

April 30, 2024

1 Generative Adversarial Networks

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

```
[1]: import os  
# os.chdir("/Path/to/your/project/  
")  
path = os.getcwd()  
path
```

```
[1]: '/Confirm/path/to/your/project'
```

```
[2]: 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
```

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

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

```
[5]: device
```

```
[5]: device(type='cuda', index=0)
```

2 GAN loss functions

You'll need to implement two different loss functions. The first is the loss from the [original GAN paper](#). The next is the loss from [LS-GAN](#).

2.0.1 GAN loss

The generator loss is given by:

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

and the discriminator loss is:

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

Note that these equations are negated because we will be *minimizing* these losses.

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.

We will be averaging over the elements of the minibatch instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log (1 - D(G(z)))$.

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

2.0.2 Least Squares GAN loss

[Least Squares GAN](#) is an alternative to the original GAN loss function. For this part, all you need to do change the loss function and retrain the model.

Specifically, you'll implement equation (9) in the paper:

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

and the discriminator loss:

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

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

3 GAN model architecture

We'll be using the architecture from [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.

You can either use LeakyReLu throughout the discriminator (and a negative slope value of 0.2) or just use relu.

The discriminator will output a single score for each sample. The output of your discriminator should be a single value score corresponding to each input sample.

Generator:

Note: Here, you'll need to use transposed convolution (sometimes known as fractionally-strided convolution). 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 generator network will need to map the output values between -1 and 1 using a `tanh` nonlinearity. The output should be of size 64x64 with 3 channels for each sample (equal dimensions to the images from the dataset).

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

4 Data loading

Our dataset is pretty small, so in order to prevent overfitting, we need to perform data augmentation.

The easiest you can do is just use the RandomCrop and ColorJitter, but feel free to play around with other augmentations as well to see how it affects the performance of your model. See <https://pytorch.org/vision/stable/transforms.html>.

```
[9]: # # Define the transform for the training data.  
# transform = transforms.Compose([  
#     transforms.RandomResizedCrop(64, scale=(0.8, 1.0), ratio=(0.95, 1.05)), #  
#     # Randomly crop and resize the image to 64x64.  
#     transforms.RandomHorizontalFlip(), # Randomly flip the image  
#     #horizontally.  
#     transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1), #  
#     #Randomly change the brightness, contrast, and saturation of the image.  
#     transforms.ToTensor(), # Convert the image to a PyTorch tensor.  
#     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # Normalize the  
#     #image.  
# ])
```

```
[10]: print(os.path.exists('./cats')) # This should return True if the directory  
# exists
```

True

```
[11]: !ls
```

cats cats.zip gan part1.ipynb part1.pdf

```
[12]: batch_size = 32  
imsize = 64  
cat_root = './cats'  
  
cat_train = ImageFolder(root=cat_root, transform=transforms.Compose([  
    transforms.ToTensor(),  
    transforms.RandomResizedCrop(64, scale=(0.8, 1.0), ratio=(0.95, 1.05)), #  
    #Randomly crop and resize the image to 64x64.  
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1), #  
    #Randomly change the brightness, contrast, and saturation of the image.  
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # Normalize the  
    #image.  
  
    ## Use some some augmentations here  
]))  
  
cat_loader_train = DataLoader(cat_train, batch_size=batch_size, drop_last=True)
```

```
[13]: len(cat_train)
```

```
[13]: 15747
```

