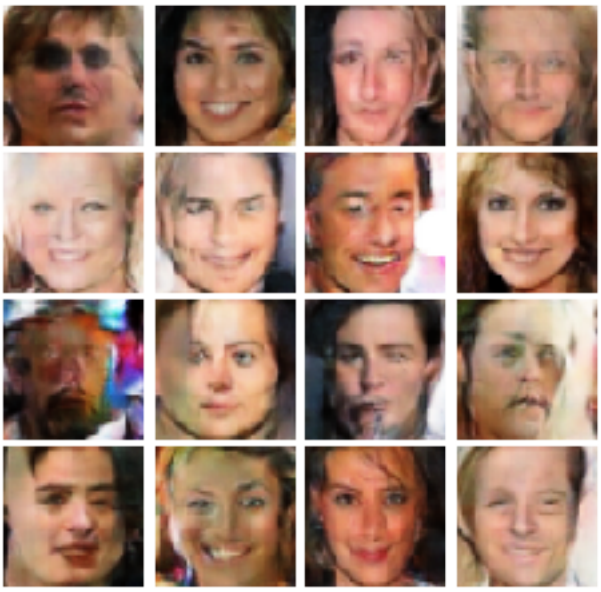
Iter: 9200, D: 0.1241, G:0.3211

Iter: 9400, D: 0.1038, G:0.494



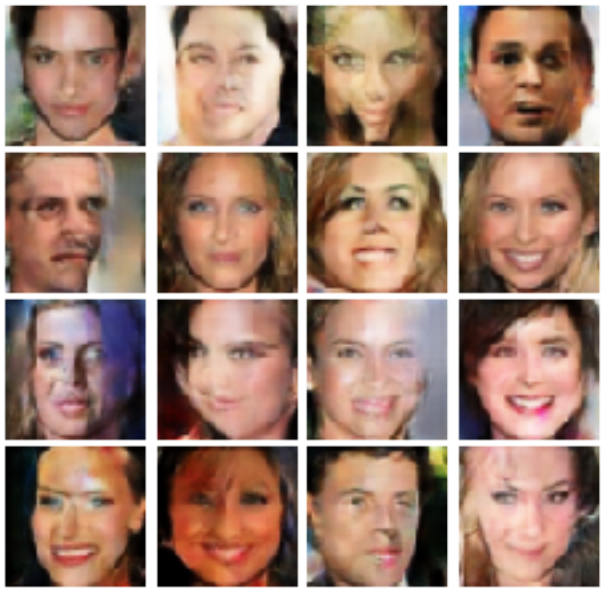
Iter: 9600, D: 0.2842, G:0.6619

Iter: 9800, D: 0.1257, G:0.3256



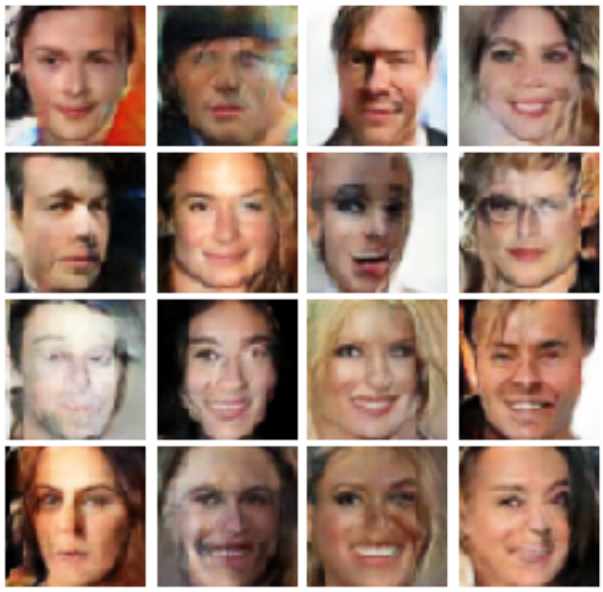
```
Saving the model as a checkpoint...
EPOCH:  11
Iter: 10000, D: 0.2099, G:0.2879
```

```
Iter: 10200, D: 0.2098, G:0.5236
```

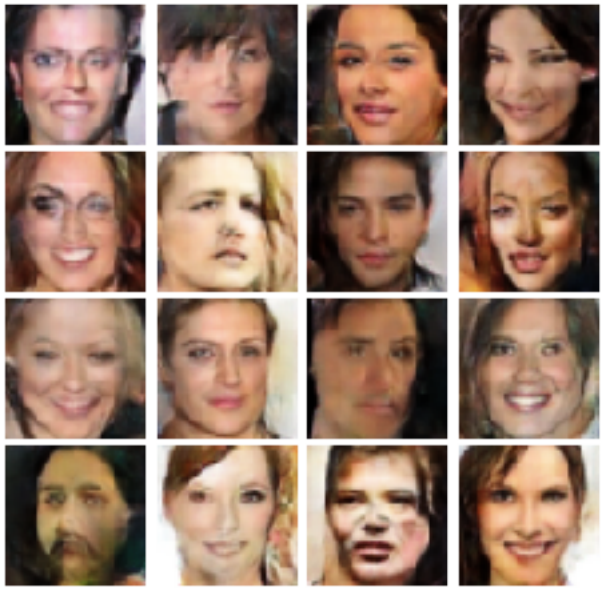


Iter: 10400, D: 0.1164, G:0.259

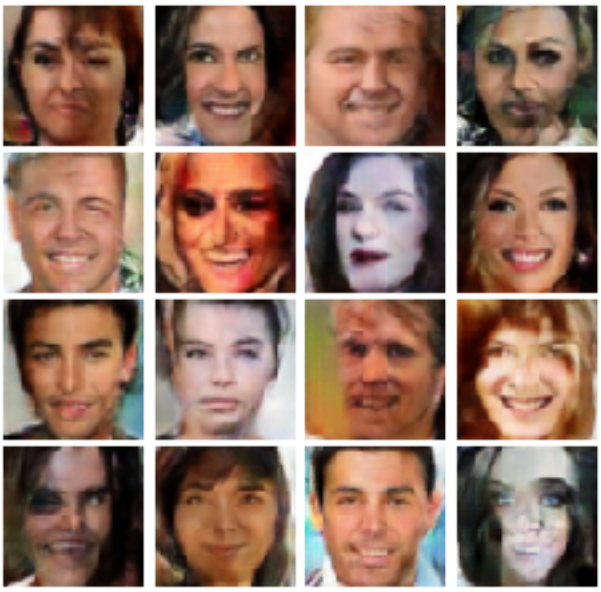
Iter: 10600, D: 0.1292, G:0.4572



```
Iter: 10800, D: 0.07276, G:0.4648
```

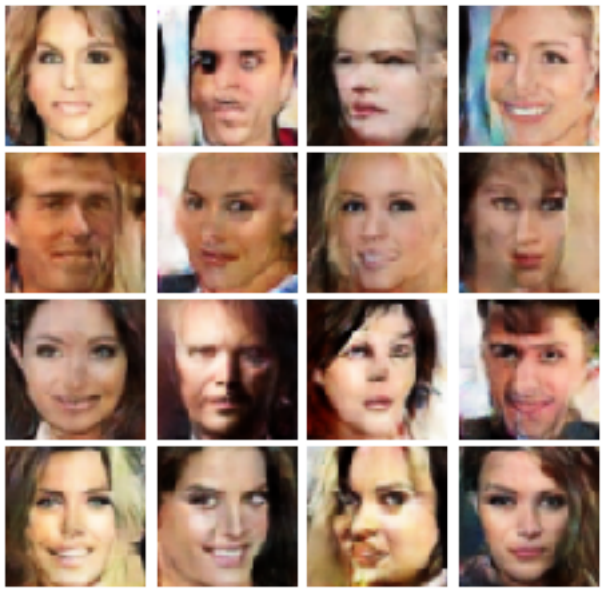


```
Saving the model as a checkpoint...
EPOCH:  12
Iter: 11000, D: 0.09715, G:0.4284
```



