MP4

April 17, 2022

1 Generative Adversarial Networks

For this part of the assignment you implement two different types of generative adversarial networks. We will train the networks on a dataset of cat face images.

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

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

%load_ext autoreload
%autoreload 2
```

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

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

2 GAN loss functions

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

2.0.1 GAN loss

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

The generator loss is given by:

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

and the discriminator loss is:

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

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

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

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

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

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

2.0.2 Least Squares GAN loss

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

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

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

and the discriminator loss:

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

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`).

[16]:
```
from gan.losses import ls_discriminator_loss, ls_generator_loss
```

3 GAN model architecture

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

We recommend the following architectures which are inspired by DCGAN:

Discriminator:

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

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

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

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

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

Generator:

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

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

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

[6]:
```
from gan.models import Discriminator, Generator
```

4 Data loading

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

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

[7]:
```
batch_size = 32
imsize = 64
cat_root = './cats'
```

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

  # Example use of RandomCrop:
  transforms.Resize(int(1.15 * imsize)),
  transforms.RandomCrop(imsize),
]))

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

4.0.1 Visualize dataset

[8]:
```
from gan.utils import show_images

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

5 Training

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

[9]:
```
NOISE_DIM = 100
NUM_EPOCHS = 50
learning_rate = 0.001
```

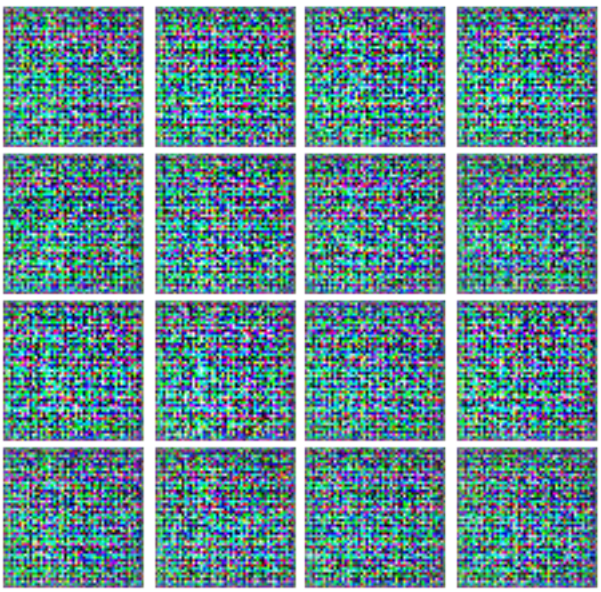
5.0.1 Train GAN

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

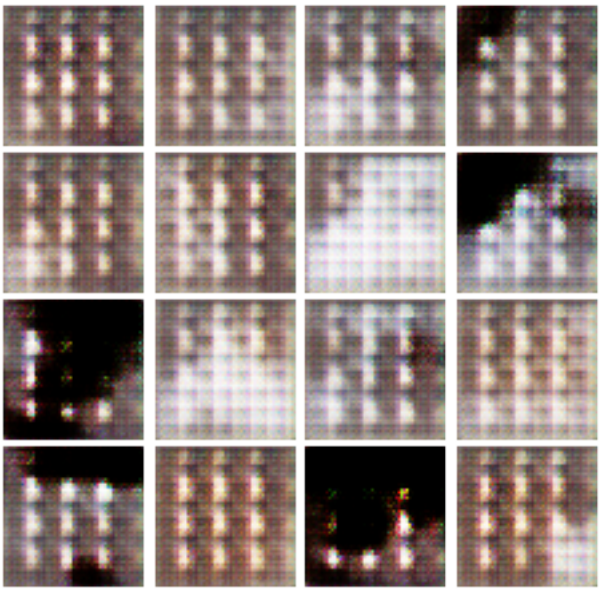
[11]:
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5,␣
↪0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5,␣
↪0.999))
```

[12]:
```
# original gan
train(D, G, D_optimizer, G_optimizer, discriminator_loss,
          generator_loss, num_epochs=NUM_EPOCHS, show_every=250,
          batch_size=batch_size, train_loader=cat_loader_train, device=device)
```

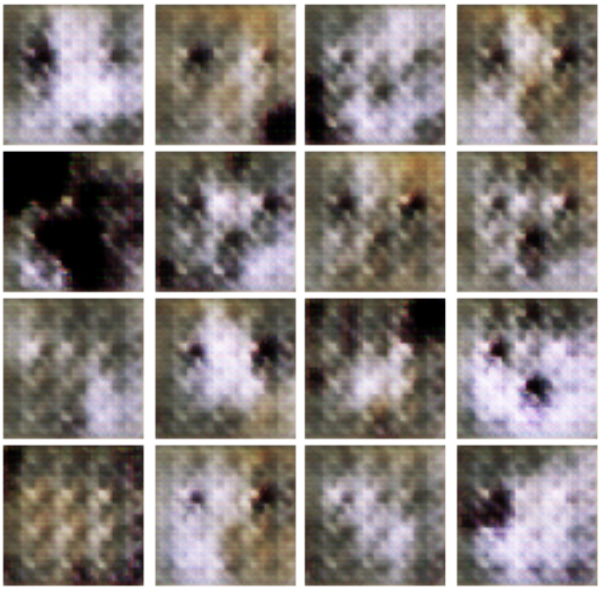
```
EPOCH:  1
Iter: 0, D: 0.5419, G:0.2016
```



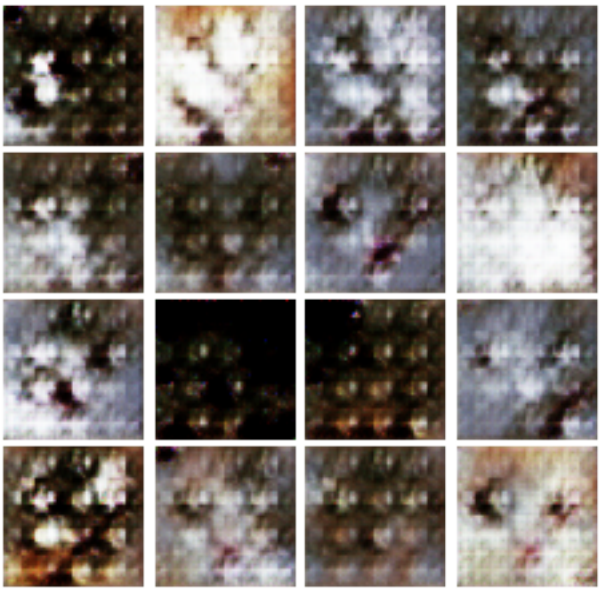
```
Iter: 250, D: 0.5134, G:1.418
```



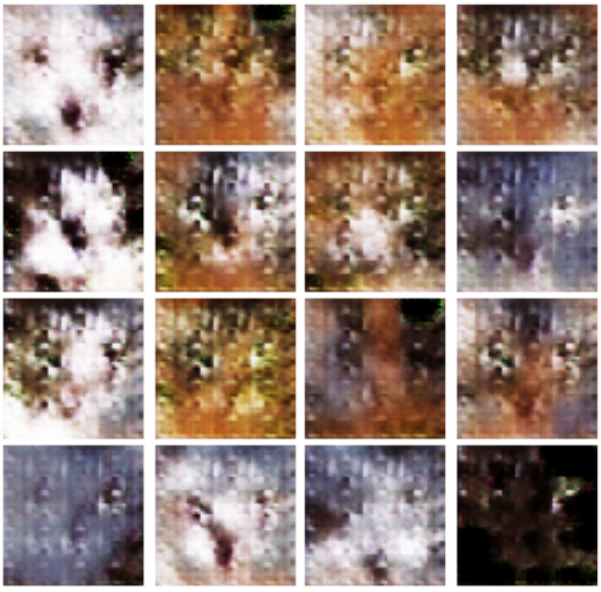
EPOCH: 2
Iter: 500, D: 0.2871, G:0.997



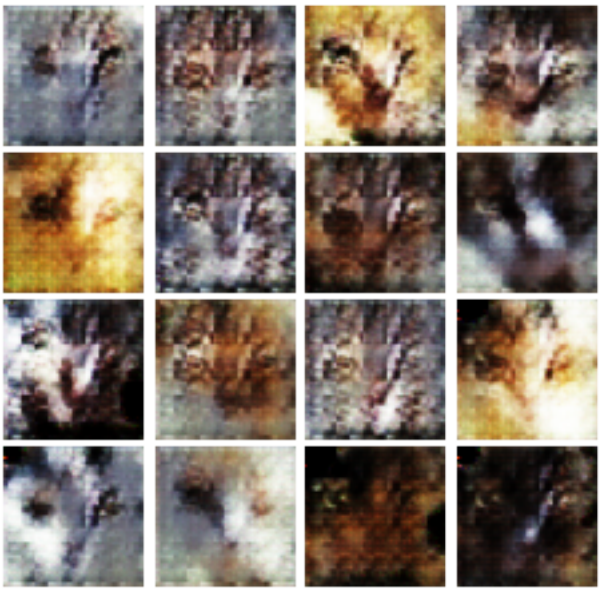
Iter: 750, D: 0.2497, G:1.194



```
EPOCH:  3
Iter: 1000, D: 0.7794, G:0.9906
```

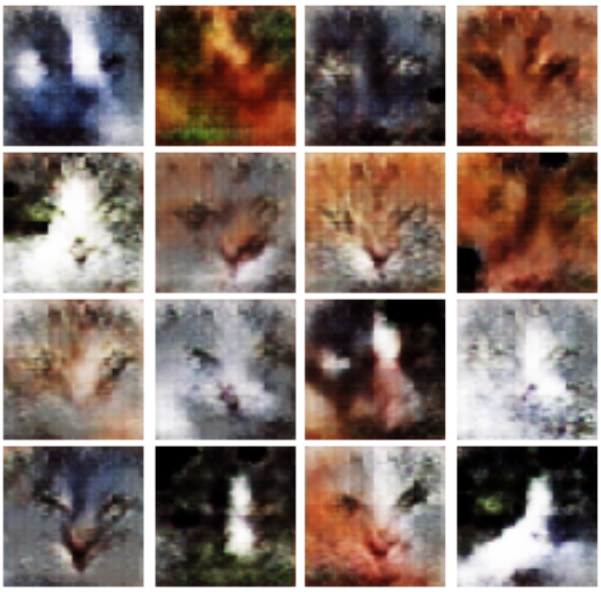


```
Iter: 1250, D: 0.279, G:1.692
```

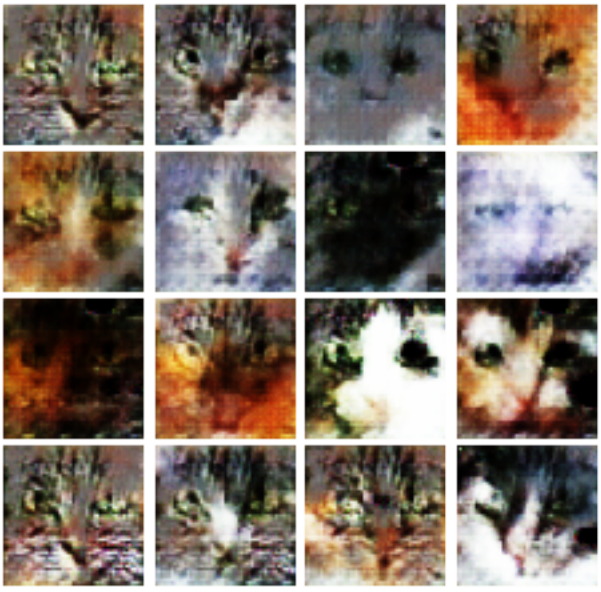


EPOCH: 4
Iter: 1500, D: 0.6582, G:3.069

Iter: 1750, D: 0.2601, G:0.8415