4.0.1 Visualize dataset

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



5 Training

```
[15]: NOISE_DIM = 100  
NUM_EPOCHS = 20  
learning_rate = 0.001
```

5.0.1 Train GAN

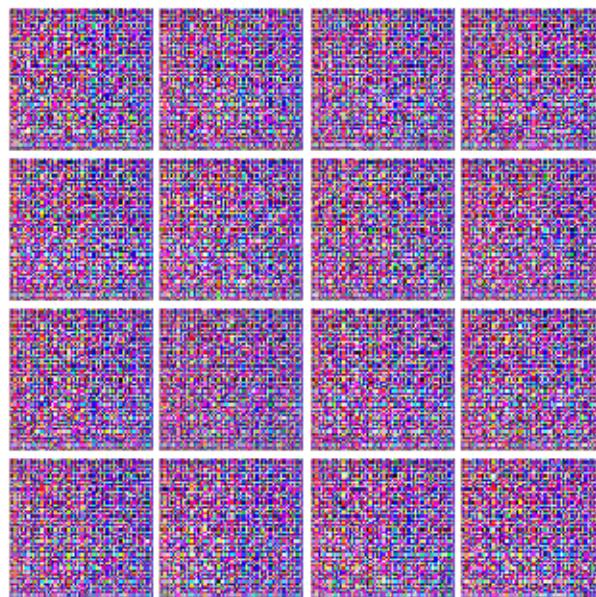
```
[16]: D = Discriminator().to(device)  
G = Generator(noise_dim=NOISE_DIM).to(device)
```

```
[17]: 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))
```