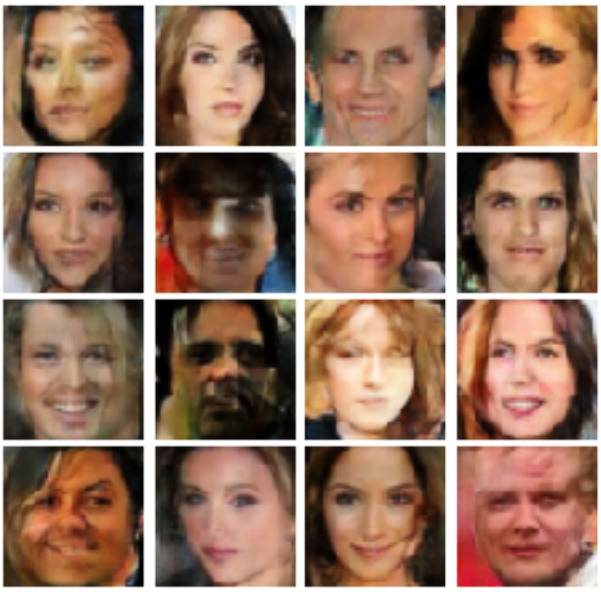
```
Iter: 11200, D: 0.0942, G:0.4098
```

```
Iter: 11400, D: 0.09395, G:0.5904
```



```
Iter: 11600, D: 0.0373, G:0.2524
```

```
Iter: 11800, D: 0.1111, G:0.3459
```

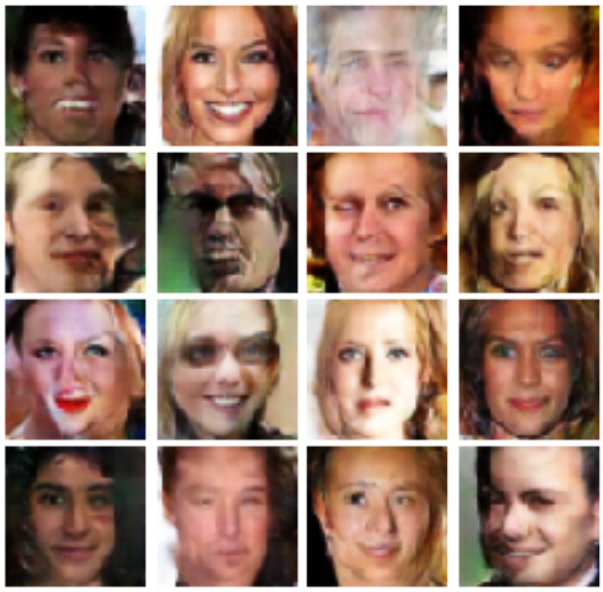


```
Saving the model as a checkpoint...
EPOCH:  13
Iter: 12000, D: 0.1485, G:0.5799
```

```
Iter: 12200, D: 0.2403, G:0.5882
```

```
Iter: 12400, D: 0.1716, G:0.3003
```

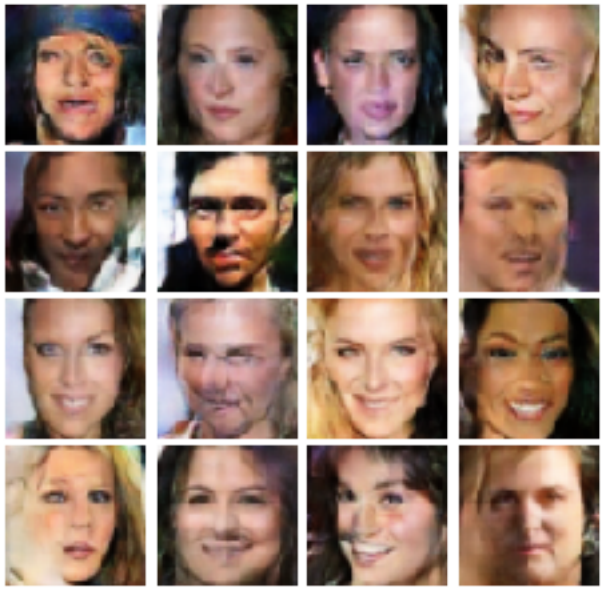
```
Iter: 12600, D: 0.07261, G:0.4543
```



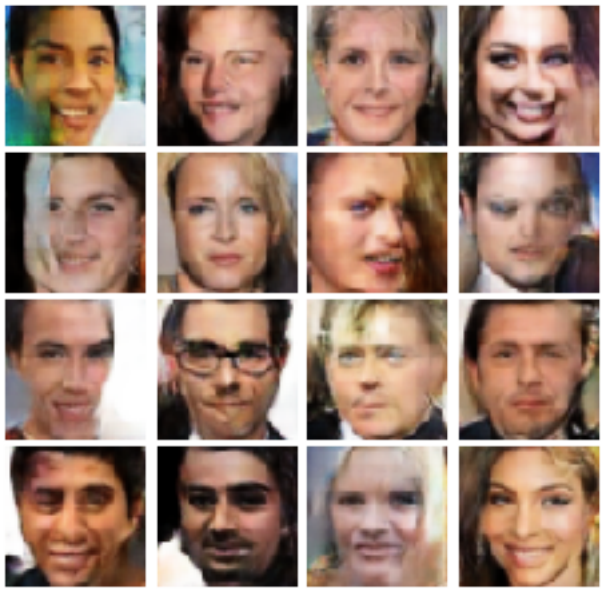
```
Iter: 12800, D: 0.1268, G:0.5717
```

```
Saving the model as a checkpoint...
EPOCH:  14
Iter: 13000, D: 0.04627, G:0.3827
```

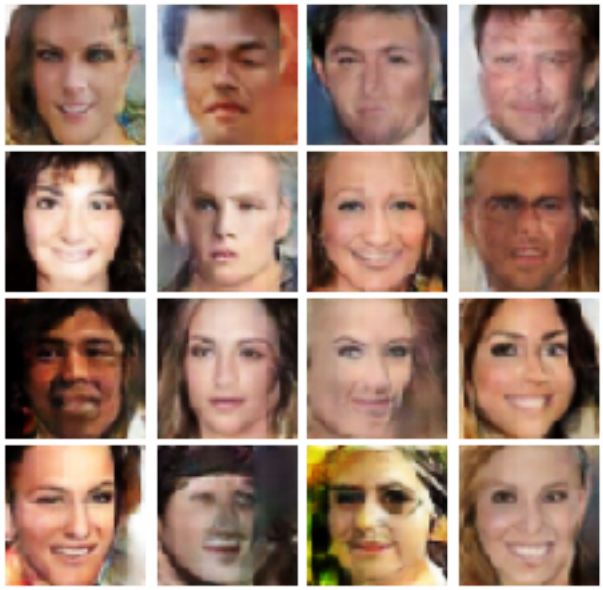
```
Iter: 13200, D: 0.08964, G:0.5204
```



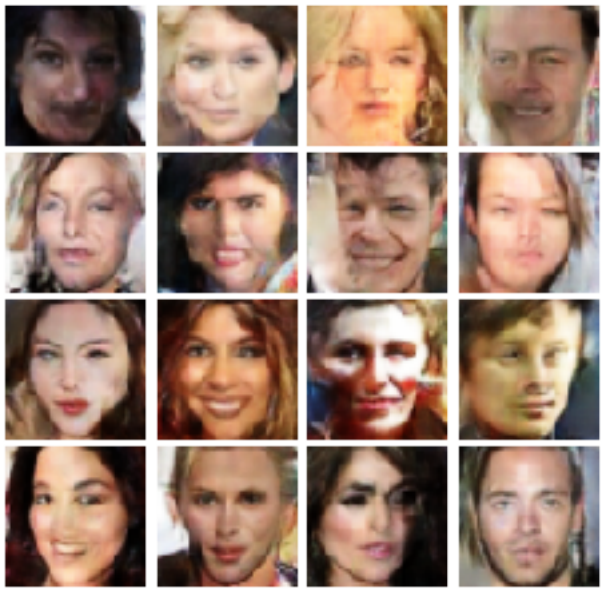
```
Iter: 13400, D: 0.05401, G:0.442
```



```
Iter: 13600, D: 0.1019, G:0.4889
```