```
EPOCH:  5
Iter: 2000, D: 0.3582, G:2.266
```

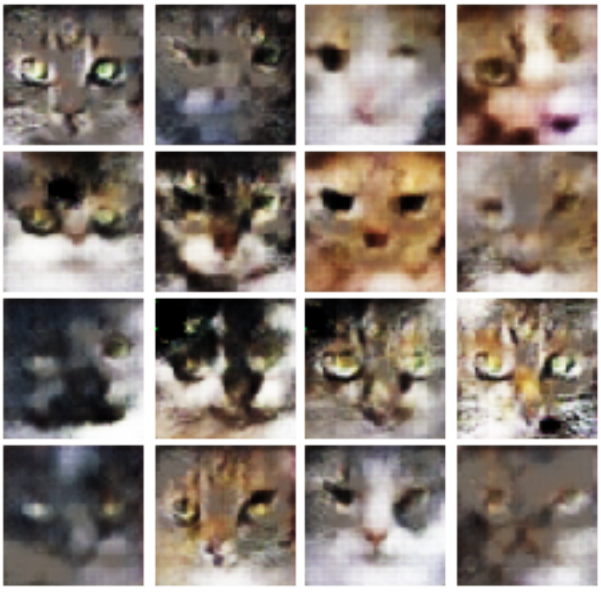


```
Iter: 2250, D: 0.2428, G:2.412
```

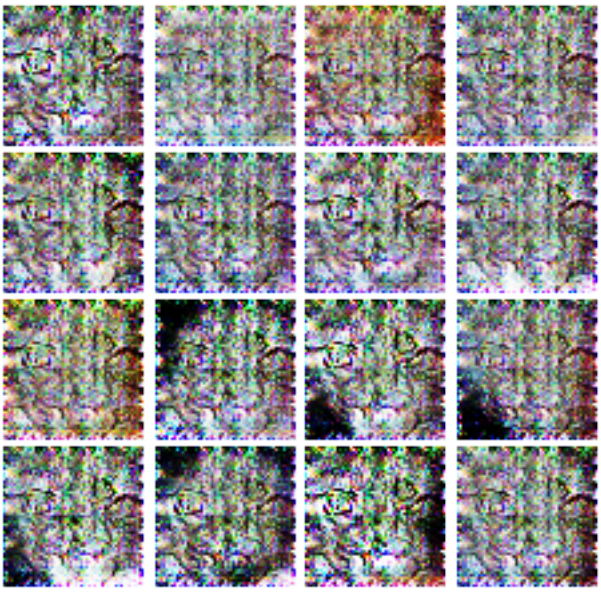
EPOCH: 6
Iter: 2500, D: 0.3178, G:0.7039

Iter: 2750, D: 0.1657, G:2.017

EPOCH: 7
Iter: 3000, D: 0.3682, G:4.999



Iter: 3250, D: 0.01325, G:3.828



```
EPOCH:  8
Iter: 3500, D: 0.185, G:2.988
```

```
Iter: 3750, D: 0.06524, G:3.066
```

```
EPOCH:  9
Iter: 4000, D: 0.1011, G:1.772
```

```
Iter: 4250, D: 0.09664, G:2.508
```

EPOCH: 10
Iter: 4500, D: 0.07991, G:3.642

Iter: 4750, D: 0.03636, G:3.717

```
EPOCH:  11
Iter: 5000, D: 0.246, G:8.392
```

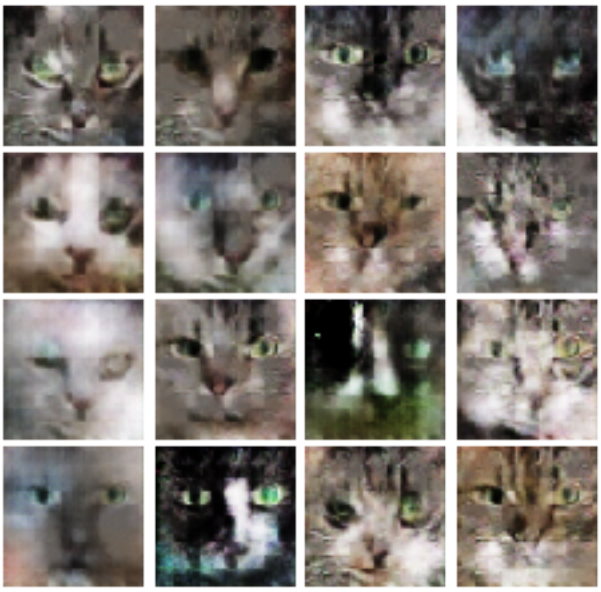
```
Iter: 5250, D: 0.02255, G:4.29
```

EPOCH: 12
Iter: 5500, D: 0.1662, G:10.71

Iter: 5750, D: 0.02816, G:4.73

EPOCH: 13
Iter: 6000, D: 0.04154, G:4.824

Iter: 6250, D: 0.0416, G:4.637



EPOCH: 14
Iter: 6500, D: 0.01875, G:4.207

Iter: 6750, D: 0.08856, G:3.825

```
EPOCH:  15
Iter: 7000, D: 0.01533, G:2.932
```

```
Iter: 7250, D: 0.1214, G:4.029
```

```
EPOCH:  16
Iter: 7500, D: 0.1817, G:7.146
```

```
Iter: 7750, D: 0.06882, G:3.563
```

EPOCH: 17
Iter: 8000, D: 0.04477, G:3.546

Iter: 8250, D: 0.09635, G:5.121

EPOCH: 18
Iter: 8500, D: 0.07346, G:4.184

Iter: 8750, D: 0.03715, G:2.944

EPOCH: 19
Iter: 9000, D: 0.07545, G:4.252

Iter: 9250, D: 0.04984, G:3.496

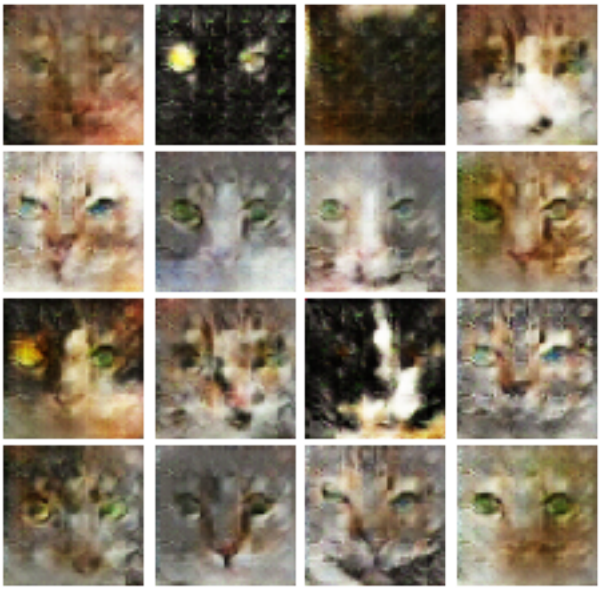
```
EPOCH:  20
Iter: 9500, D: 0.03367, G:4.386
```

```
Iter: 9750, D: 0.008548, G:5.921
```

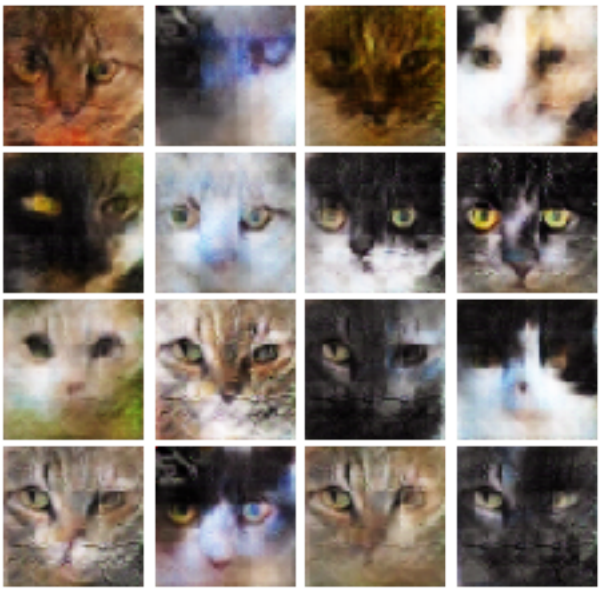
EPOCH: 21
Iter: 10000, D: 0.072, G:3.505

Iter: 10250, D: 0.03173, G:3.53

EPOCH: 22
Iter: 10500, D: 0.04577, G:4.67



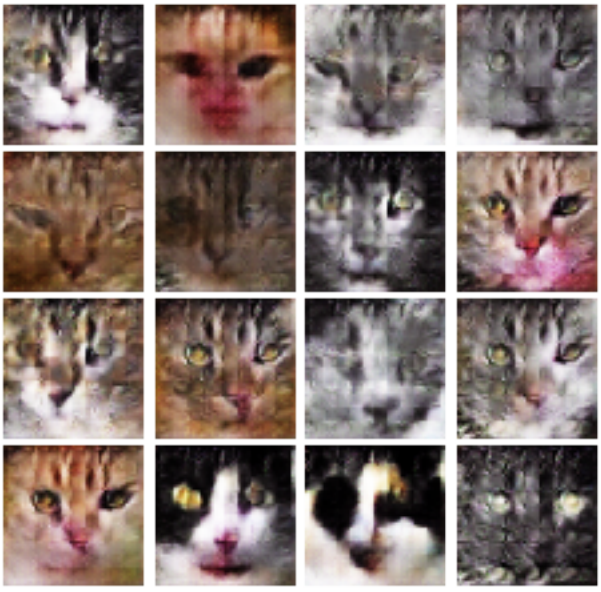
Iter: 10750, D: 0.02583, G:3.049



EPOCH: 23
Iter: 11000, D: 0.03306, G:4.18

Iter: 11250, D: 0.2814, G:6.919

```
EPOCH:  24
Iter: 11500, D: 0.03427, G:3.726
```

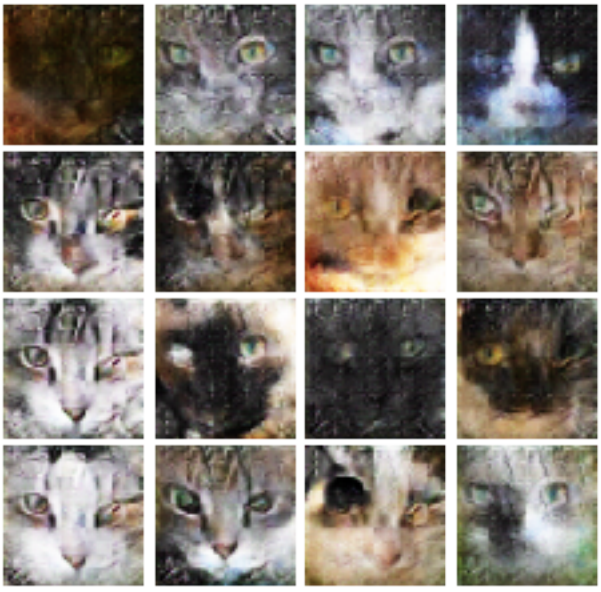