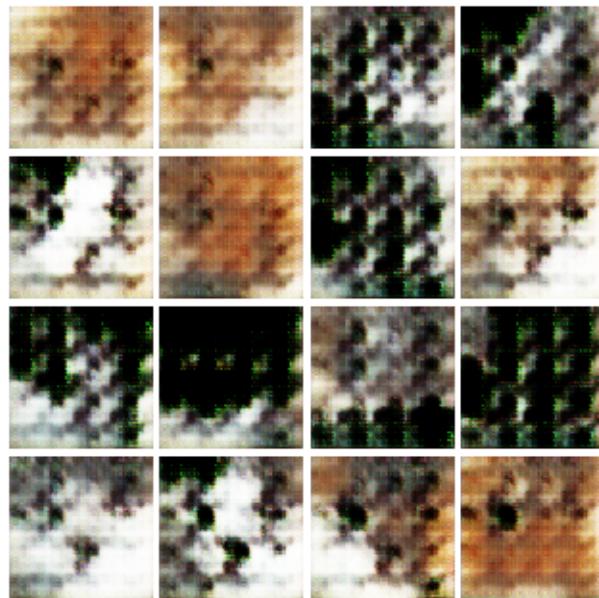
```
[18]: from gan.train import train
```

```
[19]: # 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: 0.7127, G:4.19

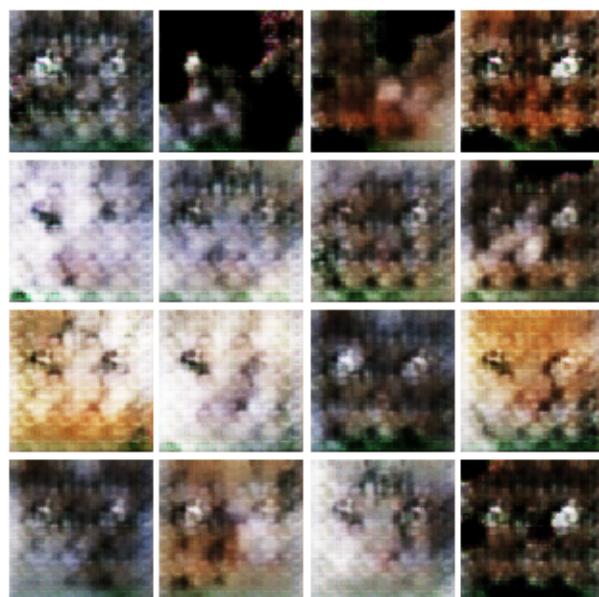


EPOCH: 2
Iter: 500, D: 0.9389, G:1.018



EPOCH: 3

Iter: 1000, D: 0.3405, G:2.709

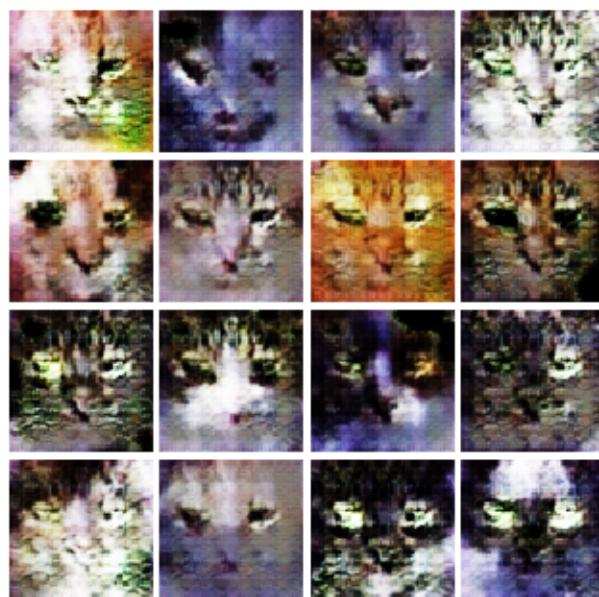


EPOCH: 4

Iter: 1500, D: 0.4914, G:2.447



EPOCH: 5
Iter: 2000, D: 0.6272, G:2.936



EPOCH: 6
Iter: 2500, D: 0.3728, G:2.41



EPOCH: 7
Iter: 3000, D: 0.4909, G:3.953



EPOCH: 8
Iter: 3500, D: 0.3892, G:1.244



EPOCH: 9

Iter: 4000, D: 0.3003, G:3.692



EPOCH: 10

Iter: 4500, D: 0.382, G:4.164



EPOCH: 11

Iter: 5000, D: 0.6974, G:7.925



EPOCH: 12

Iter: 5500, D: 0.06653, G:6.629



EPOCH: 13
Iter: 6000, D: 0.1146, G:3.97