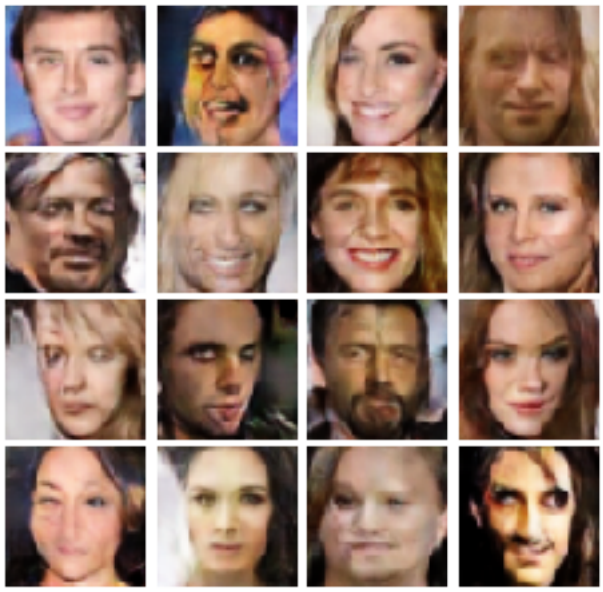


```
Iter: 13800, D: 0.05001, G:0.4037
```

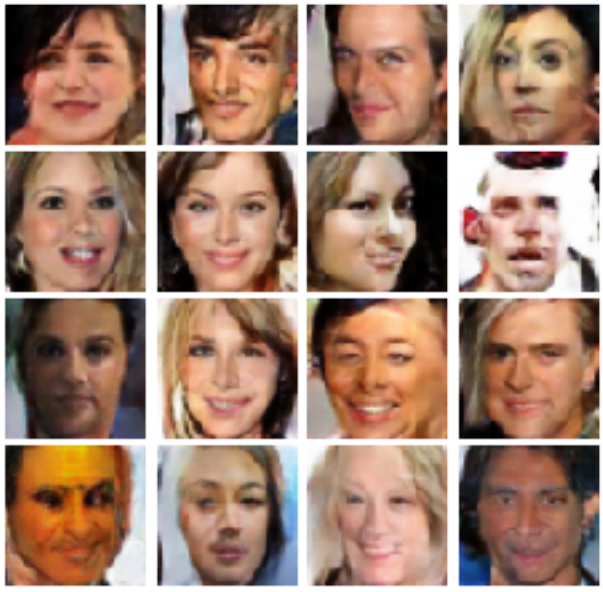


```
Saving the model as a checkpoint...
EPOCH:  15
Iter: 14000, D: 0.04998, G:0.4675
```

```
Iter: 14200, D: 0.1061, G:0.3915
```



```
Iter: 14400, D: 0.06562, G:0.5335
```

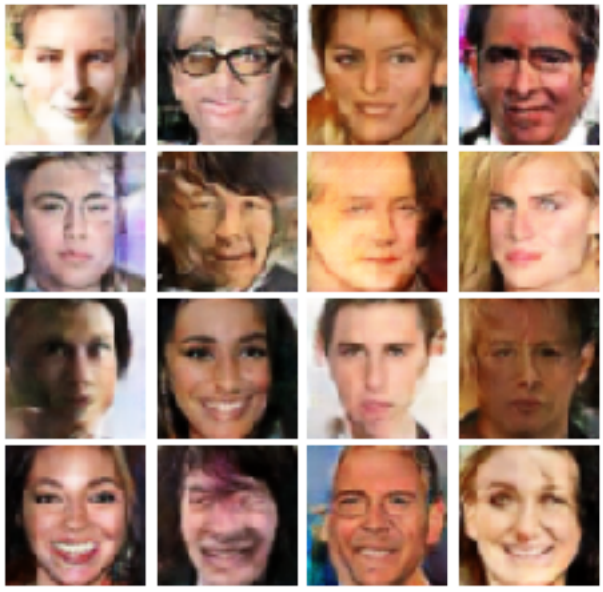


```
Iter: 14600, D: 0.08865, G:0.4253
```

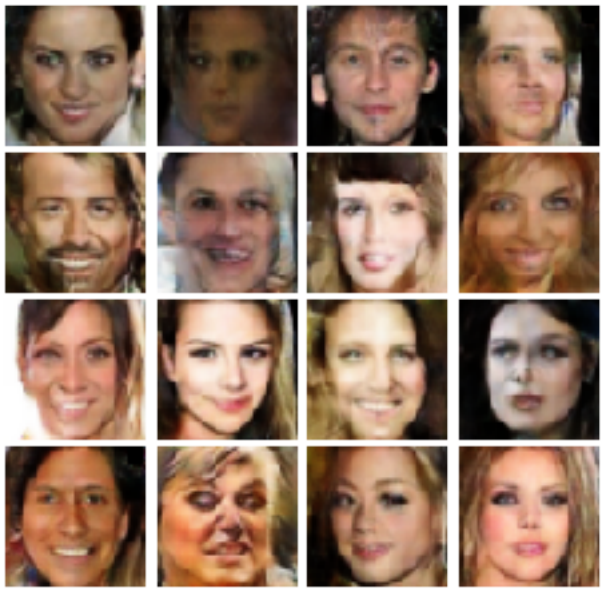
```
Saving the model as a checkpoint...
EPOCH:  16
Iter: 14800, D: 0.04332, G:0.3664
```

```
Iter: 15000, D: 0.05888, G:0.5161
```

```
Iter: 15200, D: 0.1082, G:0.4069
```

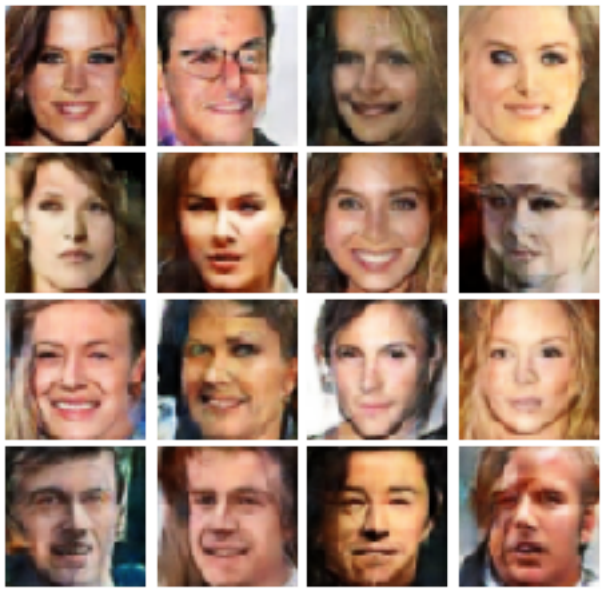


```
Iter: 15400, D: 0.04184, G:0.5134
```



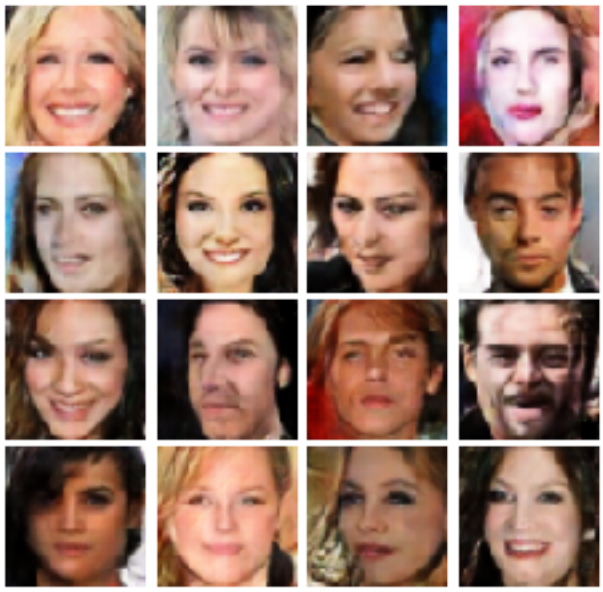
```
Iter: 15600, D: 0.07799, G:0.4987
```

```
Saving the model as a checkpoint...
EPOCH:  17
Iter: 15800, D: 0.03528, G:0.5378
```

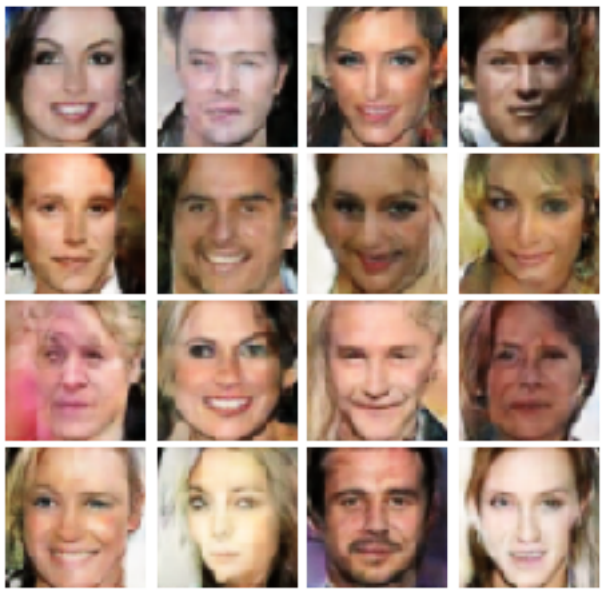


```
Iter: 16000, D: 0.06807, G:0.2614
```

```
Iter: 16200, D: 0.04126, G:0.5134
```