```
Iter: 11750, D: 0.08617, G:6.449
```

EPOCH: 25
Iter: 12000, D: 0.01915, G:3.488

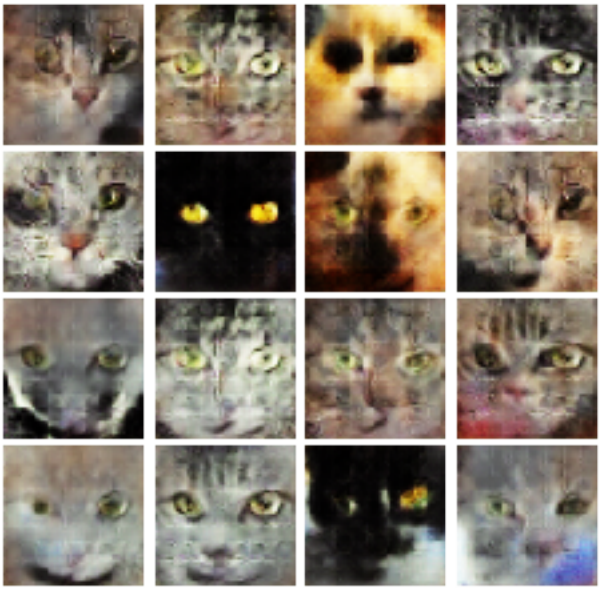
Iter: 12250, D: 0.02077, G:4.9

EPOCH: 26
Iter: 12500, D: 0.09245, G:2.249

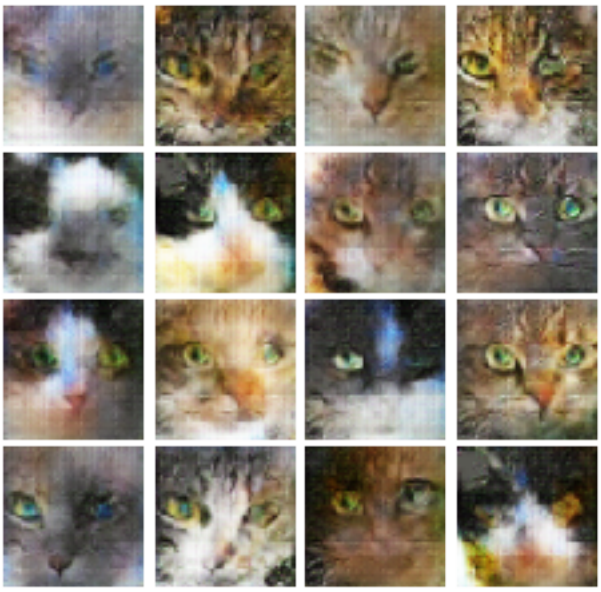


Iter: 12750, D: 0.06511, G:3.141

EPOCH: 27
Iter: 13000, D: 0.05285, G:4.435



Iter: 13250, D: 0.02902, G:6.772



EPOCH: 28
Iter: 13500, D: 0.01659, G:3.788

Iter: 13750, D: 0.03198, G:5.224

EPOCH: 29
Iter: 14000, D: 0.1571, G:5.411

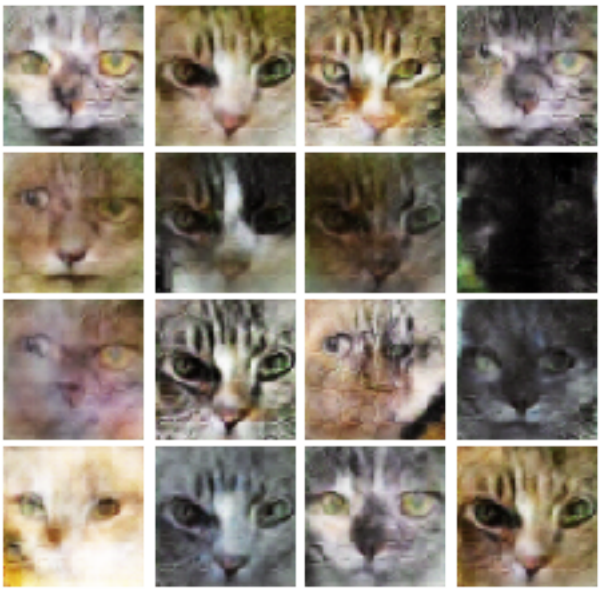
Iter: 14250, D: 0.03265, G:4.094

EPOCH: 30
Iter: 14500, D: 0.01531, G:4.302

Iter: 14750, D: 0.02788, G:3.681

```
EPOCH:  31
Iter: 15000, D: 0.03224, G:4.691
```

```
Iter: 15250, D: 0.006476, G:5.744
```

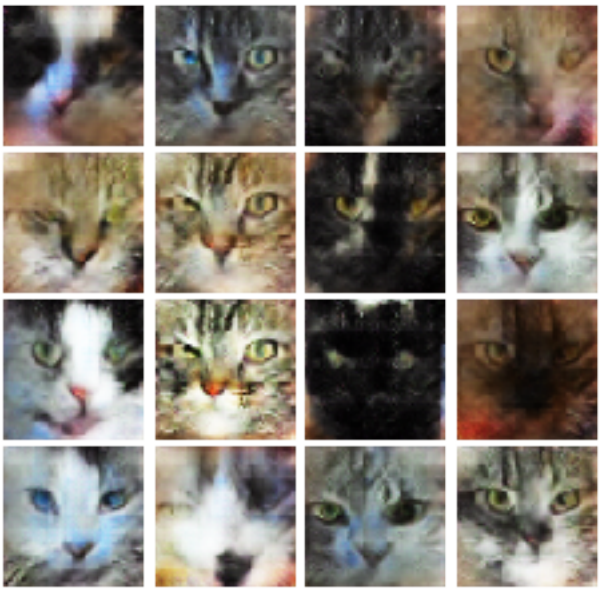


EPOCH: 32
Iter: 15500, D: 0.02498, G:5.908

EPOCH: 33
Iter: 15750, D: 0.02207, G:4.454

Iter: 16000, D: 0.1395, G:7.691

EPOCH: 34
Iter: 16250, D: 0.02109, G:4.569



Iter: 16500, D: 0.02601, G:6.723

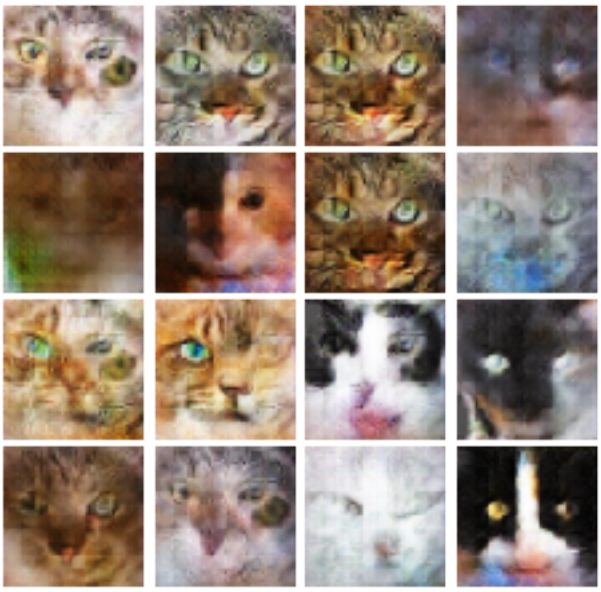
EPOCH: 35
Iter: 16750, D: 0.0222, G:4.822

Iter: 17000, D: 0.01012, G:6.653

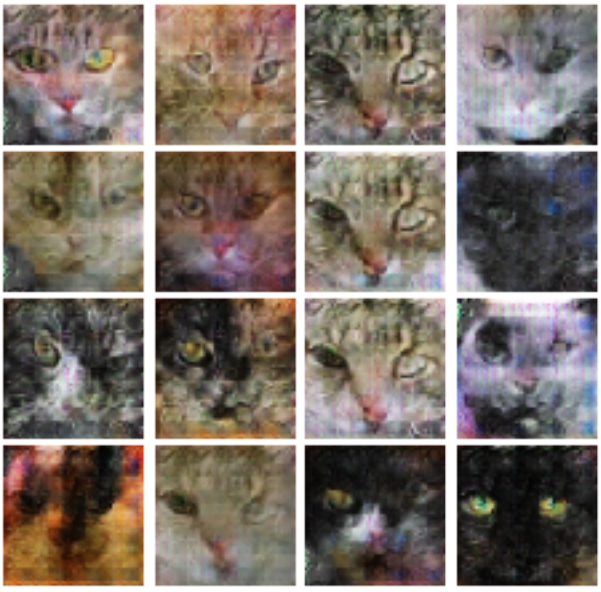
EPOCH: 36
Iter: 17250, D: 0.02458, G:3.531

Iter: 17500, D: 0.1081, G:2.549

EPOCH: 37
Iter: 17750, D: 0.02056, G:5.022



Iter: 18000, D: 0.05432, G:6.221



EPOCH: 38
Iter: 18250, D: 0.02272, G:4.345

Iter: 18500, D: 0.03192, G:6.023

EPOCH: 39
Iter: 18750, D: 0.02688, G:3.838

```
Iter: 19000, D: 0.01843, G:4.612
```

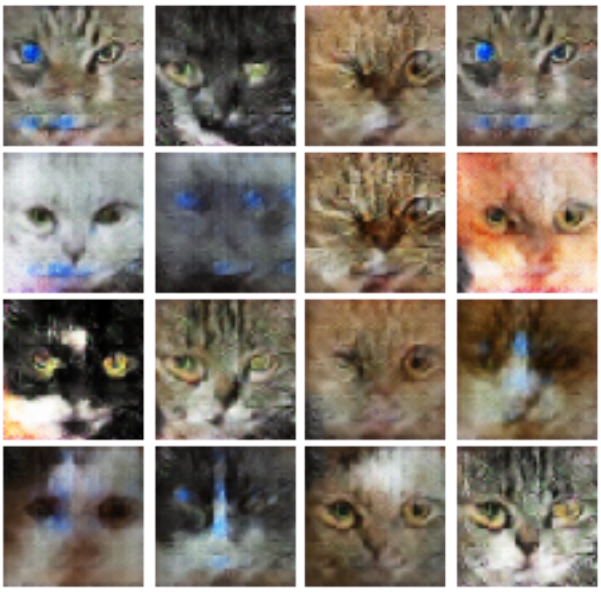
```
EPOCH:  40
Iter: 19250, D: 0.01598, G:3.572
```

Iter: 19500, D: 0.05785, G:3.486

EPOCH: 41
Iter: 19750, D: 0.01231, G:4.04

Iter: 20000, D: 0.008936, G:2.982

EPOCH: 42
Iter: 20250, D: 0.03987, G:4.177

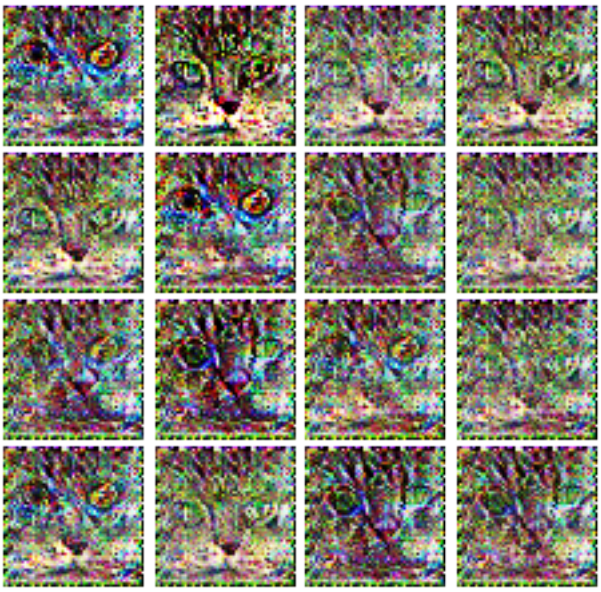


Iter: 20500, D: 0.00758, G:4.233

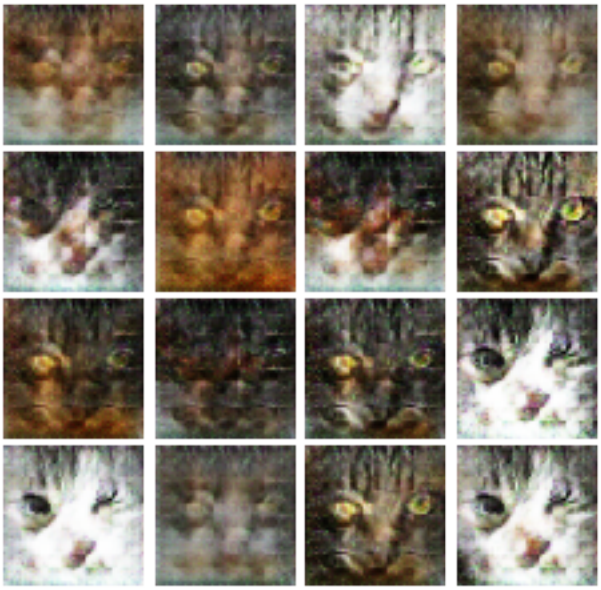
EPOCH: 43
Iter: 20750, D: 0.06202, G:2.718