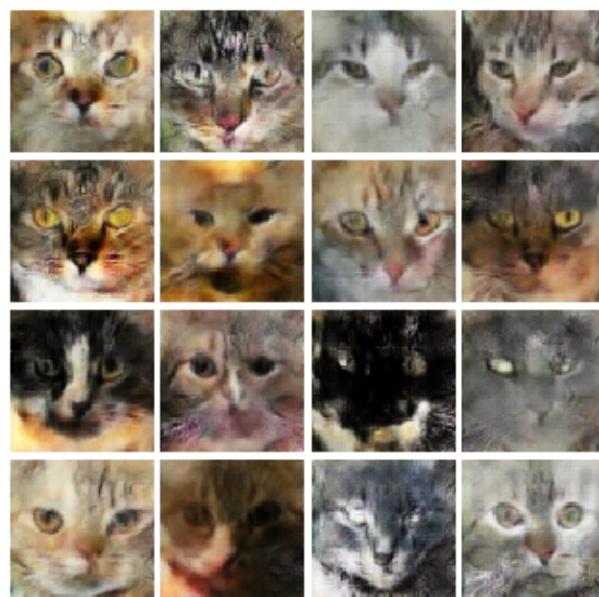


EPOCH: 14
Iter: 6500, D: 0.1337, G:5.532



EPOCH: 15

Iter: 7000, D: 0.04765, G:5.568



EPOCH: 16

Iter: 7500, D: 0.06044, G:5.518



EPOCH: 17

Iter: 8000, D: 0.04569, G:3.245



EPOCH: 18

Iter: 8500, D: 0.5909, G:8.673



EPOCH: 19

Iter: 9000, D: 0.06816, G:4.749



EPOCH: 20

Iter: 9500, D: 0.1145, G:6.261



5.0.2 Train LS-GAN

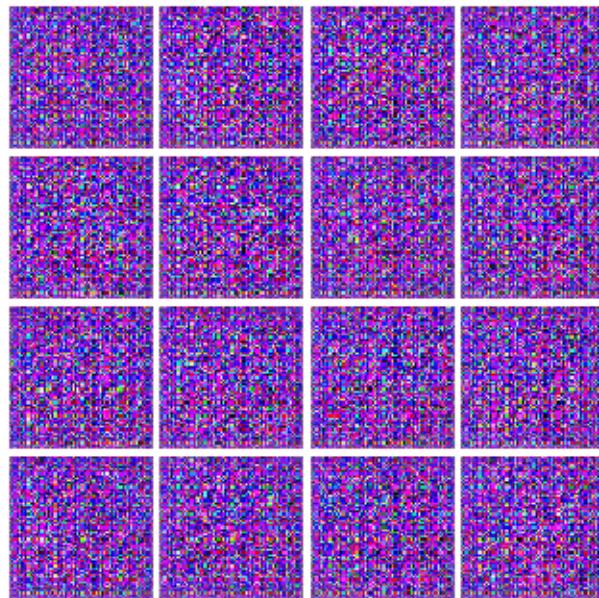
```
[21]: NOISE_DIM = 100
      NUM_EPOCHS = 20
      learning_rate = 0.001
```

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

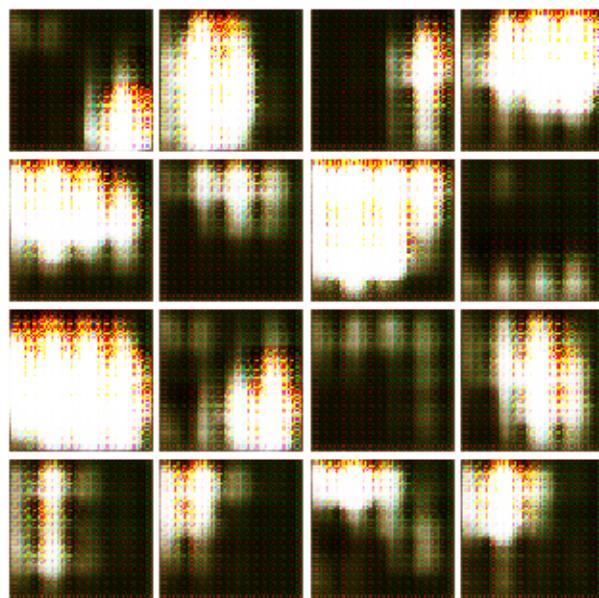
```
[26]: D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.999))
      G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.999))
```