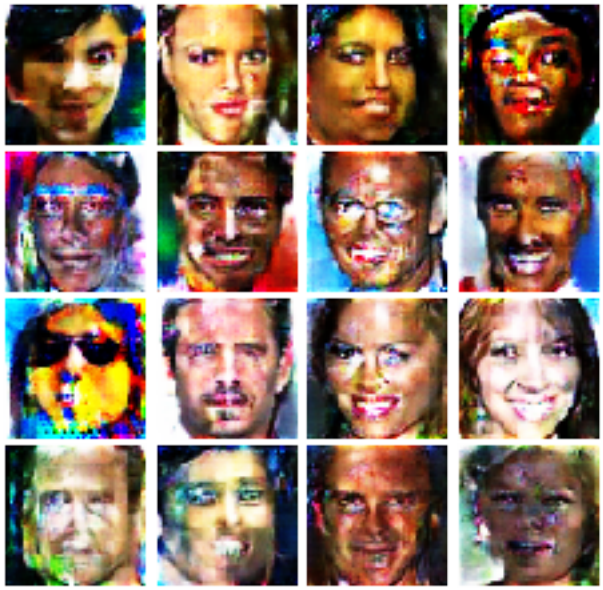


Iter: 16400, D: 0.0379, G:0.3274



Iter: 16600, D: 0.04021, G:0.3096

```
Saving the model as a checkpoint...
EPOCH:  18
Iter: 16800, D: 0.106, G:0.6336
```



```
Iter: 17000, D: 0.03643, G:0.4047
```

```
Iter: 17200, D: 0.05067, G:0.4933
```

```
Iter: 17400, D: 0.03425, G:0.4734
```

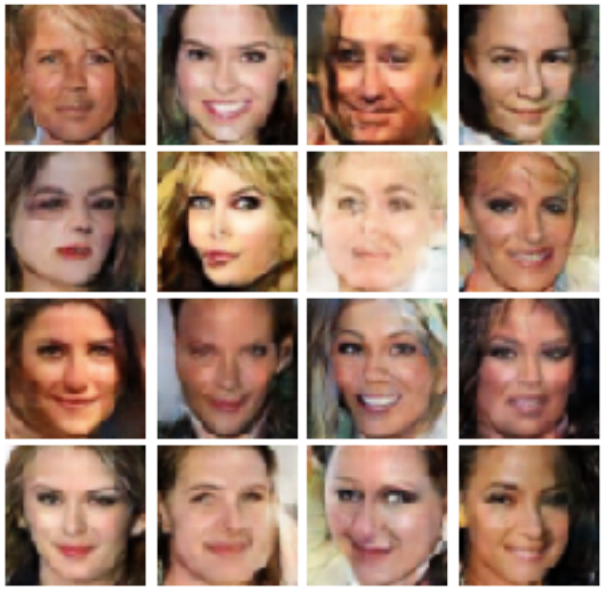
```
Iter: 17600, D: 0.09341, G:0.1841
```

```
Saving the model as a checkpoint...
EPOCH:  19
Iter: 17800, D: 0.1213, G:0.2111
```

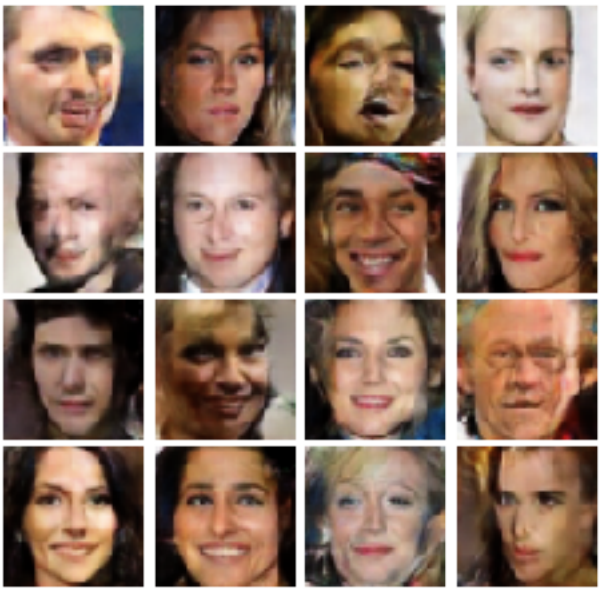
Iter: 18000, D: 0.07009, G:0.4818

Iter: 18200, D: 0.05736, G:0.2766

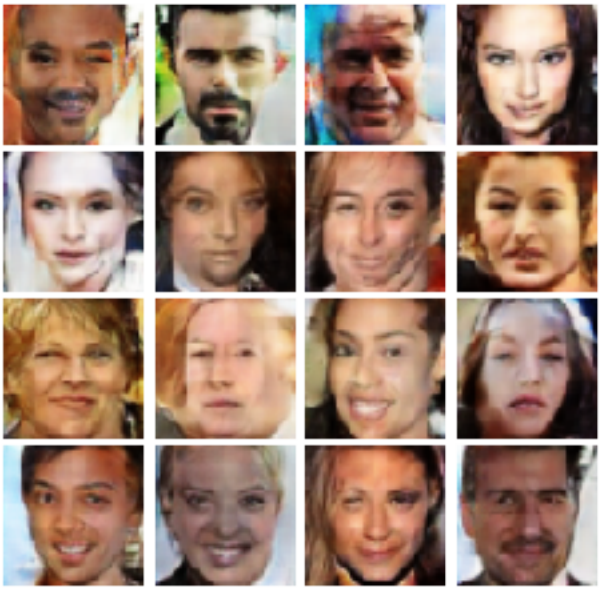
```
Iter: 18400, D: 0.1009, G:0.4774
```



```
Iter: 18600, D: 0.03963, G:0.2947
```



```
Saving the model as a checkpoint...
EPOCH:  20
Iter: 18800, D: 0.0744, G:0.4753
```



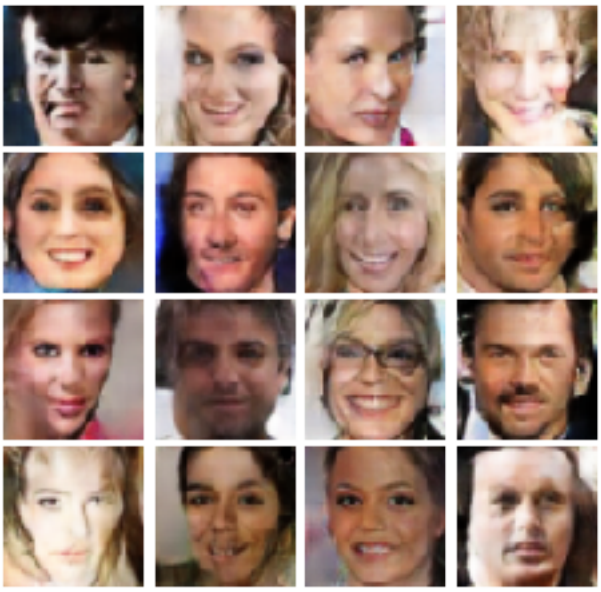
```
Iter: 19000, D: 0.04323, G:0.3754
```

Iter: 19200, D: 0.1576, G:0.5583

Iter: 19400, D: 0.0613, G:0.4025

```
Iter: 19600, D: 0.02756, G:0.4819
```

```
Saving the model as a checkpoint...
EPOCH:  21
Iter: 19800, D: 0.04858, G:0.5545
```