```
Iter: 21000, D: 0.003143, G:7.974
```

```
EPOCH:  44
Iter: 21250, D: 0.0006643, G:7.217
```



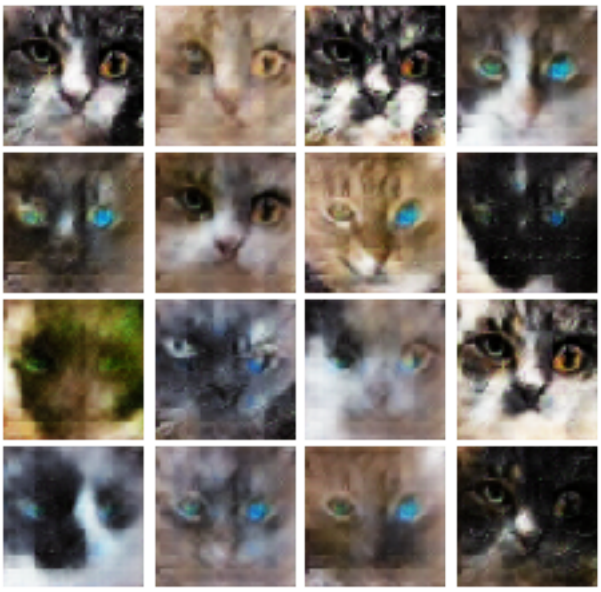
Iter: 21500, D: 0.01106, G:5.186



EPOCH: 45
Iter: 21750, D: 0.01132, G:4.39

Iter: 22000, D: 0.007328, G:4.708

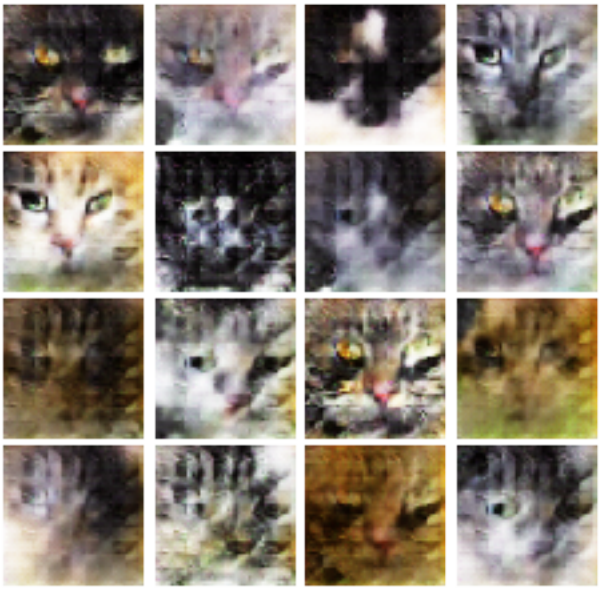
EPOCH: 46
Iter: 22250, D: 0.0142, G:11.16



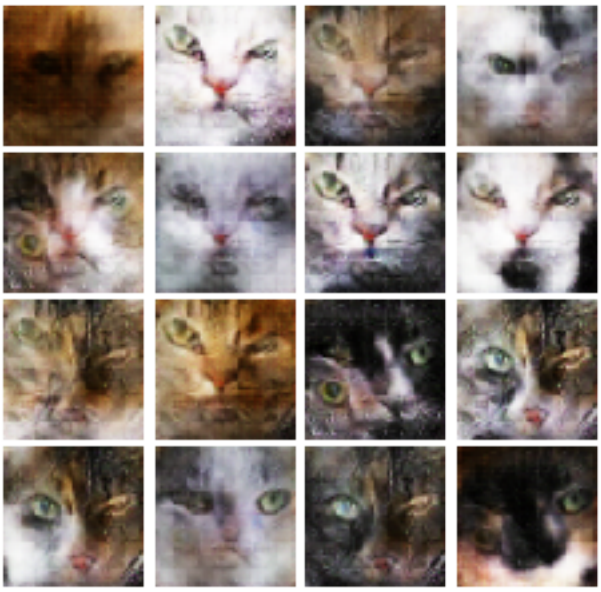
Iter: 22500, D: 0.03819, G:6.969

EPOCH: 47
Iter: 22750, D: 0.02217, G:4.886

Iter: 23000, D: 0.01085, G:6.207



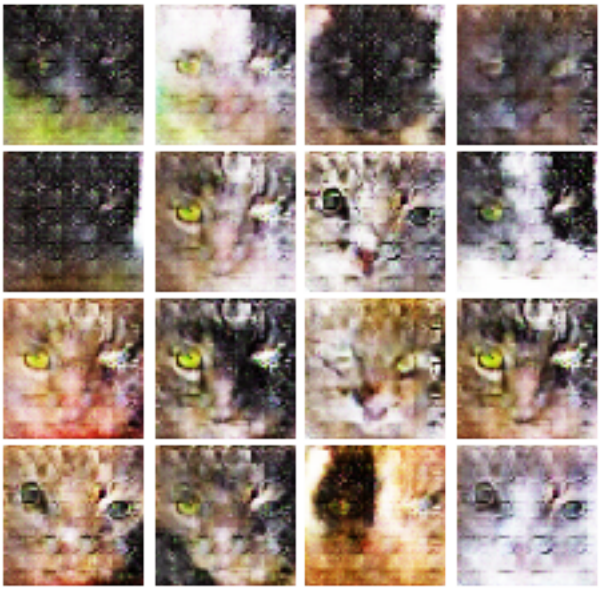
EPOCH: 48
Iter: 23250, D: 0.008789, G:5.004



Iter: 23500, D: 0.0534, G:6.307

EPOCH: 49
Iter: 23750, D: 0.08832, G:5.105

Iter: 24000, D: 0.0116, G:6.107



EPOCH: 50
Iter: 24250, D: 0.01134, G:6.116

Iter: 24500, D: 0.02577, G:4.939

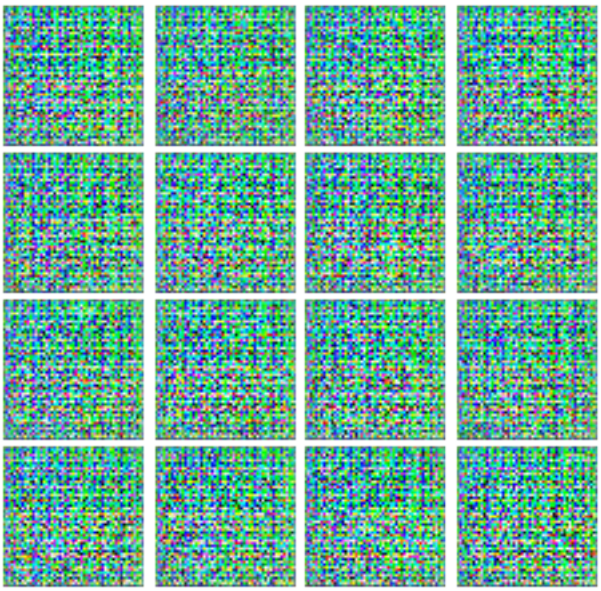
5.0.2 Train LS-GAN

[13]:
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

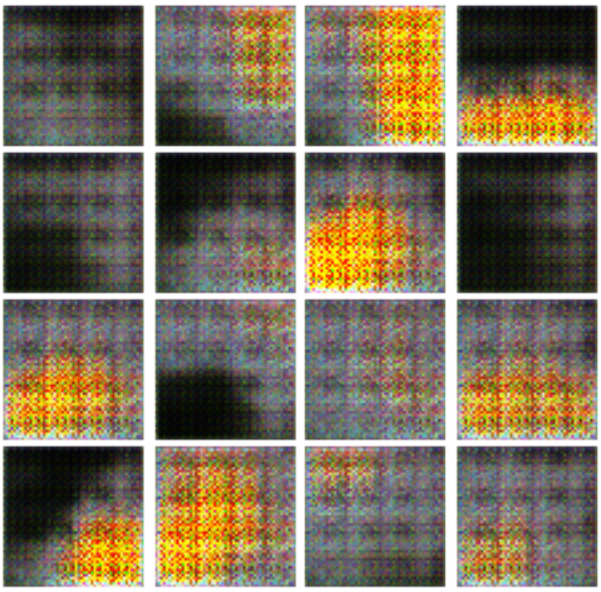
[14]:
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5,␣
↪0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5,␣
↪0.999))
```

[17]:
```
# ls-gan
train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
          ls_generator_loss, num_epochs=NUM_EPOCHS, show_every=250,
          batch_size=batch_size, train_loader=cat_loader_train, device=device)
```

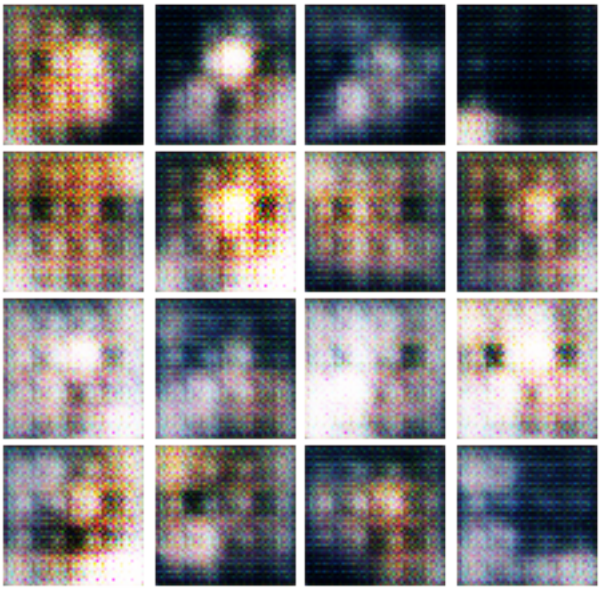
```
EPOCH:  1
Iter: 0, D: 0.9445, G:255.7
```