```
[27]: # 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: 0.2604, G: 0.9795

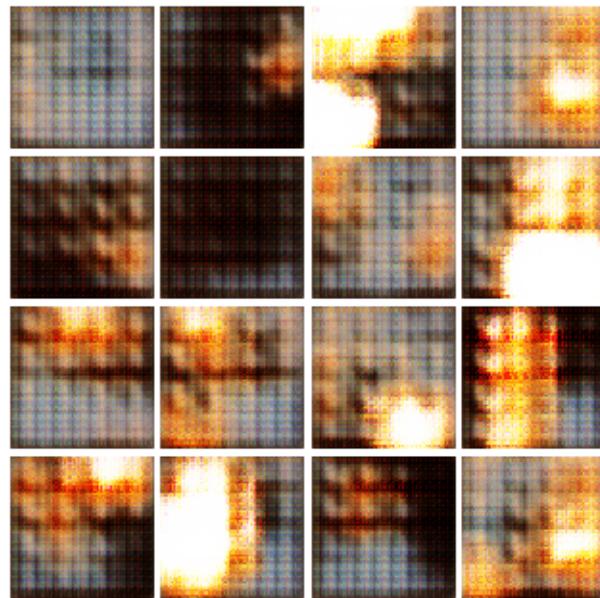


Iter: 250, D: 0.003747, G:0.9884

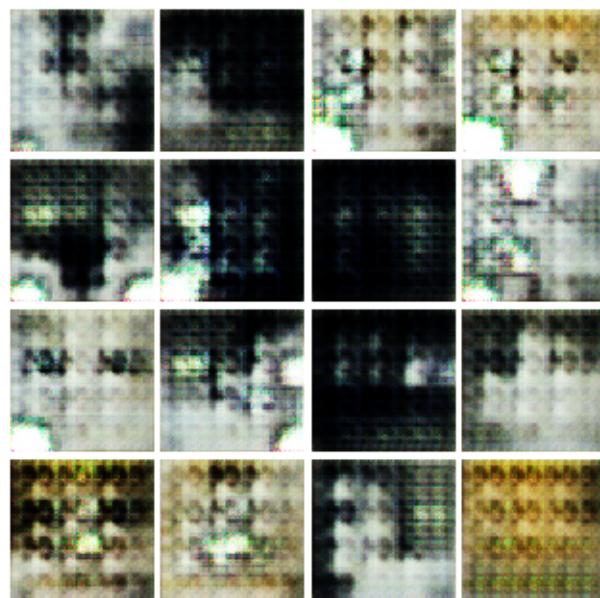


EPOCH: 2

Iter: 500, D: 0.3438, G:0.7185



Iter: 750, D: 0.3125, G:0.4301



EPOCH: 3

Iter: 1000, D: 0.195, G:0.7984

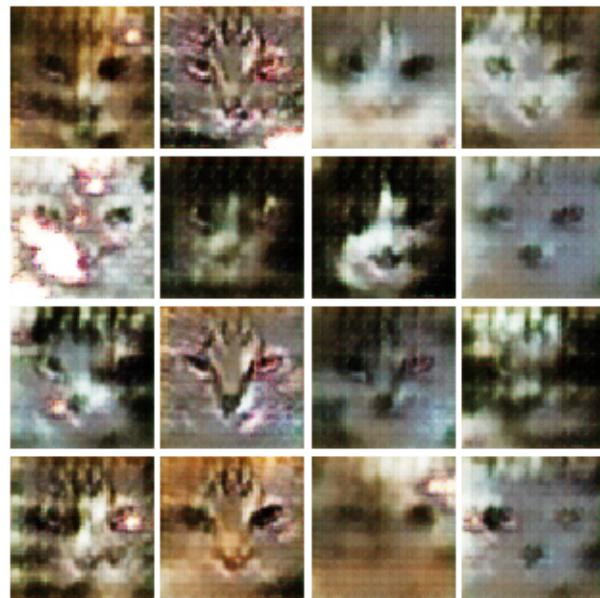


Iter: 1250, D: 0.2617, G:0.6795



EPOCH: 4

Iter: 1500, D: 0.1192, G:0.8129



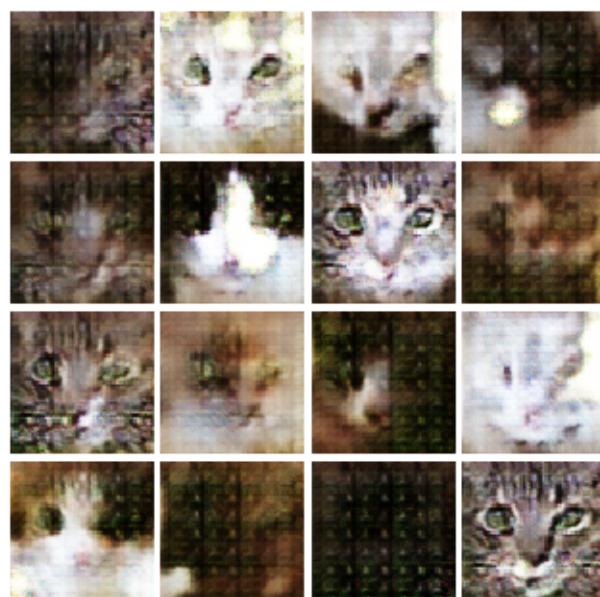
Iter: 1750, D: 0.1228, G:0.8671



EPOCH: 5
Iter: 2000, D: 0.1786, G:0.6939

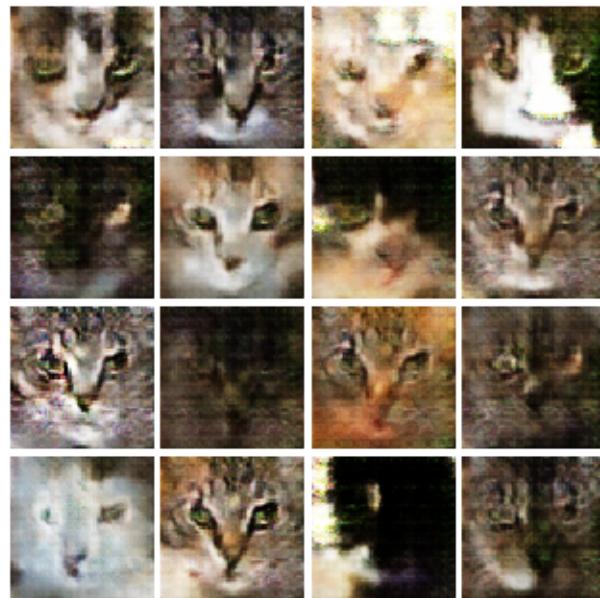


Iter: 2250, D: 0.09538, G:0.916



EPOCH: 6

Iter: 2500, D: 0.1319, G:0.8149