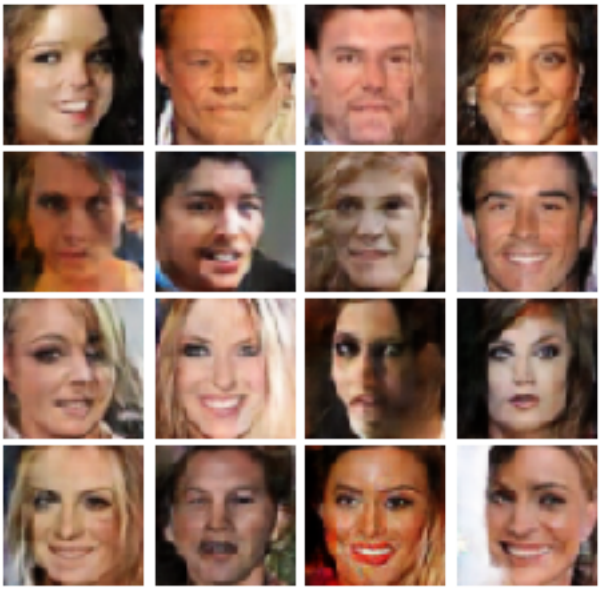
```
Iter: 20000, D: 0.05066, G:0.4018
```

```
Iter: 20200, D: 0.0214, G:0.5585
```

```
Iter: 20400, D: 0.03784, G:0.4776
```

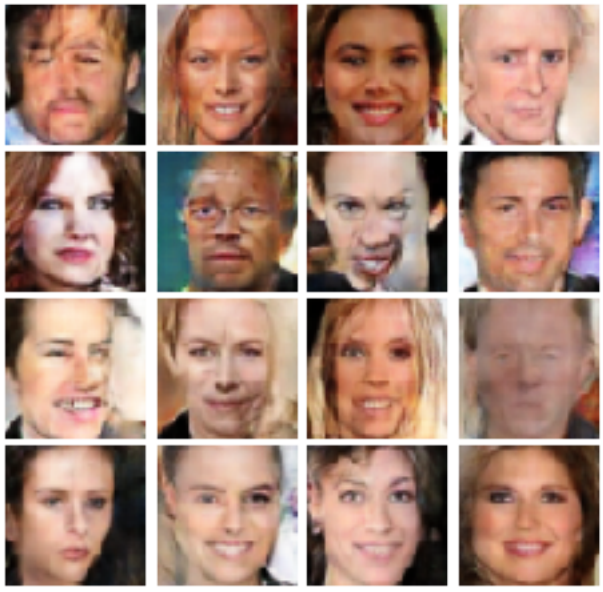
```
Iter: 20600, D: 0.06417, G:0.2686
```

```
Saving the model as a checkpoint...
EPOCH:  22
Iter: 20800, D: 0.02492, G:0.5305
```



```
Iter: 21000, D: 0.01938, G:0.4636
```

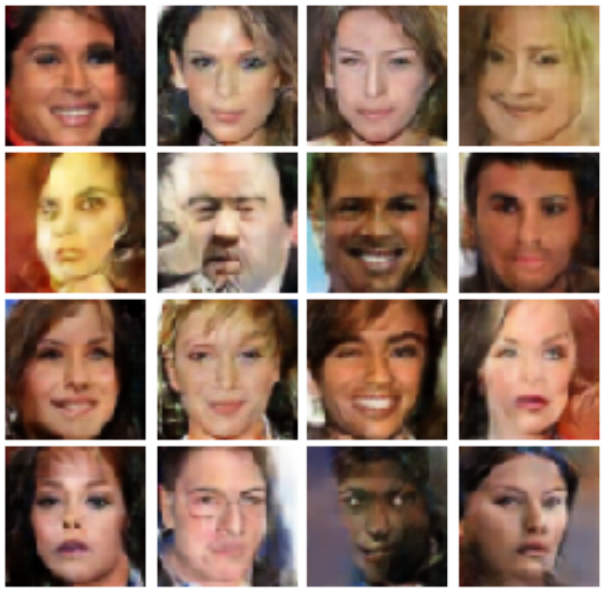
Iter: 21200, D: 0.05781, G:0.4464



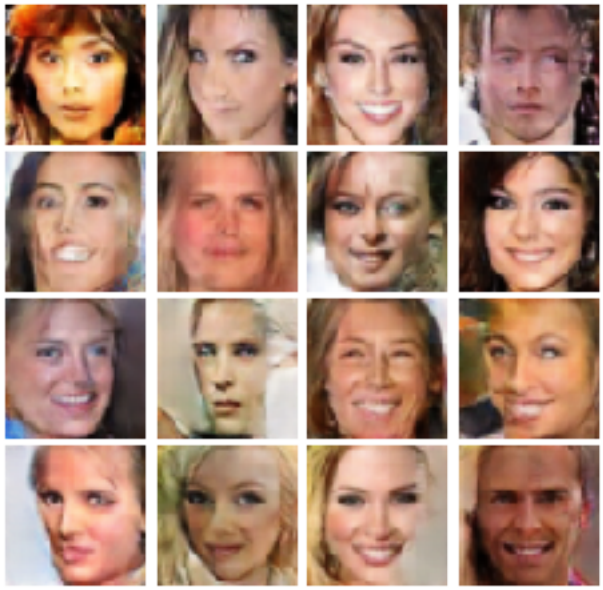
Iter: 21400, D: 0.04155, G:0.5412

```
Iter: 21600, D: 0.04313, G:0.4238
```

```
Saving the model as a checkpoint...
EPOCH:  23
Iter: 21800, D: 0.04868, G:0.5443
```



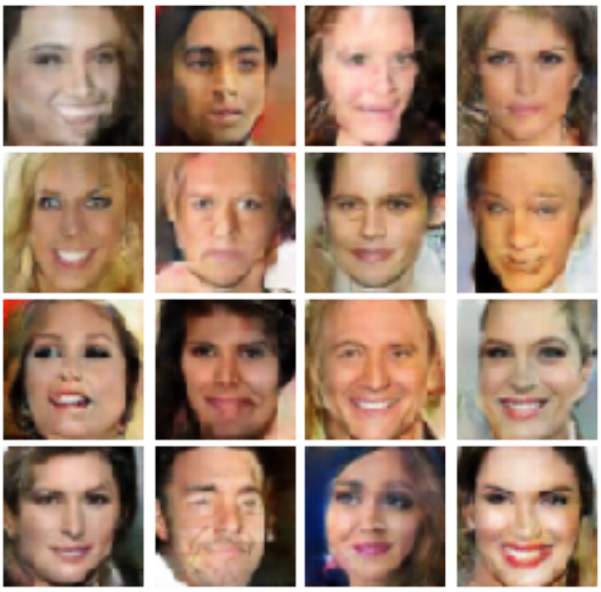
```
Iter: 22000, D: 0.02429, G:0.5061
```



```
Iter: 22200, D: 0.03327, G:0.3419
```

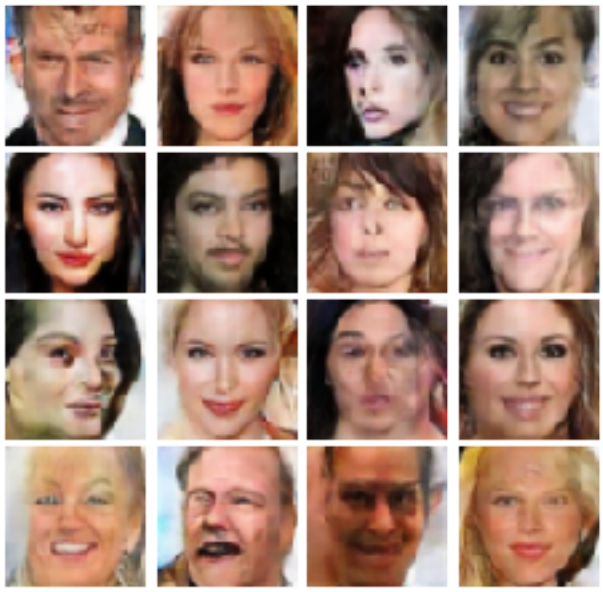
```
Iter: 22400, D: 0.04036, G:0.2092
```

```
Iter: 22600, D: 0.02953, G:0.4085
```

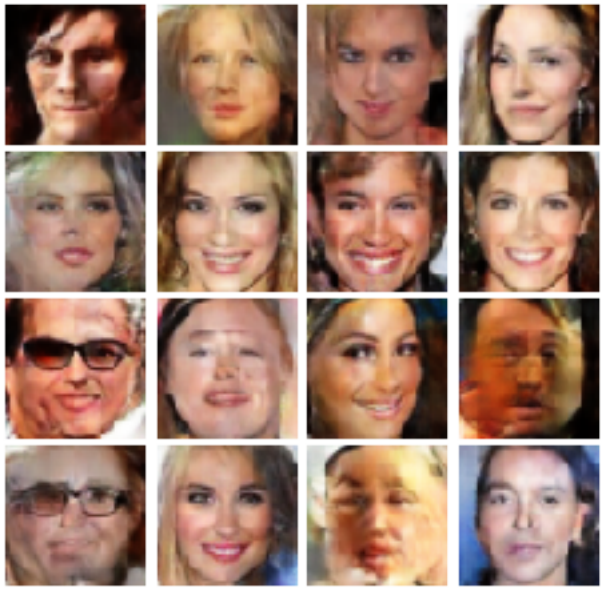


```
Saving the model as a checkpoint...
EPOCH:  24
Iter: 22800, D: 0.04062, G:0.4497
```