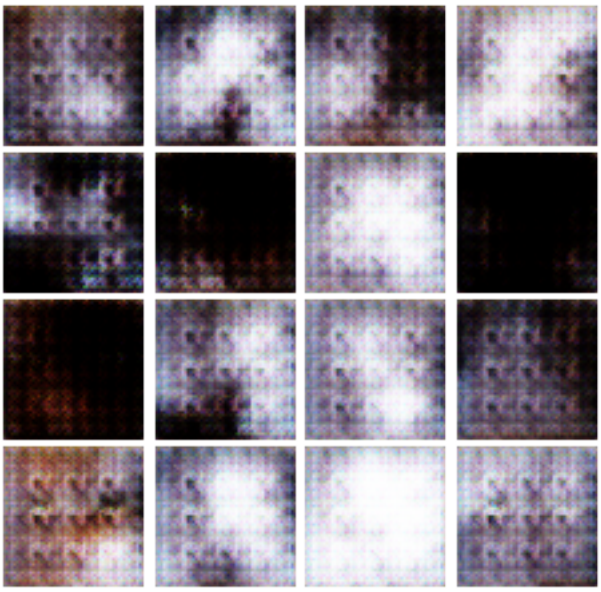
```
Iter: 250, D: 0.1592, G:0.8952
```



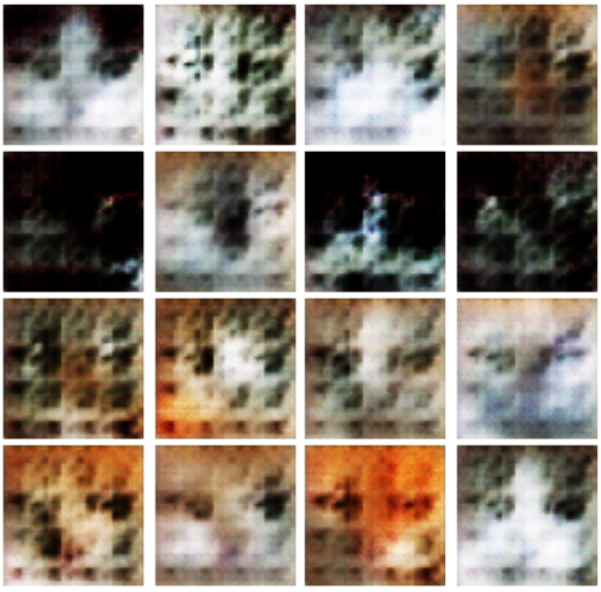
```
EPOCH:  2
Iter: 500, D: 0.1965, G:0.9881
```



```
Iter: 750, D: 0.0825, G:1.38
```

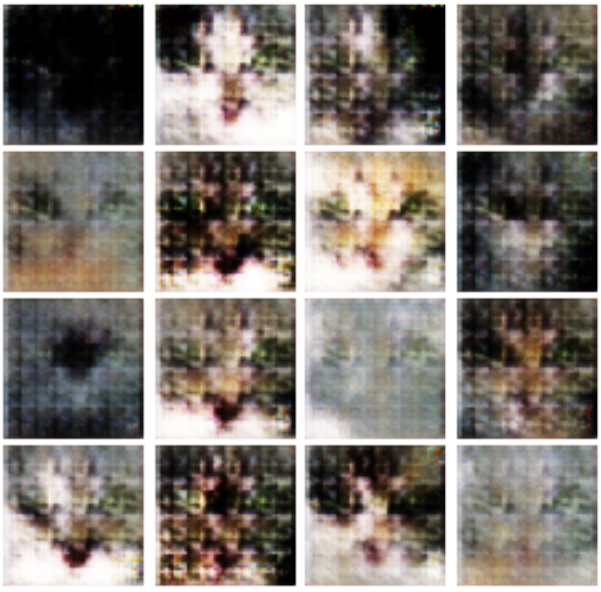


```
EPOCH:  3
Iter: 1000, D: 0.178, G:0.5018
```

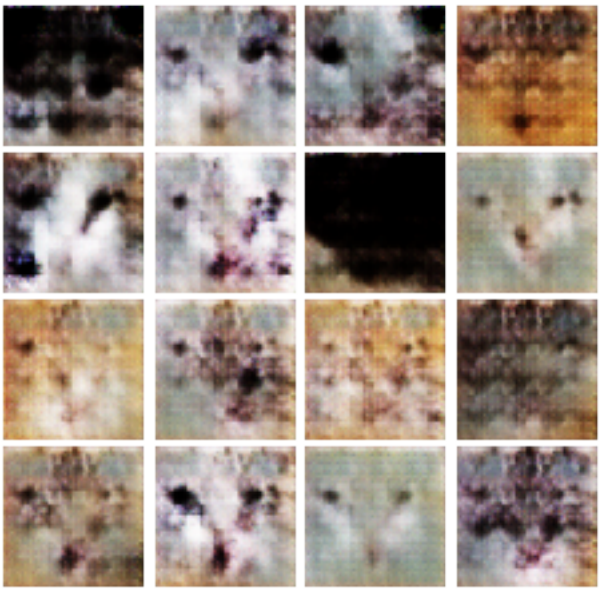


```
Iter: 1250, D: 0.1428, G:0.5411
```

```
EPOCH:  4
Iter: 1500, D: 0.2193, G:0.705
```

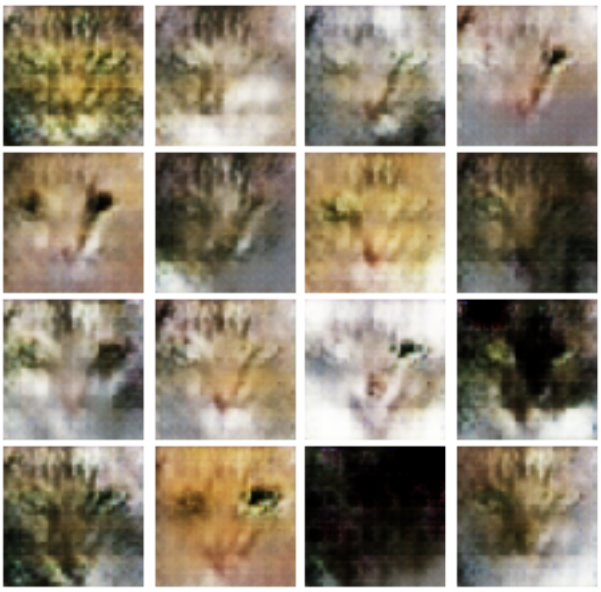


```
Iter: 1750, D: 0.189, G:0.6023
```

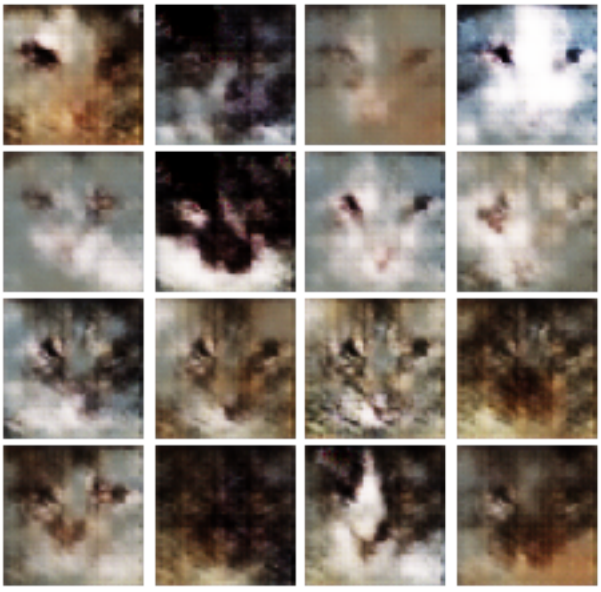


```
EPOCH:  5
Iter: 2000, D: 0.2069, G:0.7338
```

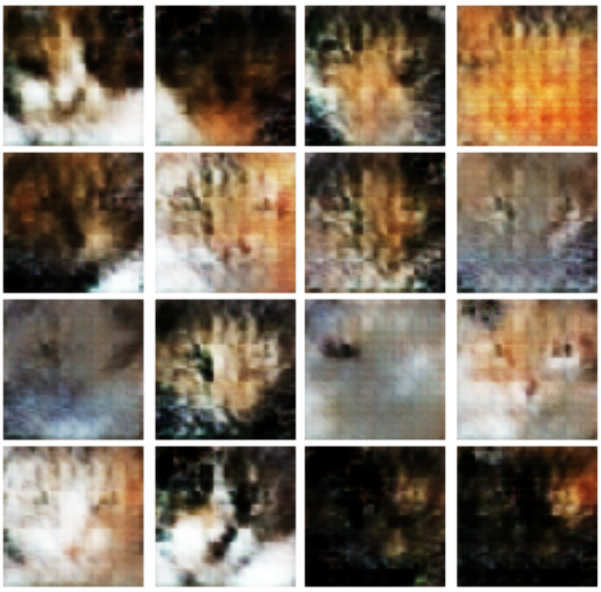
```
Iter: 2250, D: 0.5814, G:0.6087
```



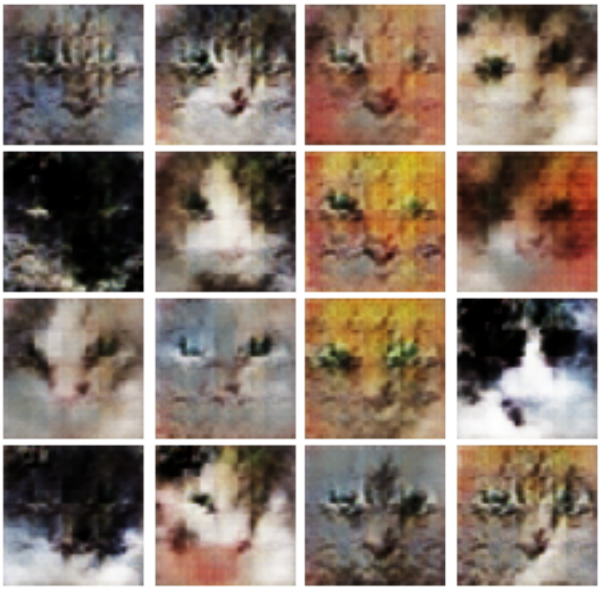
EPOCH: 6
Iter: 2500, D: 0.1713, G:0.5729



Iter: 2750, D: 0.1876, G:0.764



```
EPOCH:  7
Iter: 3000, D: 0.1538, G:0.9212
```