Iter: 2750, D: 0.1564, G:0.8501

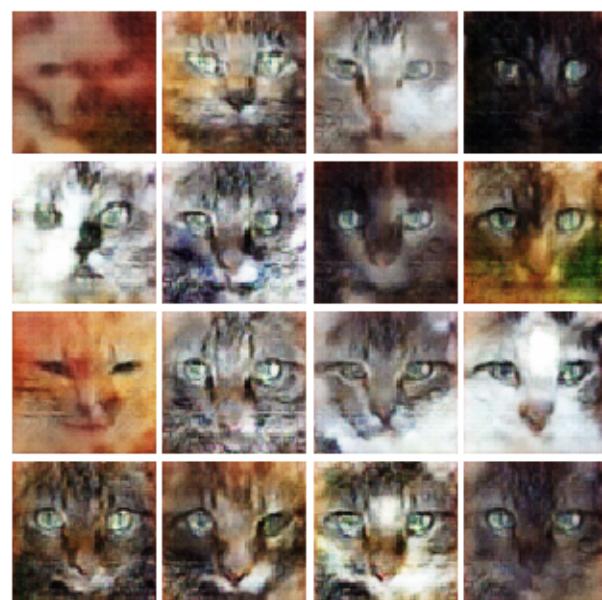


EPOCH: 7

Iter: 3000, D: 0.1556, G:0.7582

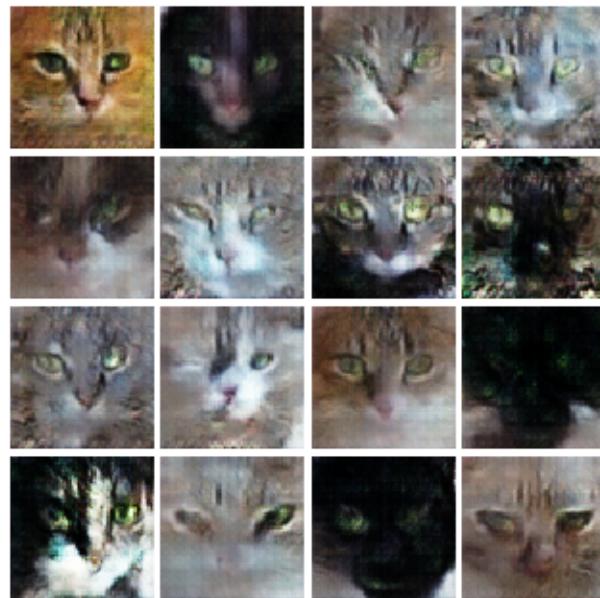


Iter: 3250, D: 0.1785, G:0.3048



EPOCH: 8

Iter: 3500, D: 0.1746, G:0.6137



Iter: 3750, D: 0.1475, G: 0.6294



EPOCH: 9

Iter: 4000, D: 0.1261, G: 0.7095



Iter: 4250, D: 0.1455, G:0.4997



EPOCH: 10
Iter: 4500, D: 0.1758, G:0.3328



Iter: 4750, D: 0.2195, G:0.6301



EPOCH: 11
Iter: 5000, D: 0.1994, G:0.5739



Iter: 5250, D: 0.1664, G:0.7533



EPOCH: 12
Iter: 5500, D: 0.1842, G:0.7227



Iter: 5750, D: 0.2389, G:0.603



EPOCH: 13
Iter: 6000, D: 0.1801, G:0.7105