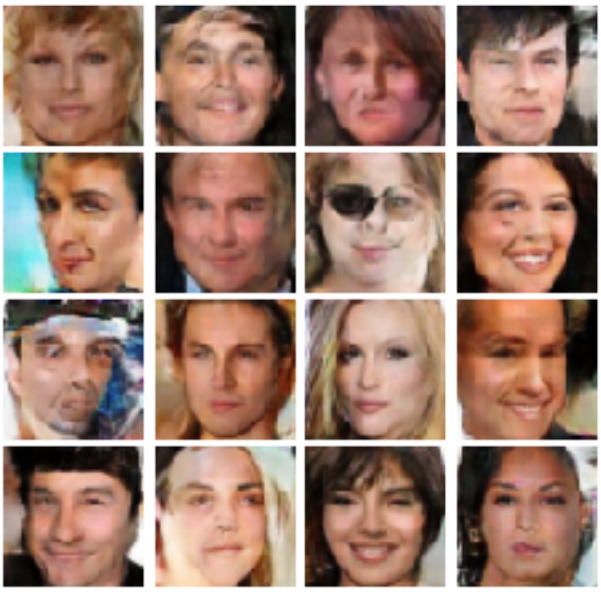
```
Iter: 23000, D: 0.06767, G:0.3397
```



Iter: 23200, D: 0.1264, G:0.5443

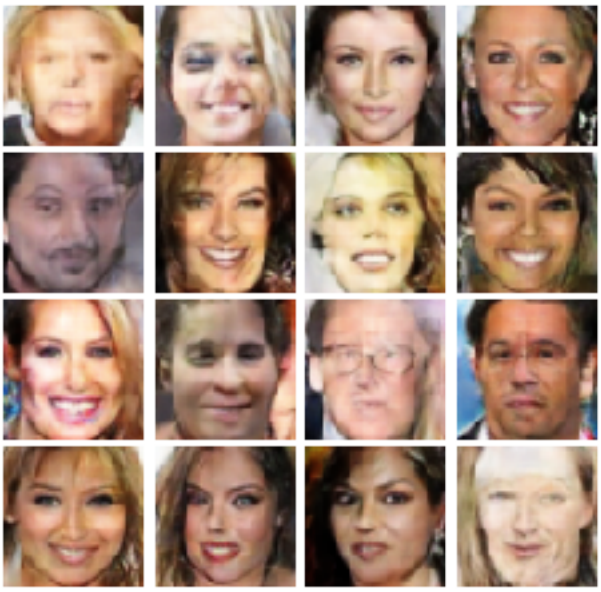


Iter: 23400, D: 0.03788, G:0.5158



```
Iter: 23600, D: 0.02618, G:0.452
```

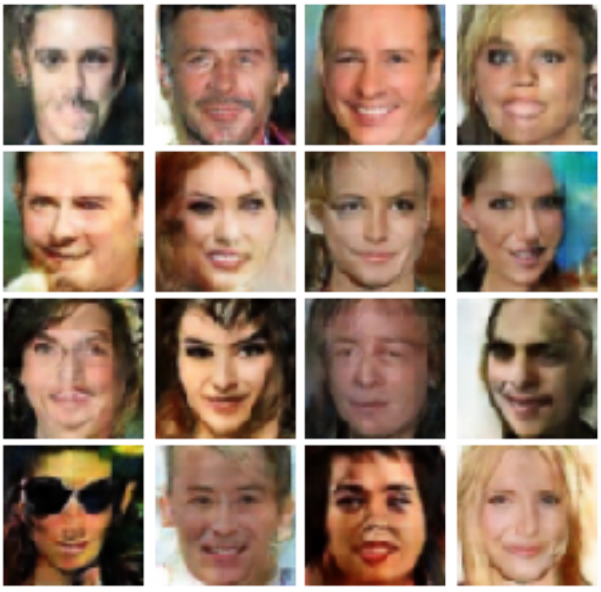
```
Saving the model as a checkpoint...
EPOCH:  25
Iter: 23800, D: 0.04964, G:0.5466
```



```
Iter: 24000, D: 0.1095, G:0.606
```

```
Iter: 24200, D: 0.02165, G:0.5196
```

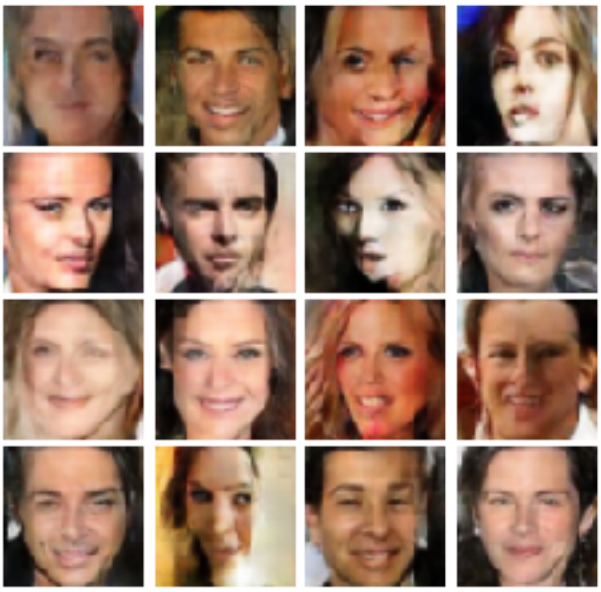
Iter: 24400, D: 0.08117, G:0.4152



Iter: 24600, D: 0.07538, G:0.3322

```
Saving the model as a checkpoint...
EPOCH:  26
Iter: 24800, D: 0.02229, G:0.4868
```

```
Iter: 25000, D: 0.03097, G:0.5348
```



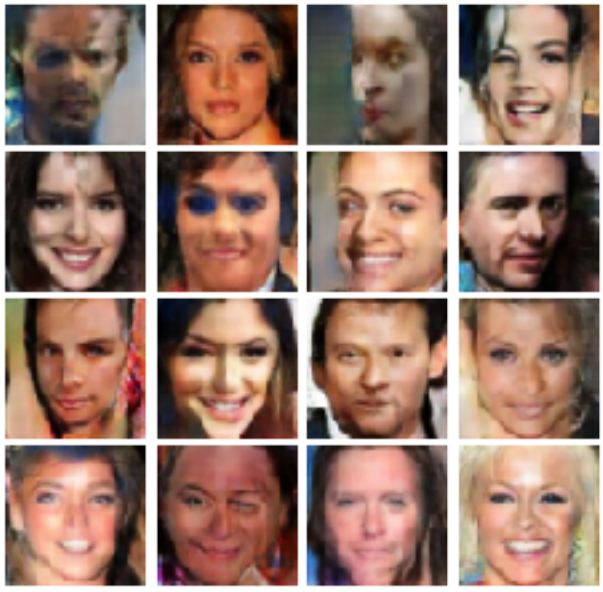
```
Iter: 25200, D: 0.09722, G:0.442
```

```
Iter: 25400, D: 0.03887, G:0.3697
```

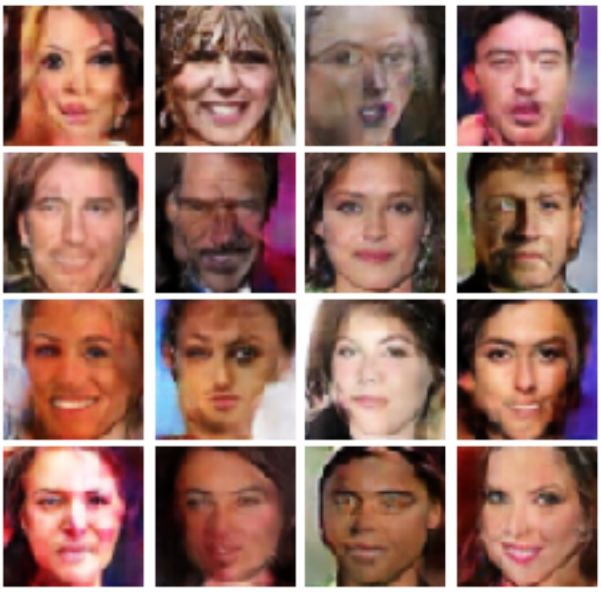
```
Iter: 25600, D: 0.04209, G:0.5733
```

```
Saving the model as a checkpoint...
EPOCH:  27
Iter: 25800, D: 0.02997, G:0.5028
```