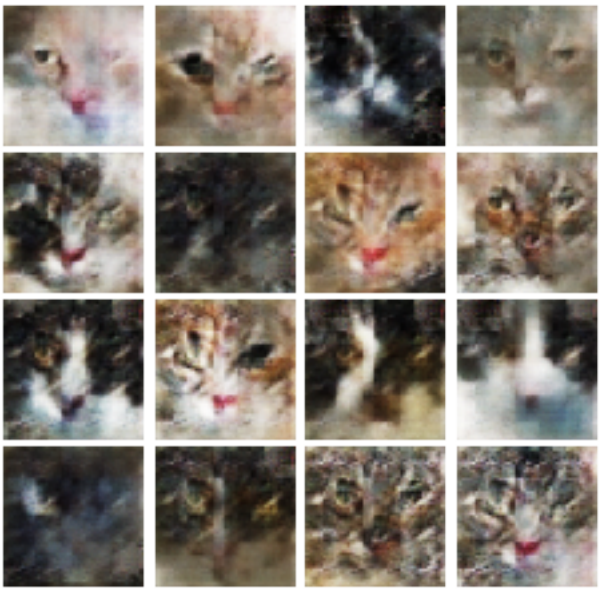
```
Iter: 3250, D: 0.1709, G:0.7127
```

```
EPOCH:  8
Iter: 3500, D: 0.2482, G:1.569
```

```
Iter: 3750, D: 0.06651, G:1.194
```

```
EPOCH:  9
Iter: 4000, D: 0.1007, G:0.9568
```

```
Iter: 4250, D: 0.0631, G:0.6403
```



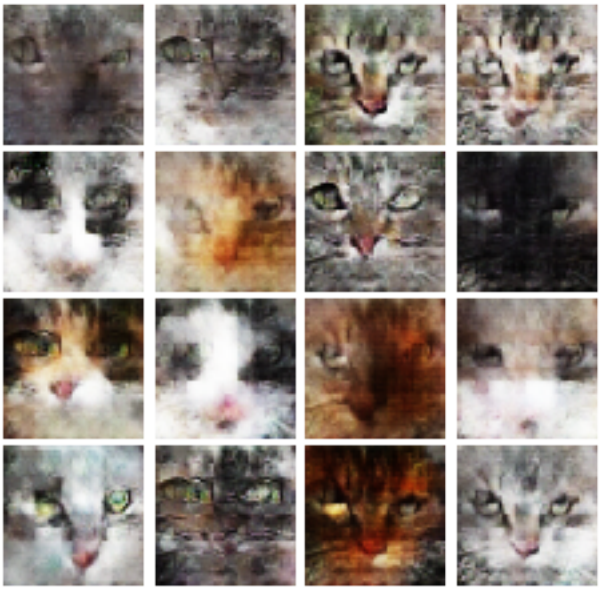
```
EPOCH:  10
Iter: 4500, D: 0.09882, G:1.011
```

```
Iter: 4750, D: 0.1036, G:1.713
```

```
EPOCH:  11
Iter: 5000, D: 0.1462, G:1.594
```

```
Iter: 5250, D: 0.02819, G:0.885
```

```
EPOCH:  12
Iter: 5500, D: 0.05475, G:0.9914
```



```
Iter: 5750, D: 0.04547, G:1.298
```

EPOCH: 13
Iter: 6000, D: 0.08845, G:1.26

Iter: 6250, D: 0.1224, G:1.739

EPOCH: 14
Iter: 6500, D: 0.06242, G:1.393

Iter: 6750, D: 0.05047, G:1.26

```
EPOCH:  15
Iter: 7000, D: 0.04122, G:1.715
```

```
Iter: 7250, D: 0.05408, G:1.429
```

EPOCH: 16
Iter: 7500, D: 0.04412, G:1.012

Iter: 7750, D: 0.217, G:0.3268

```
EPOCH:  17
Iter: 8000, D: 0.289, G:0.7245
```

```
Iter: 8250, D: 0.07658, G:1.425
```

```
EPOCH:  18
Iter: 8500, D: 0.09123, G:0.8738
```

```
Iter: 8750, D: 0.0441, G:0.8462
```

```
EPOCH:  19
Iter: 9000, D: 0.02897, G:0.8615
```

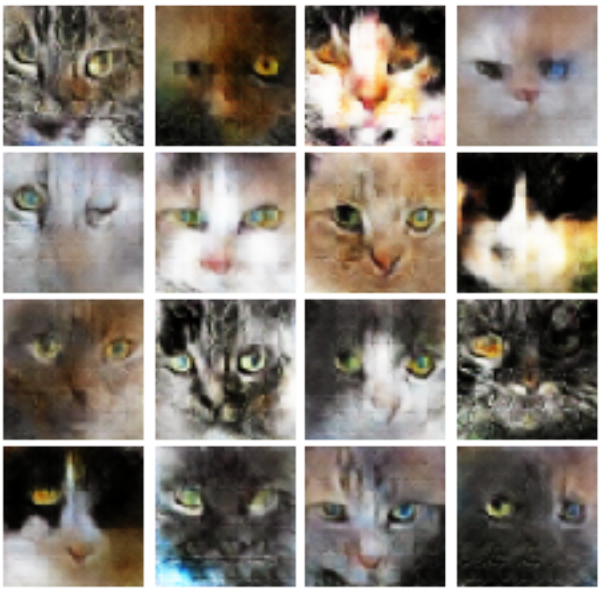
```
Iter: 9250, D: 0.05496, G:0.9372
```

```
EPOCH:  20
Iter: 9500, D: 0.05339, G:1.326
```

```
Iter: 9750, D: 0.01811, G:1.109
```

EPOCH: 21
Iter: 10000, D: 0.06138, G:1.144

Iter: 10250, D: 0.03175, G:0.5981



```
EPOCH:  22
Iter: 10500, D: 0.05124, G:1.549
```

```
Iter: 10750, D: 0.1603, G:0.4874
```

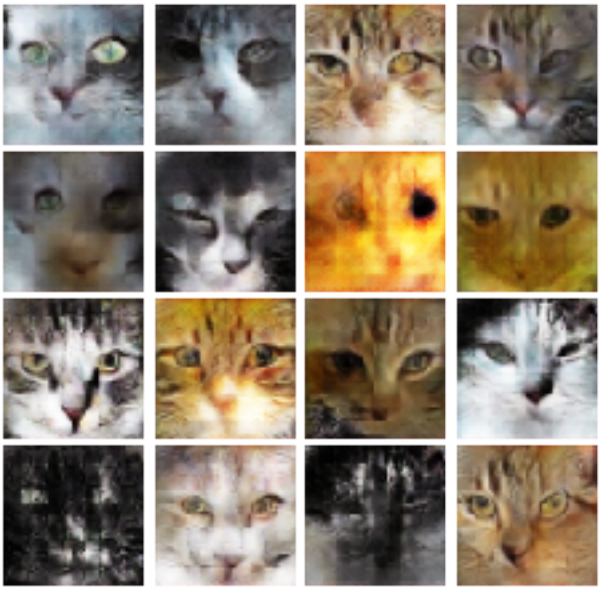
EPOCH: 23
Iter: 11000, D: 0.05563, G:1.103

Iter: 11250, D: 0.1599, G:1.856

```
EPOCH:  24
Iter: 11500, D: 0.01807, G:1.109
```

```
Iter: 11750, D: 0.0394, G:1.245
```

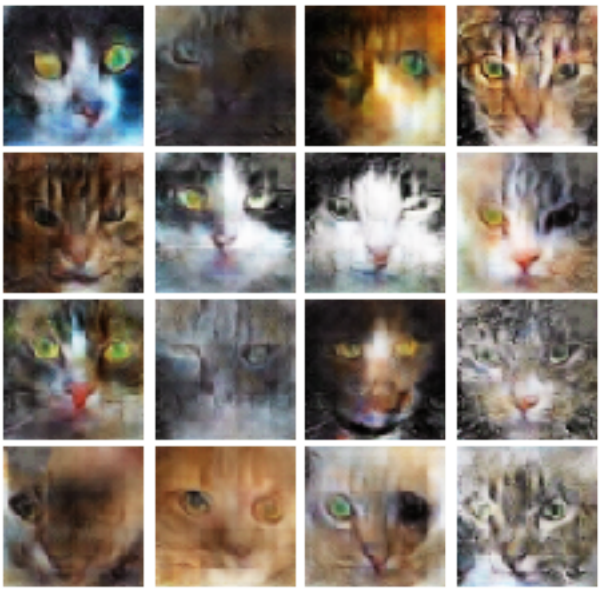
EPOCH: 25
Iter: 12000, D: 0.03951, G:0.848



Iter: 12250, D: 0.02886, G:0.6916

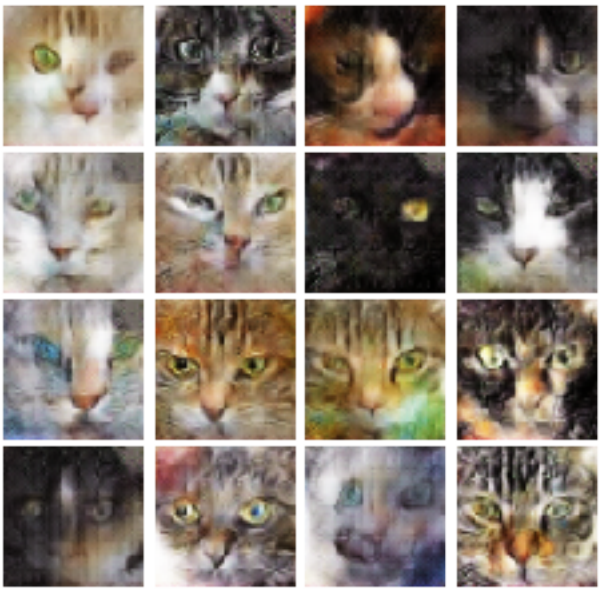
EPOCH: 26
Iter: 12500, D: 0.04181, G:0.917

Iter: 12750, D: 0.01627, G:1.242

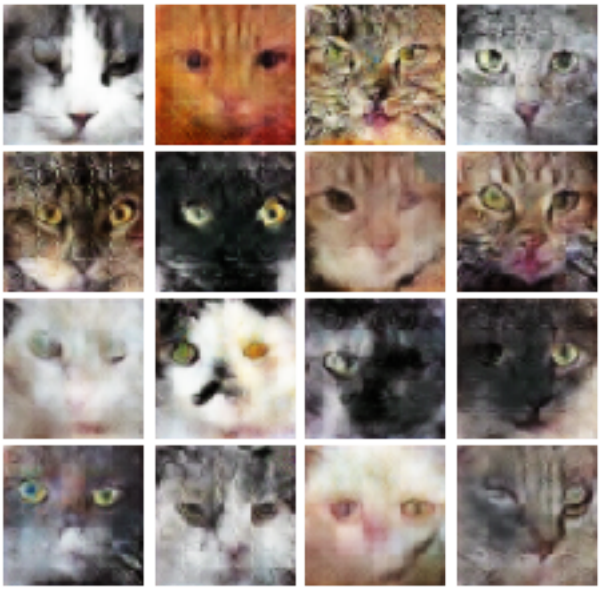


```
EPOCH:  27
Iter: 13000, D: 0.0116, G:1.363
```

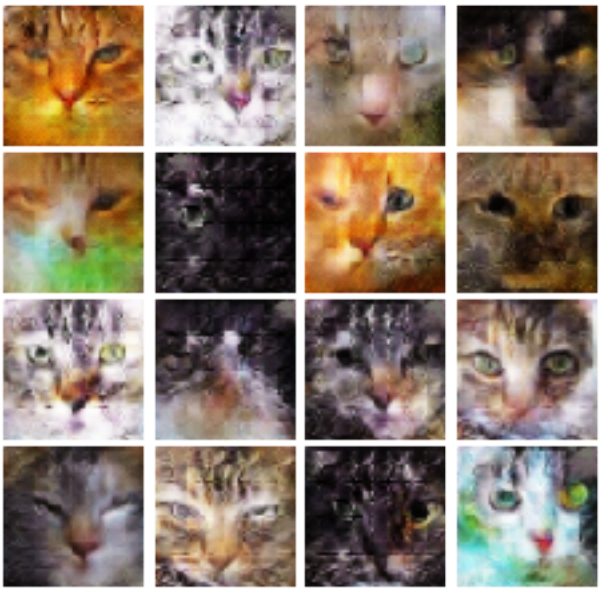
```
Iter: 13250, D: 0.03403, G:0.4368
```