Iter: 6250, D: 0.1662, G:0.4738



EPOCH: 14
Iter: 6500, D: 0.1508, G:0.5974



Iter: 6750, D: 0.1482, G:0.7407



EPOCH: 15
Iter: 7000, D: 0.1864, G:0.4463



Iter: 7250, D: 0.1473, G:0.5582



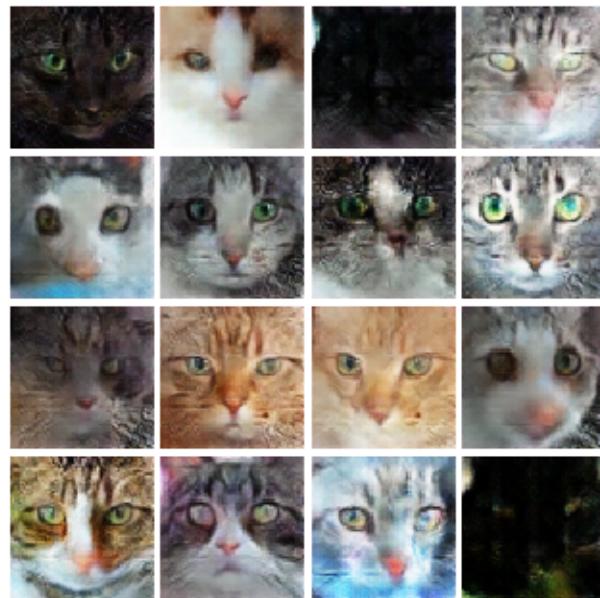
EPOCH: 16
Iter: 7500, D: 0.2664, G:0.7539



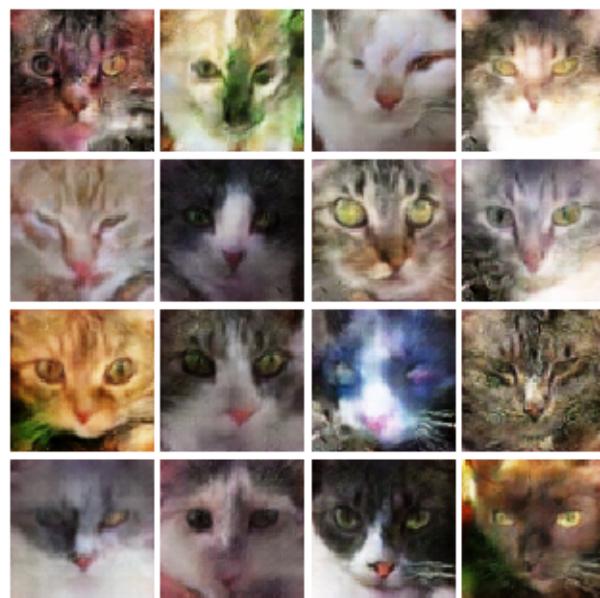
Iter: 7750, D: 0.1045, G:0.7488



EPOCH: 17
Iter: 8000, D: 0.1277, G:0.5346



Iter: 8250, D: 0.1305, G: 0.7865



EPOCH: 18
Iter: 8500, D: 0.09923, G: 0.7142



Iter: 8750, D: 0.1969, G:0.9193



EPOCH: 19

Iter: 9000, D: 0.01865, G:0.8821



Iter: 9250, D: 0.1067, G: 0.4685



EPOCH: 20
Iter: 9500, D: 0.06706, G: 0.6573



Iter: 9750, D: 0.1558, G: 0.9762