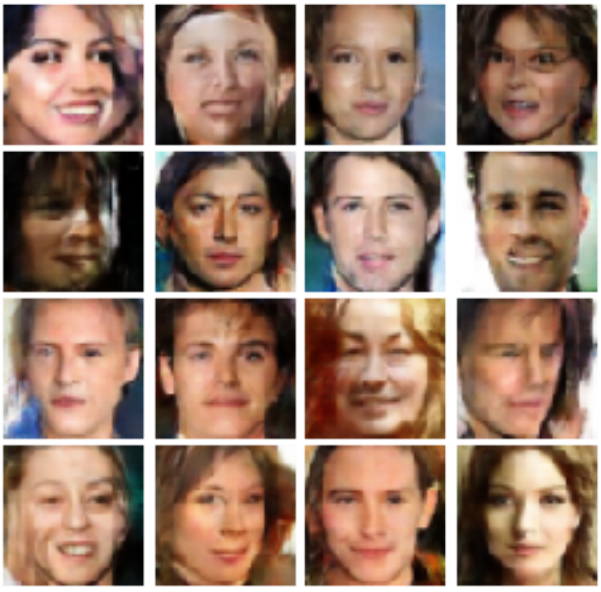
```
Iter: 26000, D: 0.0237, G:0.3771
```



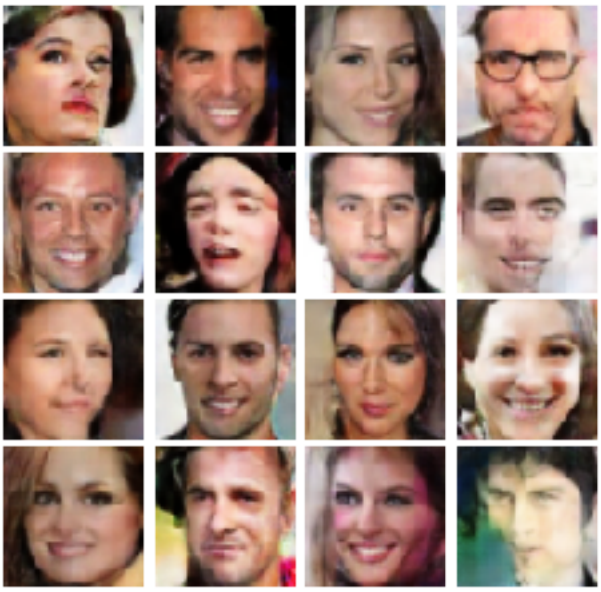
```
Iter: 26200, D: 0.138, G:0.5709
```



Iter: 26400, D: 0.03218, G:0.2654



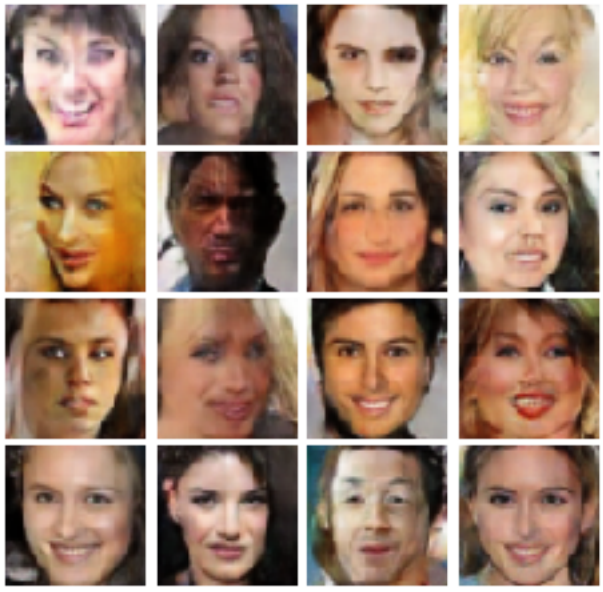
Iter: 26600, D: 0.03086, G:0.2195



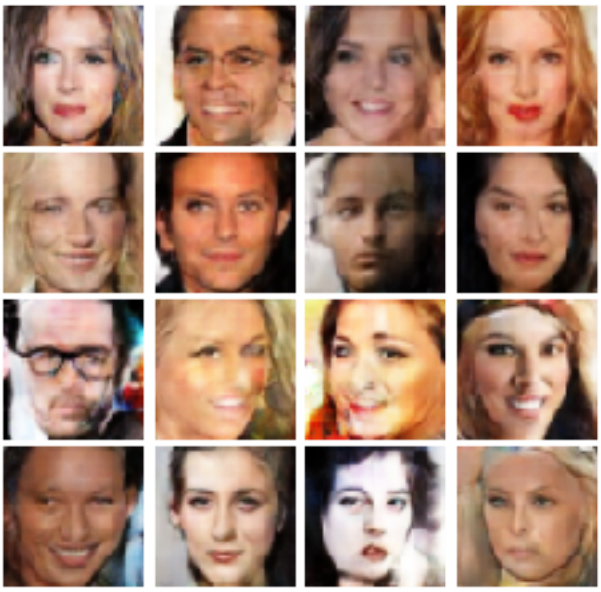
```
Saving the model as a checkpoint...
EPOCH:  28
Iter: 26800, D: 0.0234, G:0.4709
```

```
Iter: 27000, D: 0.01155, G:0.4213
```

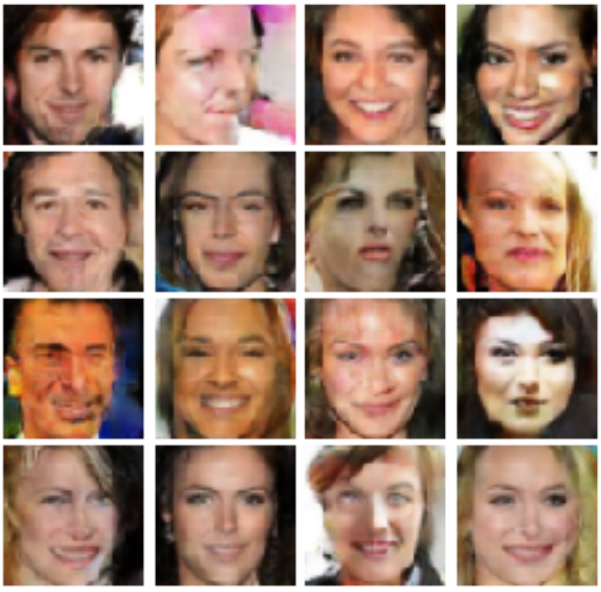
```
Iter: 27200, D: 0.01559, G:0.4572
```



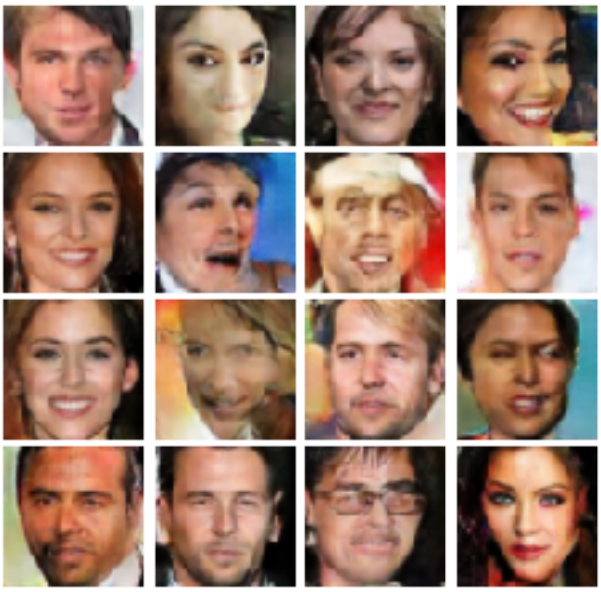
```
Iter: 27400, D: 0.02154, G:0.4623
```