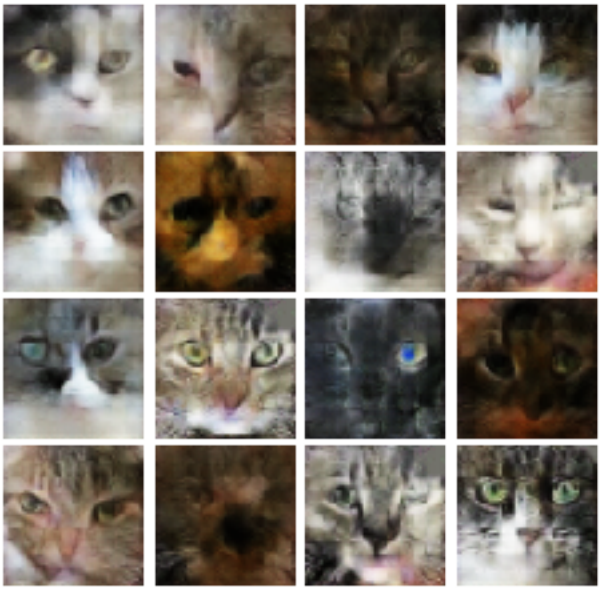
EPOCH: 28
Iter: 13500, D: 0.04512, G:0.8367



Iter: 13750, D: 0.01979, G:0.9807



```
EPOCH:  29
Iter: 14000, D: 0.01616, G:1.115
```



```
Iter: 14250, D: 0.02699, G:1.219
```

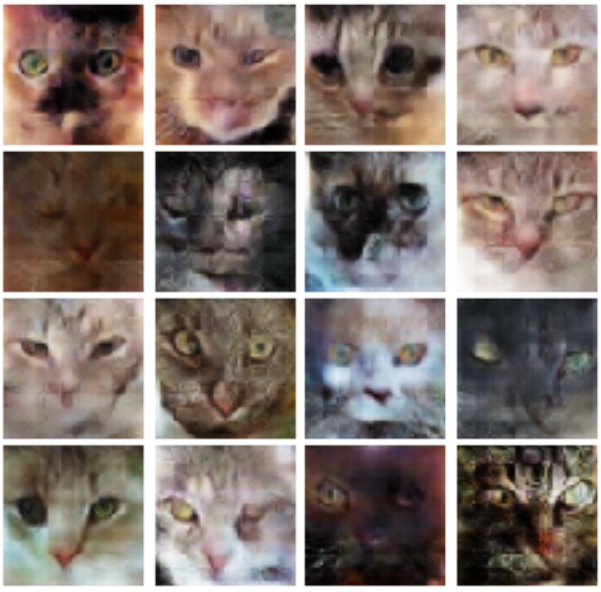
EPOCH: 30
Iter: 14500, D: 0.1159, G:0.7933

Iter: 14750, D: 0.03234, G:0.9464

```
EPOCH:  31
Iter: 15000, D: 0.04441, G:0.7865
```

```
Iter: 15250, D: 0.07084, G:1.266
```

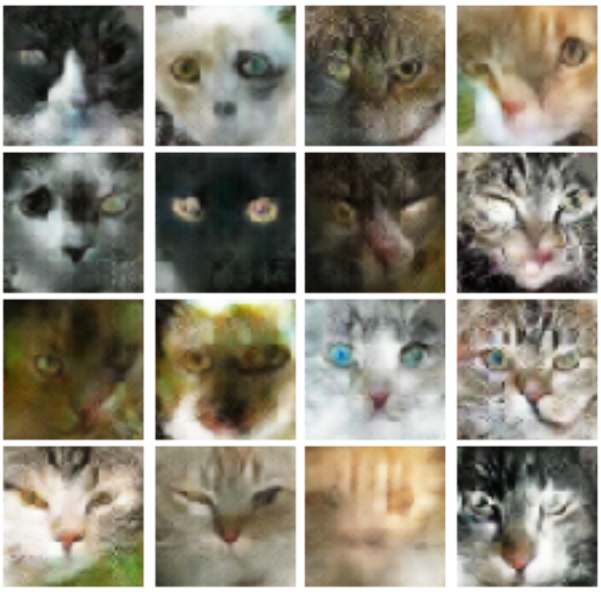
```
EPOCH:  32
Iter: 15500, D: 0.02577, G:1.219
```



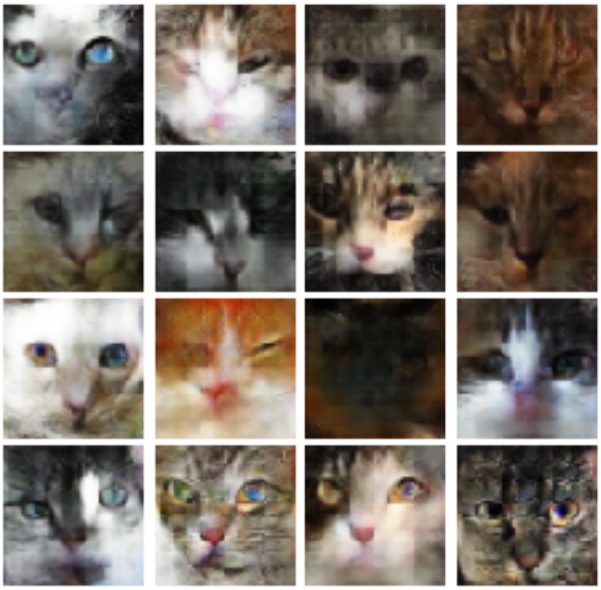
```
EPOCH:  33
Iter: 15750, D: 0.0295, G:1.529
```

Iter: 16000, D: 0.03307, G:1.14

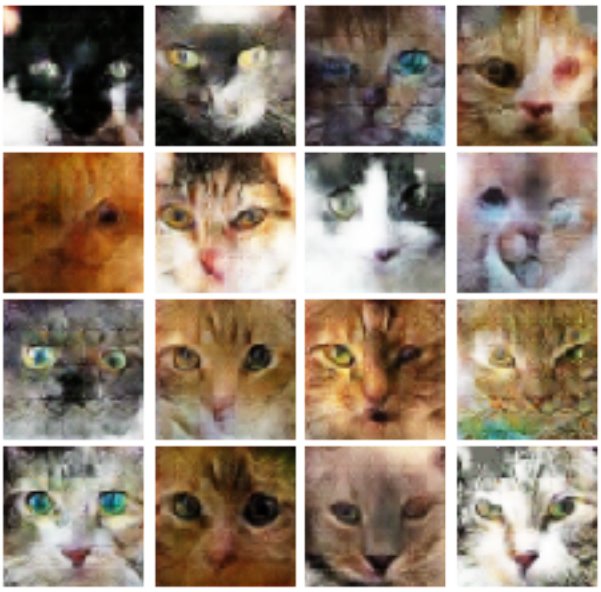
EPOCH: 34
Iter: 16250, D: 0.05605, G:0.7635



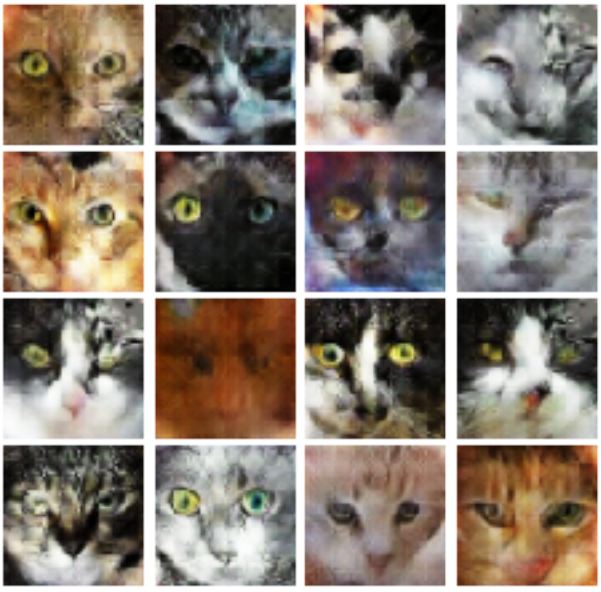
Iter: 16500, D: 0.02344, G:0.9316



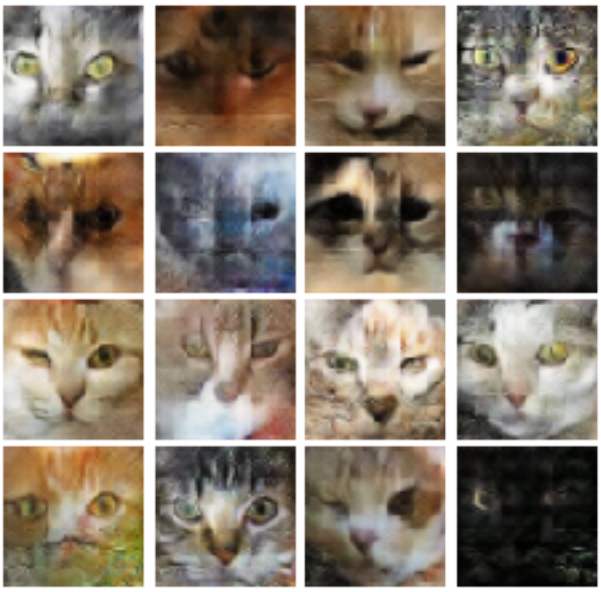
EPOCH: 35
Iter: 16750, D: 0.01, G:1.115



```
Iter: 17000, D: 0.01628, G:0.9987
```

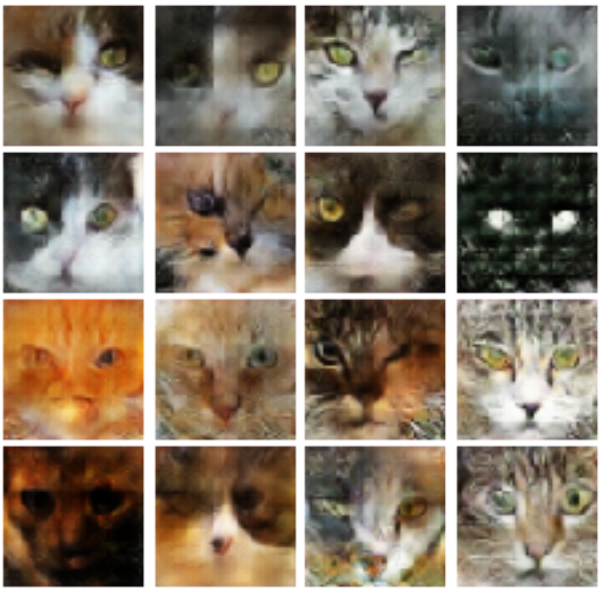


```
EPOCH:  36
Iter: 17250, D: 0.04667, G:0.6505
```