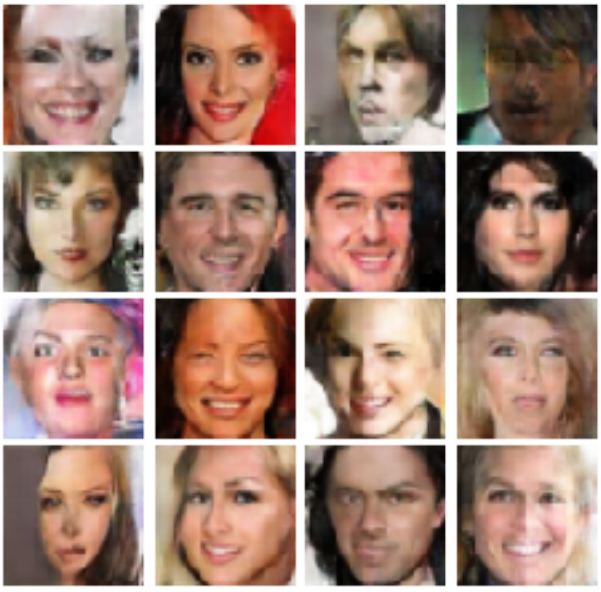
```
Iter: 27600, D: 0.01426, G:0.5482
```



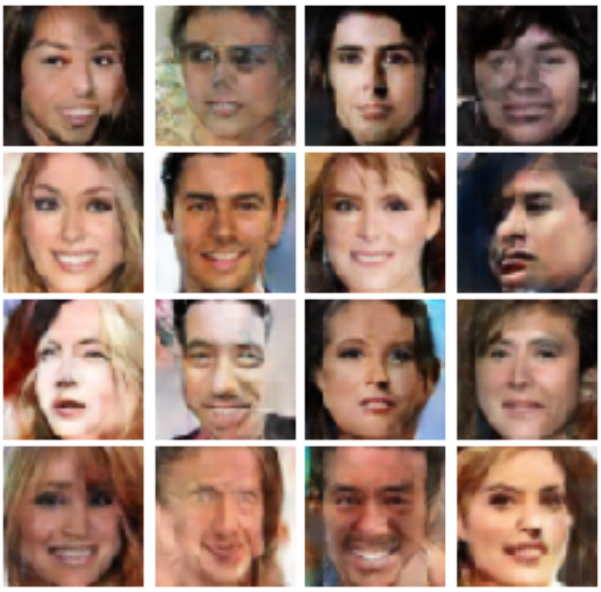
```
Saving the model as a checkpoint...
EPOCH:  29
Iter: 27800, D: 0.04394, G:0.4892
```



```
Iter: 28000, D: 0.01373, G:0.5806
```



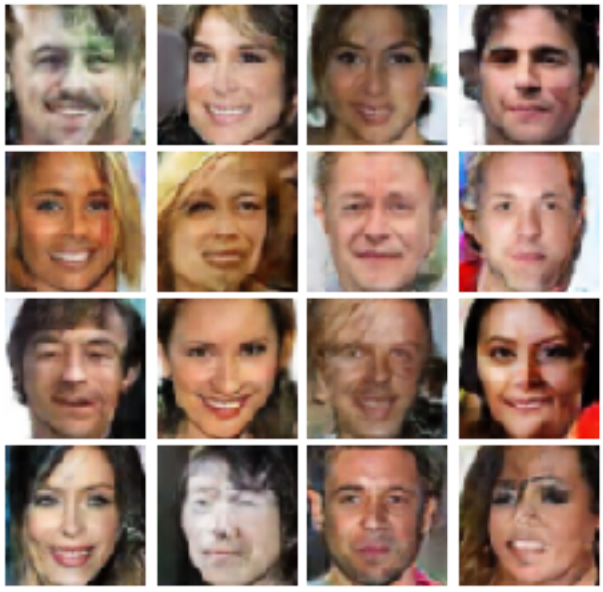
```
Iter: 28200, D: 0.02286, G:0.3249
```



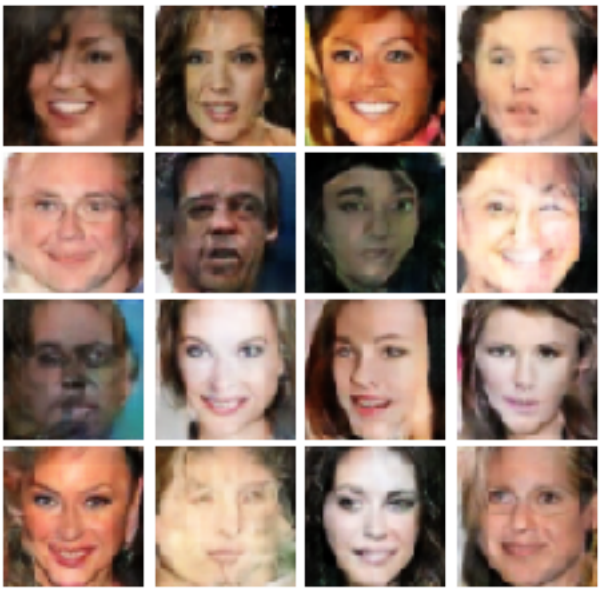
```
Iter: 28400, D: 0.04206, G:0.3032
```

```
Saving the model as a checkpoint...
EPOCH:  30
Iter: 28600, D: 0.02368, G:0.4357
```

Iter: 28800, D: 0.1061, G:0.5641

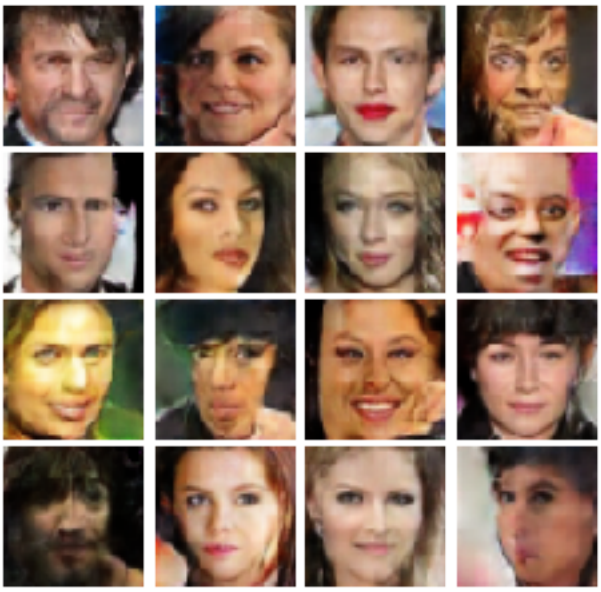


Iter: 29000, D: 0.03452, G:0.5016



Iter: 29200, D: 0.1048, G:0.6234

Iter: 29400, D: 0.0151, G:0.528



```
Saving the model as a checkpoint...
```

4.0.5 Section 2.6 Conclusion? [10 pts]

You have successfully trained you GAN models. Hurray! Now, we want you to answer the following question in few sentences. 1. Have you observed any difference between GAN and LSGAN (e.g. training, generation quality)?

The LSGAN are more stable than GAN at training. However, I didn't observed significant differences for results of GAN and LSGAN.

2. Did your GAN models generate you diverse faces? Or did it should give you similar stuff all the time?

Yes, my GAN models generate diverse faces.

3. How can you quantitatively evaluate your GAN model? Is the metric(s) meaningful? Is there any drawback about this metric(s)?

Inception Score (IS) is perhaps the most widely adopted score for GAN evaluation. A higher inception score indicates better-quality generated images.

FID performs well in terms of discriminability, robustness and computational efficiency. It has been shown that FID is consistent with human judgments and is more robust to noise than IS. A lower FID score indicates more realistic images that match the statistical properties of real images.

4.0.6 Guidelines for Downloading PDF in Google Colab

- Run below cells only in Google Colab, Comment out in case of Jupyter notebook

[ ]:
```
#Run below two lines (in google colab), installation steps to get .pdf of the␣
 ↪notebook

!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc

# After installation, comment above two lines and run again to remove␣
 ↪installation comments from the notebook.
```

[ ]:
```
# Find path to your notebook file in drive and enter in below line

!jupyter nbconvert --to PDF "your_notebook_path_here/DL_Assignment_5.ipynb"

#Example: "/content/drive/My Drive/DL_Fall_2020/Assignment_5/DL_Assignment_5.
 ↪ipynb"
```