Iter: 17500, D: 0.05236, G:0.5642

EPOCH: 37
Iter: 17750, D: 0.02395, G:0.9202



Iter: 18000, D: 0.02232, G:1.035

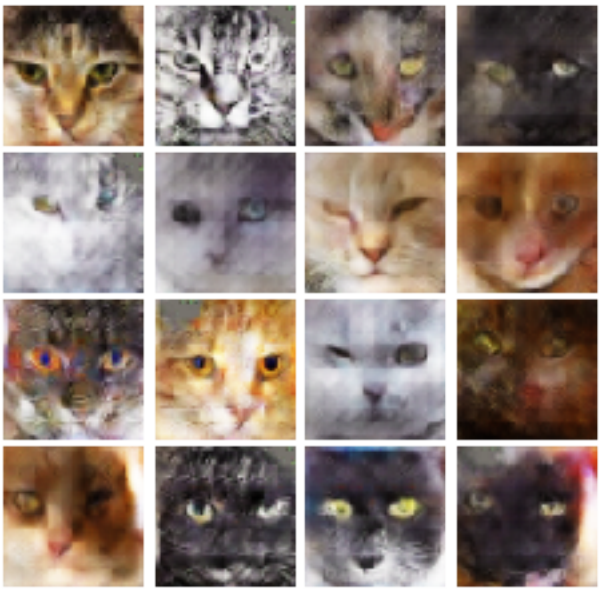
EPOCH: 38
Iter: 18250, D: 0.03833, G:0.7652

Iter: 18500, D: 0.01895, G:0.976

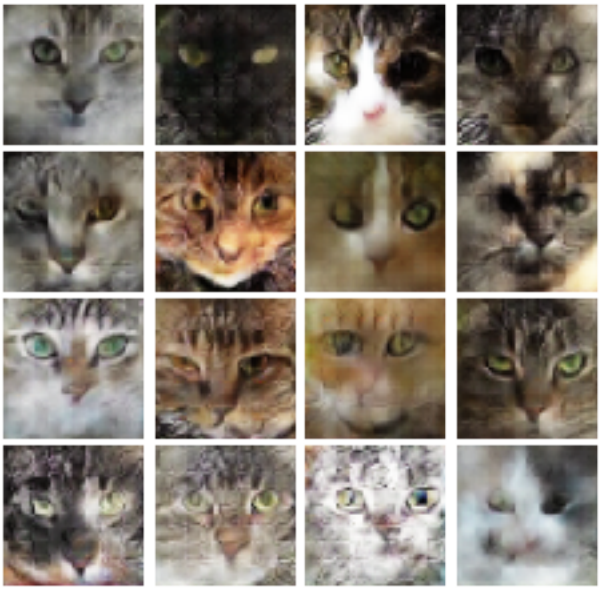
EPOCH: 39
Iter: 18750, D: 0.0136, G:1.003

```
Iter: 19000, D: 0.01103, G:0.8844
```

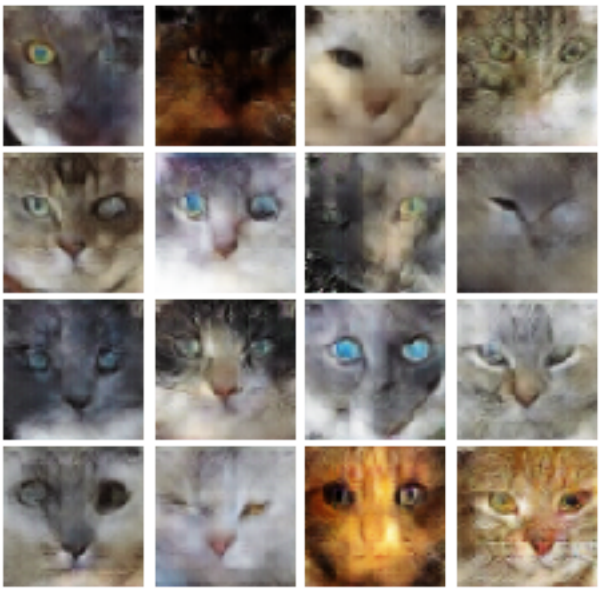
```
EPOCH:  40
Iter: 19250, D: 0.00873, G:0.9108
```



Iter: 19500, D: 0.05041, G:1.116

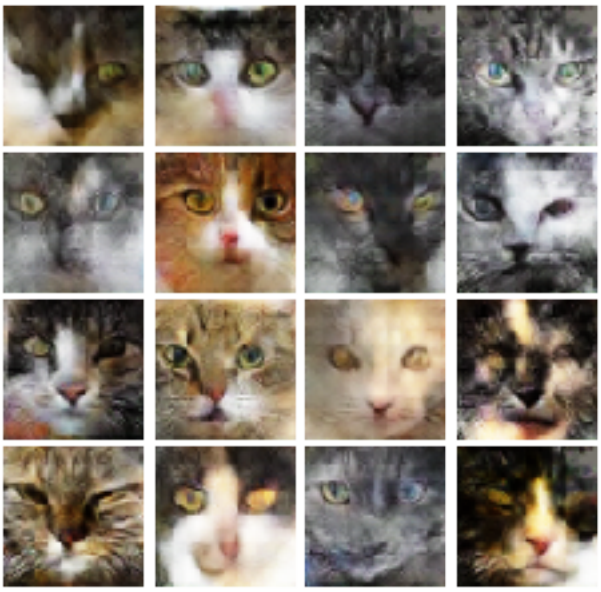


EPOCH: 41
Iter: 19750, D: 0.06259, G:0.5558

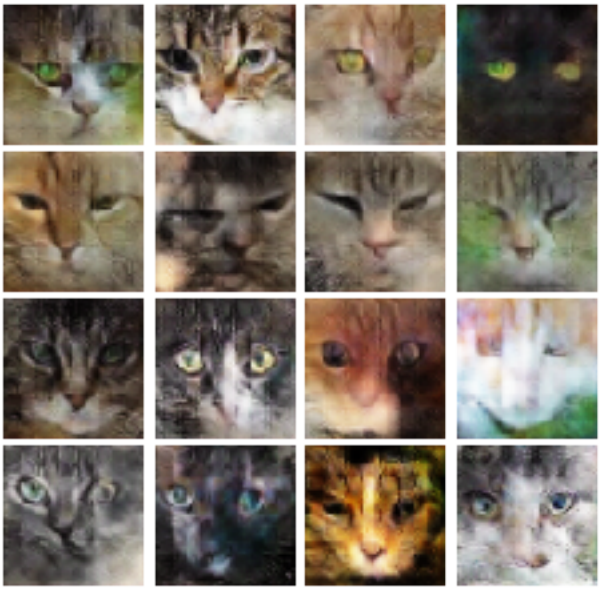


Iter: 20000, D: 0.02012, G:0.9161

EPOCH: 42
Iter: 20250, D: 0.01332, G:0.8587



```
Iter: 20500, D: 0.0721, G:0.9979
```



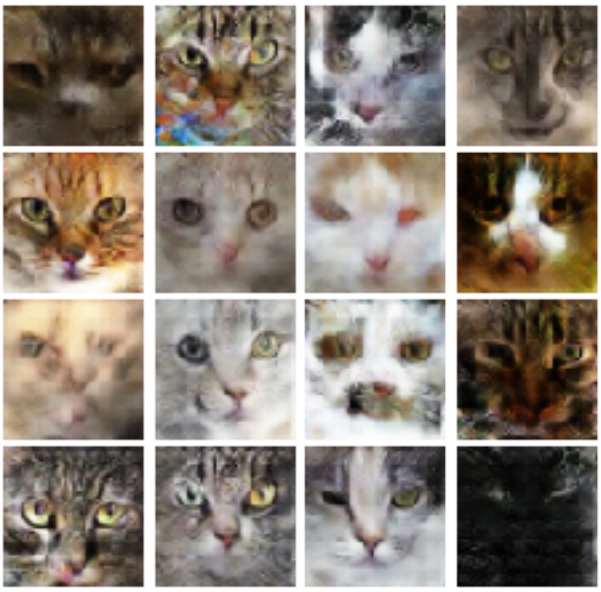
```
EPOCH:  43
Iter: 20750, D: 0.07577, G:0.7157
```

Iter: 21000, D: 0.1872, G:0.3597

EPOCH: 44
Iter: 21250, D: 0.0217, G:0.9086

```
Iter: 21500, D: 0.02811, G:1.044
```

```
EPOCH:  45
Iter: 21750, D: 0.03301, G:1.239
```



Iter: 22000, D: 0.0119, G:0.9814

EPOCH: 46
Iter: 22250, D: 0.3501, G:0.6238

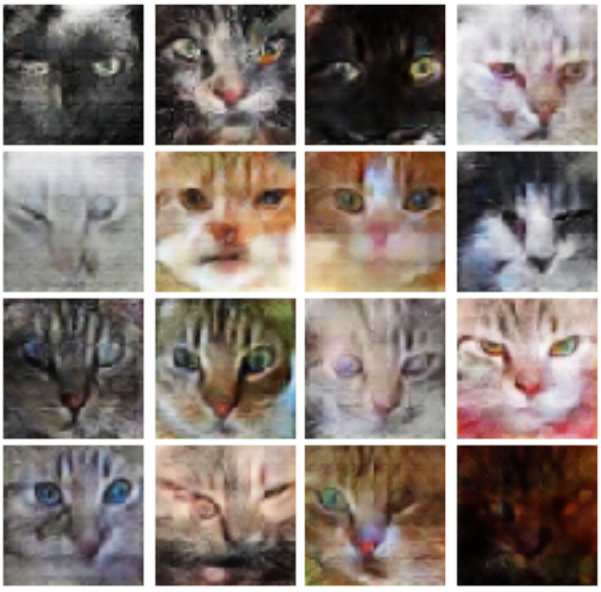
```
Iter: 22500, D: 0.00842, G:1.166
```

```
EPOCH:  47
Iter: 22750, D: 0.01202, G:1.051
```

```
Iter: 23000, D: 0.02476, G:0.5607
```

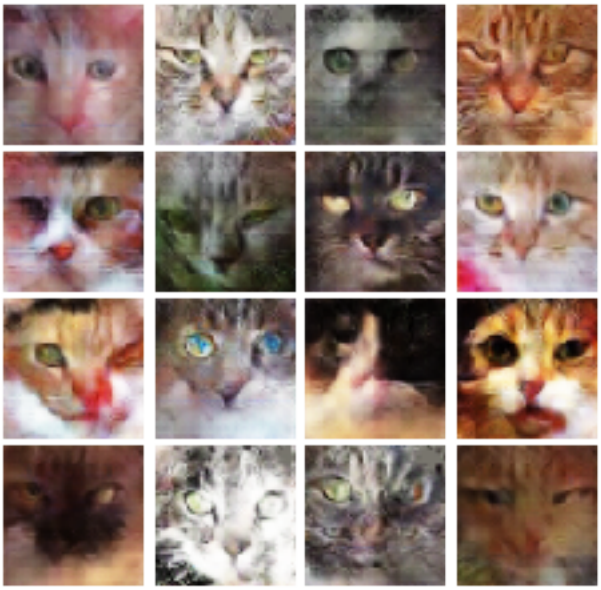
```
EPOCH:  48
Iter: 23250, D: 0.0204, G:1.52
```

Iter: 23500, D: 0.04886, G:1.107



EPOCH: 49
Iter: 23750, D: 0.02163, G:0.9345

```
Iter: 24000, D: 0.03091, G:1.112
```



```
EPOCH:  50
Iter: 24250, D: 0.01475, G:1.392
```

```
Iter: 24500, D: 0.01059, G:1.003
```

[ ]: