(Optional) Colab Setup

If you aren't using Colab, you can delete the following code cell. This is just to help students with mounting to Google Drive to access the other .py files and downloading the data, which is a little trickier on Colab than on your local machine using Jupyter.

```
# you will be prompted with a window asking to grant permissions
from google.colab import drive
drive.mount("/content/drive")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# fill in the path in your Google Drive in the string below. Note: do not escape slashes or spaces
import os
datadir = "/content/drive/MyDrive/CS444/assignment4"
if not os.path.exists(datadir):
  !ln -s "/content/drive/MyDrive/path_to/assignment4" $datadir # TODO: Fill your Assignment 4 path
os.chdir(datadir)
!pwd
```

```
/content/drive/MyDrive/CS444/assignment4
```

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.

```
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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
from gan.train import train
```

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

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.

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:

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

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.

```
1 from gan.losses import discriminator_loss, generator_loss
```

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:

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

and the discriminator loss:

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

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`).

```
1 from gan.losses import ls_discriminator_loss, ls_generator_loss
```

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).

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

Data loading

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

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

```
#!sh download_cat.sh
```

```
batch_size = 64
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)
```

Visualize dataset

```
from gan.utils import show_images

try:
  imgs = next(iter(cat_loader_train))[0].numpy().squeeze()
except:
  imgs = cat_loader_train.__iter__().next()[0].numpy().squeeze()

show_images(imgs, color=True)
```

Training

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

```
NOISE_DIM = 100
NUM_EPOCHS = 50
learning_rate = 0.001
```

Train GAN

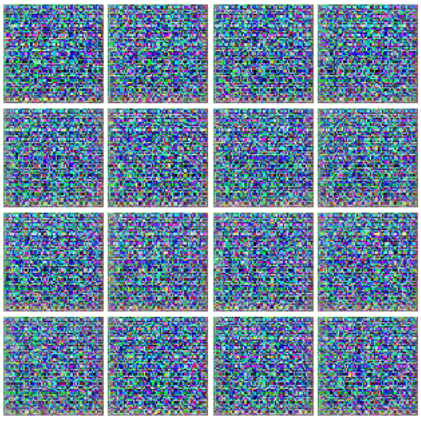
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

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

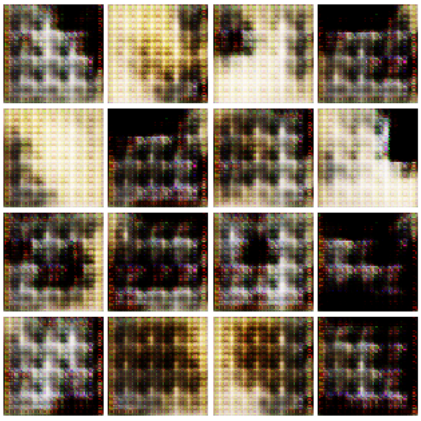
```
# original gan
train(D, G, D_optimizer, G_optimizer, discriminator_loss,
          generator_loss, num_epochs=NUM_EPOCHS, show_every=250,
          batch_size=batch_size, train_loader=cat_loader_train, device=device)

```

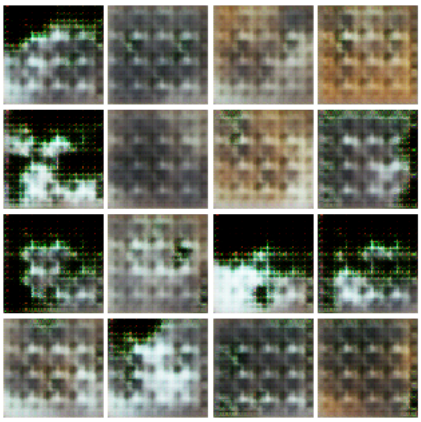
EPOCH: 1
Iter: 0, D: 0.7079, G:6.138



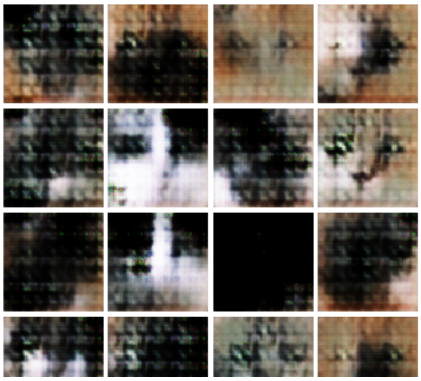
EPOCH: 2
Iter: 250, D: 0.7495, G:1.464

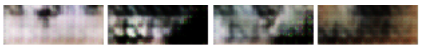


EPOCH: 3
Iter: 500, D: 0.5613, G:3.031



EPOCH: 4
Iter: 750, D: 0.4069, G:1.824



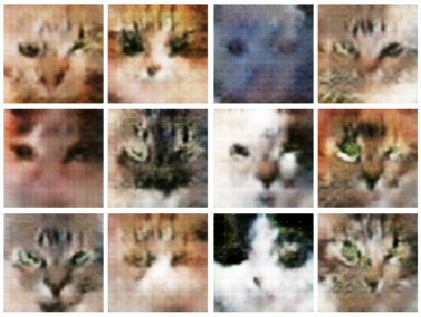


EPOCH: 5
Iter: 1000, D: 0.6298, G:1.781

EPOCH: 6
Iter: 1250, D: 0.5896, G:2.079

EPOCH: 7
Iter: 1500, D: 0.5268, G:3.207

EPOCH: 8
Iter: 1750, D: 0.6613, G:0.8647

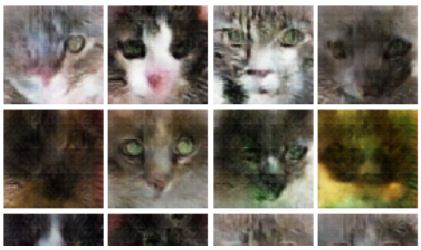


EPOCH: 9
Iter: 2000, D: 0.5169, G:3.233

EPOCH: 10
Iter: 2250, D: 0.3921, G:2.485

EPOCH: 11
Iter: 2500, D: 0.763, G:6.523

EPOCH: 12
Iter: 2750, D: 0.2897, G:3.461

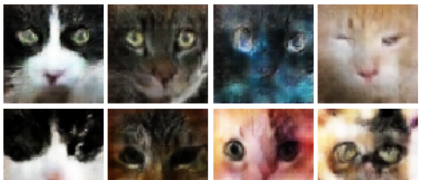


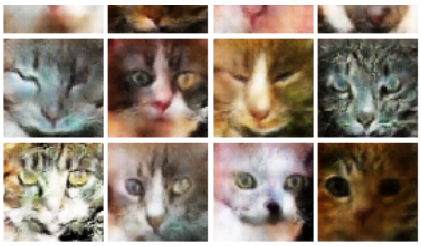
EPOCH: 13
Iter: 3000, D: 0.5748, G:4.416

EPOCH: 14
Iter: 3250, D: 0.3944, G:2.004

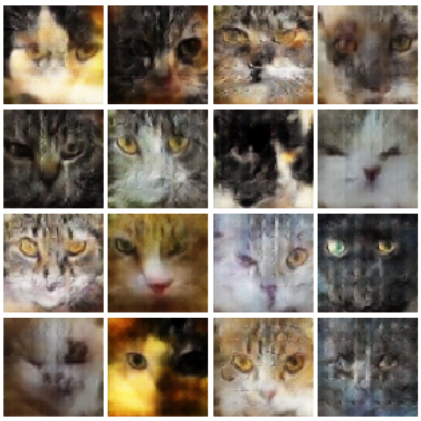
EPOCH: 15
Iter: 3500, D: 0.3888, G:2.143

EPOCH: 16
Iter: 3750, D: 0.9383, G:1.232



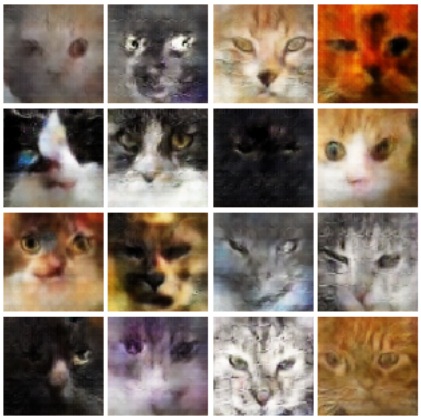


EPOCH: 17
Iter: 4000, D: 0.3382, G:2.606

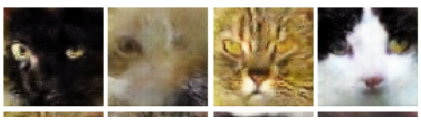


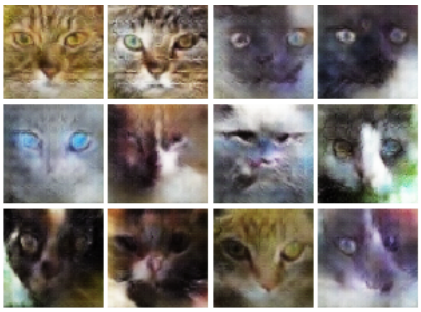
EPOCH: 18
Iter: 4250, D: 0.6438, G:1.417

EPOCH: 19
Iter: 4500, D: 0.2513, G:2.922



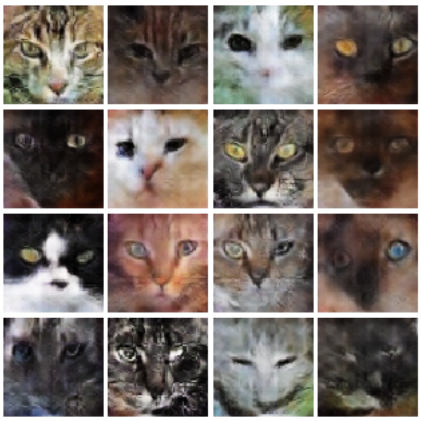
EPOCH: 20
Iter: 4750, D: 0.3874, G:3.204



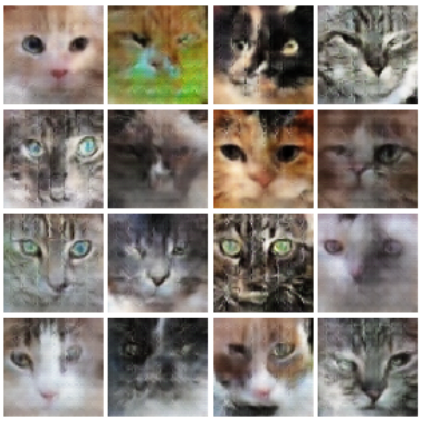


EPOCH: 21
Iter: 5000, D: 0.7917, G:2.148

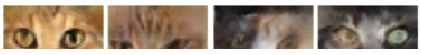
EPOCH: 22
Iter: 5250, D: 0.8108, G:0.8221



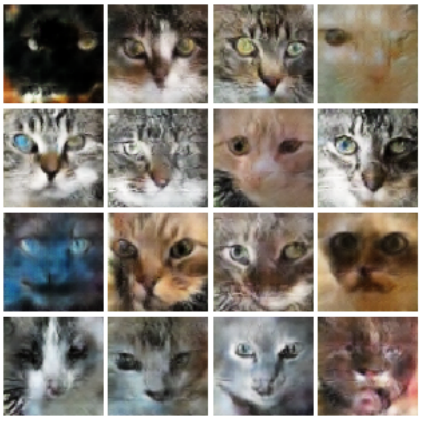
EPOCH: 23
Iter: 5500, D: 0.1771, G:5.317



EPOCH: 24
Iter: 5750, D: 0.2526, G:1.512



EPOCH: 25
Iter: 6000, D: 0.1022, G:6.074



EPOCH: 26
Iter: 6250, D: 0.3238, G:6.168

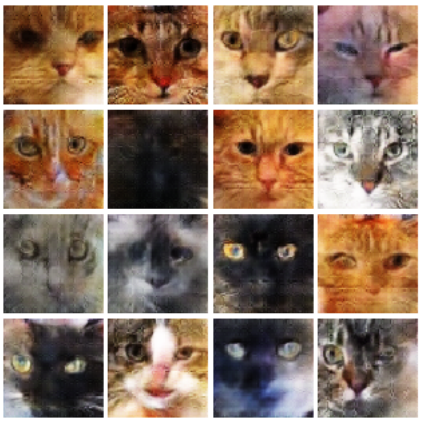
EPOCH: 27
Iter: 6500, D: 0.3592, G:7.43

EPOCH: 28
Iter: 6750, D: 0.1865, G:3.155

EPOCH: 29
Iter: 7000, D: 0.09751, G:5.333

EPOCH: 30
Iter: 7250, D: 0.131, G:5.118

EPOCH: 31
Iter: 7500, D: 0.3644, G:8.199

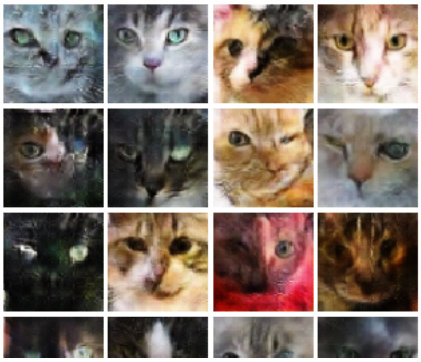


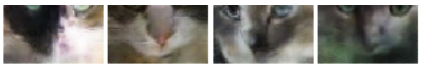
EPOCH: 32
Iter: 7750, D: 0.08055, G:5.115

EPOCH: 33
Iter: 8000, D: 0.4334, G:2.318

EPOCH: 34
Iter: 8250, D: 0.956, G:8.119

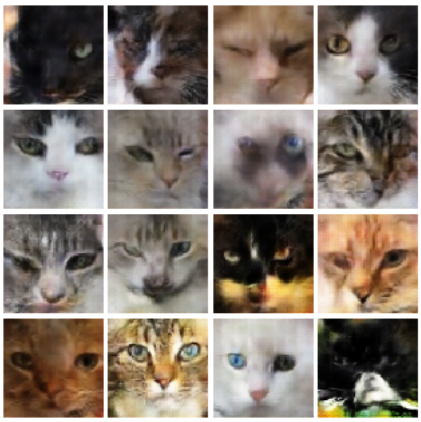
EPOCH: 35
Iter: 8500, D: 0.1908, G:4.441





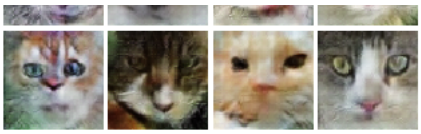
EPOCH: 36
Iter: 8750, D: 0.2224, G:2.082

EPOCH: 37
Iter: 9000, D: 0.1188, G:5.701



EPOCH: 38
Iter: 9250, D: 0.9591, G:10.47

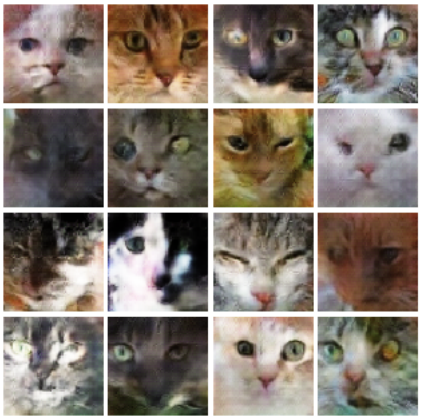
EPOCH: 39
Iter: 9500, D: 0.1058, G:5.951



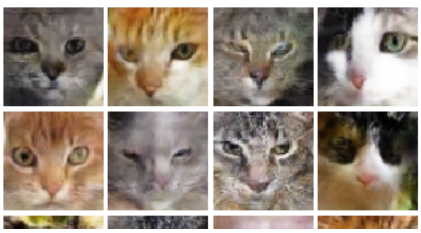
EPOCH: 40
Iter: 9750, D: 0.04171, G:4.983

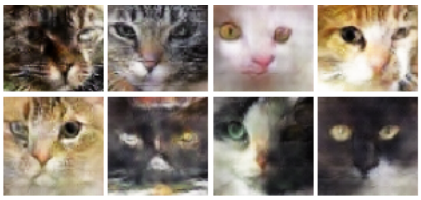
EPOCH: 41
Iter: 10000, D: 0.06327, G:5.347

EPOCH: 42
Iter: 10250, D: 0.1544, G:5.105

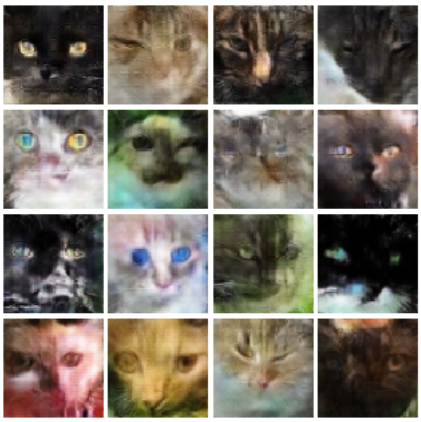


EPOCH: 43
Iter: 10500, D: 0.4622, G:4.331

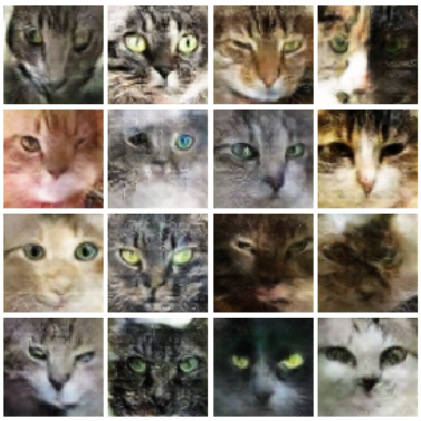




EPOCH: 44
Iter: 10750, D: 0.1333, G:4.409

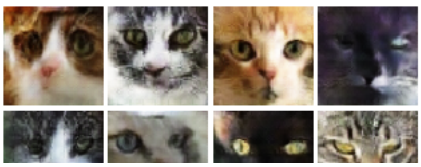


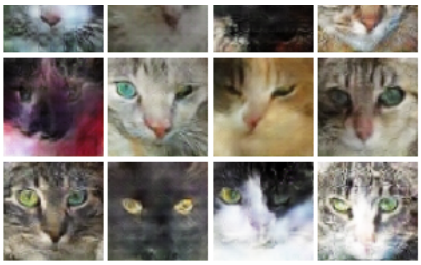
EPOCH: 45
Iter: 11000, D: 0.5665, G:12.09



EPOCH: 46
Iter: 11250, D: 1.263, G:16.85

EPOCH: 47
Iter: 11500, D: 0.09644, G:5.543

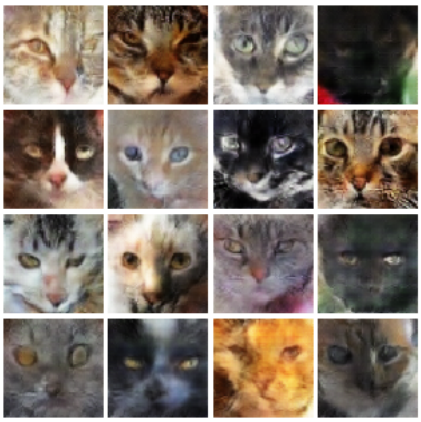




EPOCH: 48
Iter: 11750, D: 0.2328, G:3.167

EPOCH: 49
Iter: 12000, D: 0.06284, G:5.407

EPOCH: 50
Iter: 12250, D: 0.8496, G:0.9974



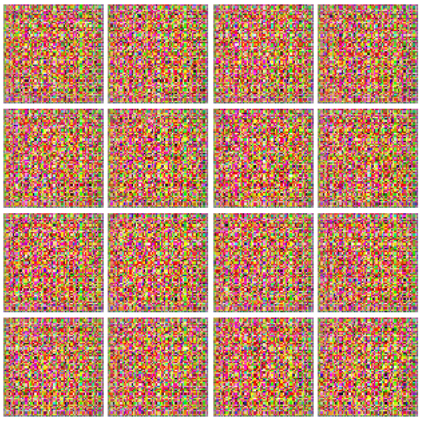
Train LS-GAN

```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

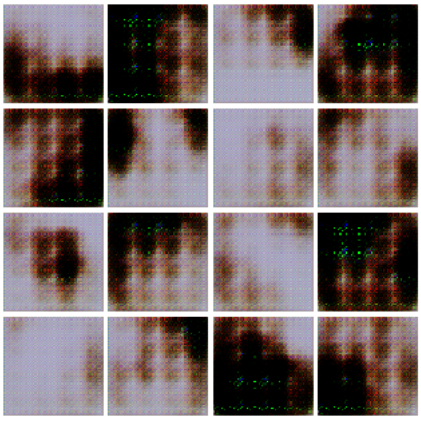
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.999))
```

```
  # 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)
```

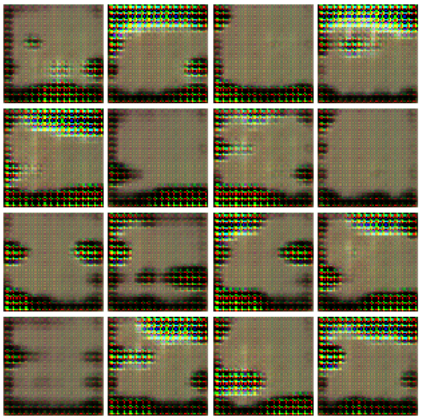
EPOCH: 1
Iter: 0, D: 0.658, G:37.58



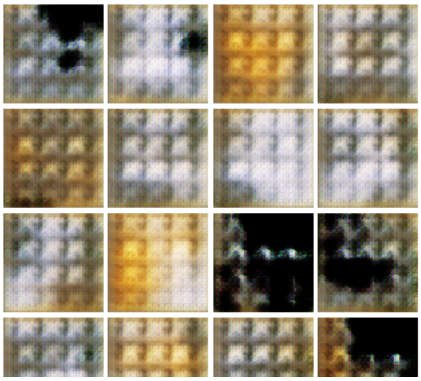
EPOCH: 2
Iter: 250, D: 0.1837, G:0.5049

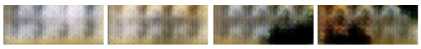


EPOCH: 3
Iter: 500, D: 3.542, G:2.323

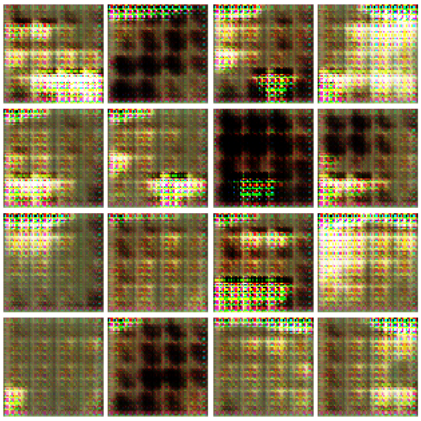


EPOCH: 4
Iter: 750, D: 0.172, G:0.1387

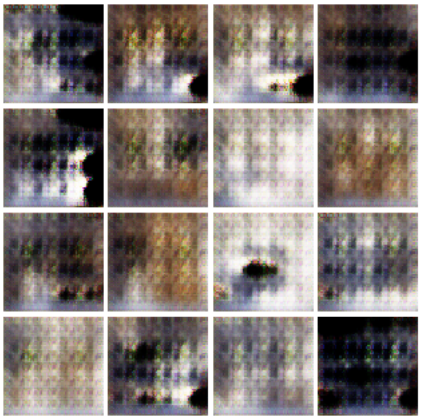




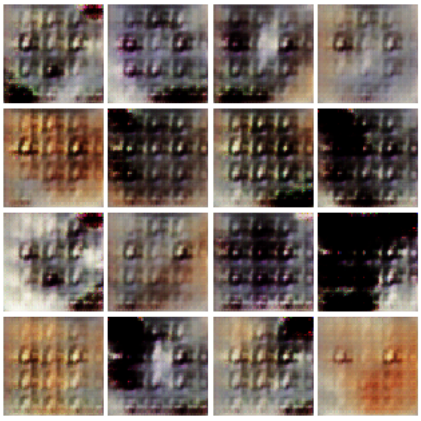
EPOCH: 5
Iter: 1000, D: 0.1274, G:0.2906



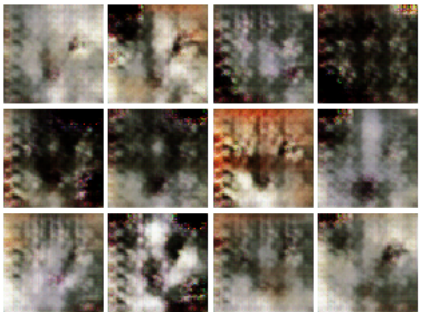
EPOCH: 6
Iter: 1250, D: 0.2233, G:0.2264



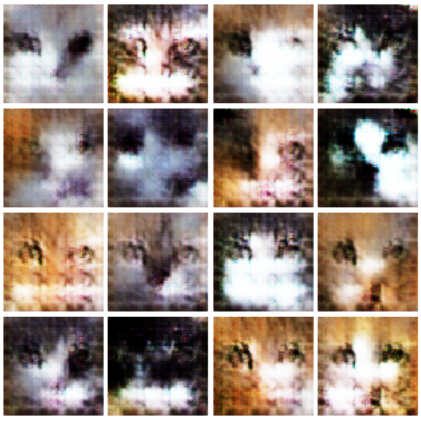
EPOCH: 7
Iter: 1500, D: 0.154, G:0.4346



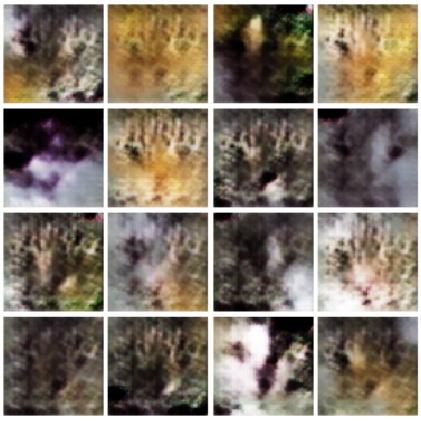
EPOCH: 8
Iter: 1750, D: 0.2533, G:0.2115



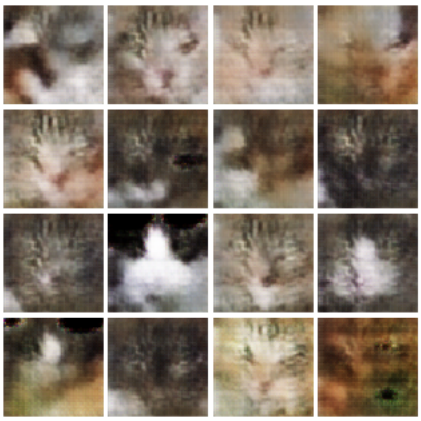
EPOCH: 9
Iter: 2000, D: 0.1987, G:0.1856



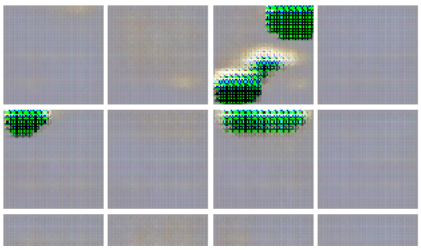
EPOCH: 10
Iter: 2250, D: 0.3094, G:0.5547

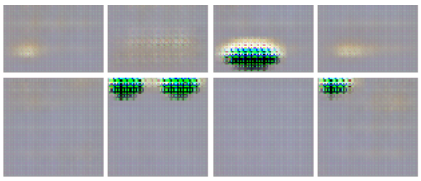


EPOCH: 11
Iter: 2500, D: 0.2129, G:0.2157

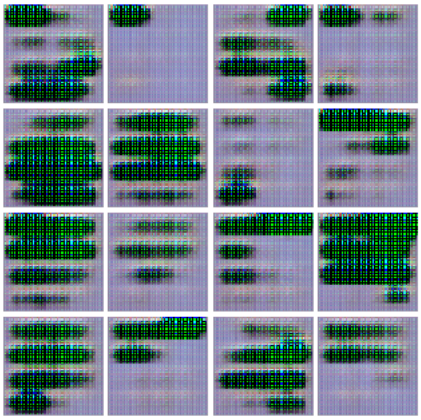


EPOCH: 12
Iter: 2750, D: 0.2046, G:0.2922

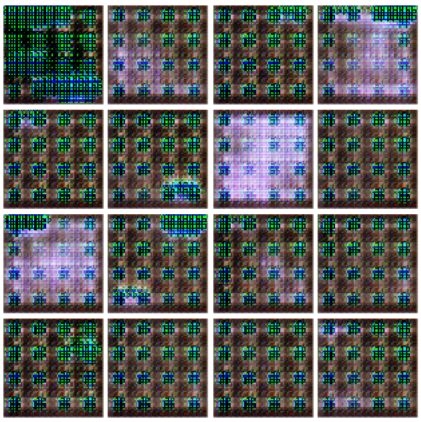




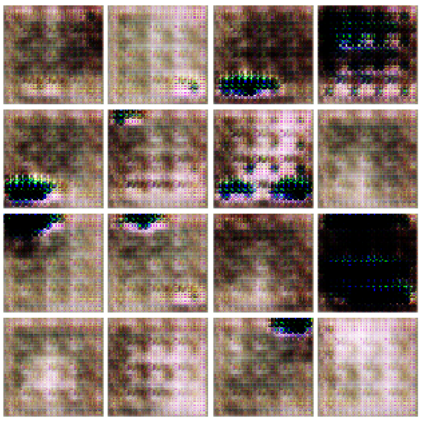
EPOCH: 13
Iter: 3000, D: 0.1734, G:0.1995



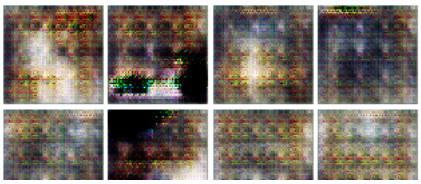
EPOCH: 14
Iter: 3250, D: 0.1306, G:0.2471

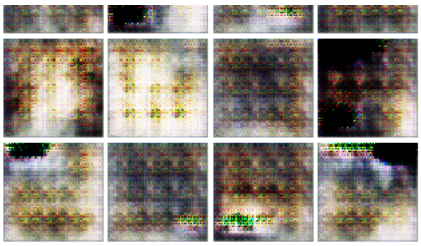


EPOCH: 15
Iter: 3500, D: 0.2343, G:0.1571

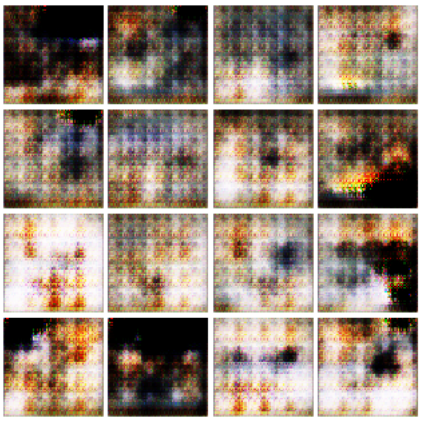


EPOCH: 16
Iter: 3750, D: 0.224, G:0.1262

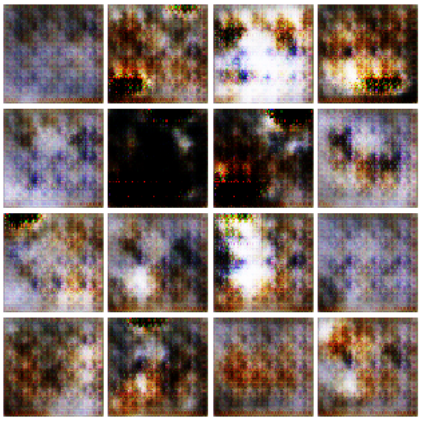




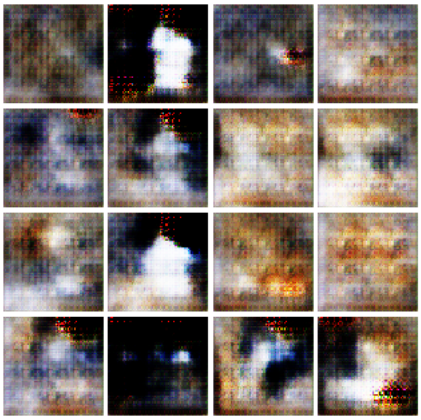
EPOCH: 17
Iter: 4000, D: 0.2781, G:0.1489



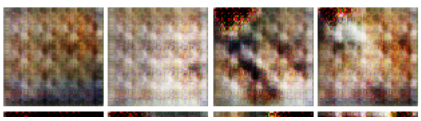
EPOCH: 18
Iter: 4250, D: 0.2575, G:0.1378

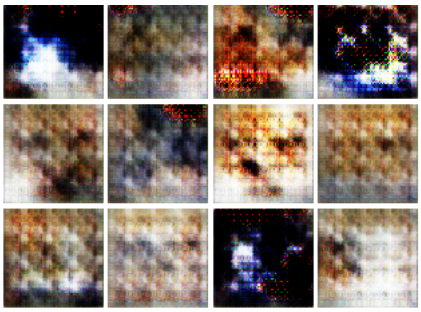


EPOCH: 19
Iter: 4500, D: 0.2855, G:0.1724

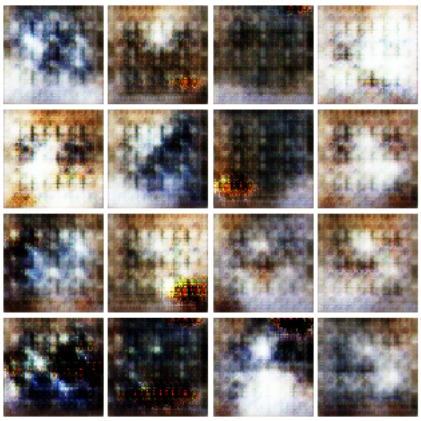


EPOCH: 20
Iter: 4750, D: 0.2466, G:0.143

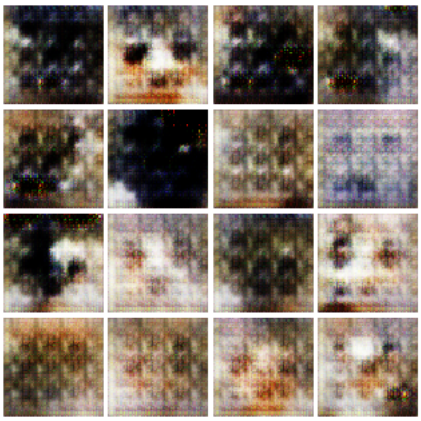




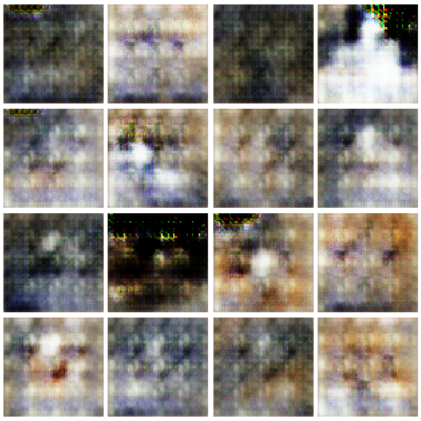
EPOCH: 21
Iter: 5000, D: 0.3408, G:0.2401



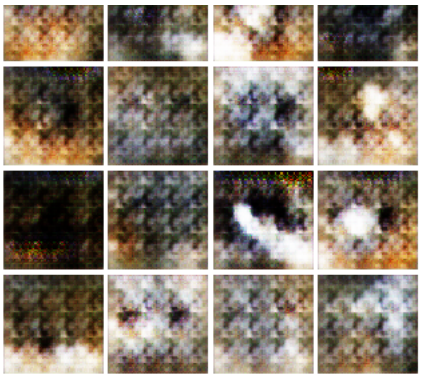
EPOCH: 22
Iter: 5250, D: 0.3061, G:0.2399



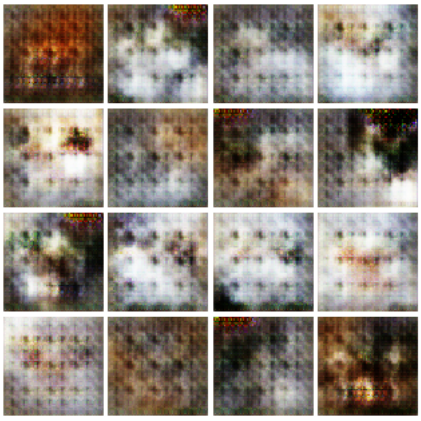
EPOCH: 23
Iter: 5500, D: 0.2643, G:0.1174



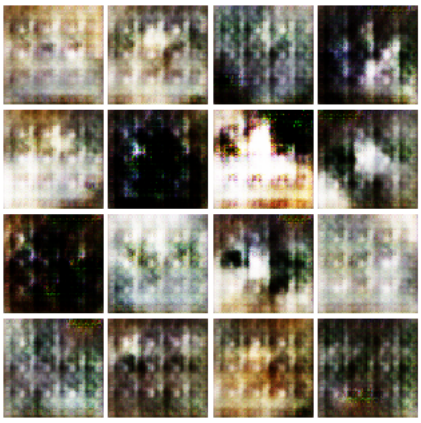
EPOCH: 24
Iter: 5750, D: 0.2575, G:0.1795



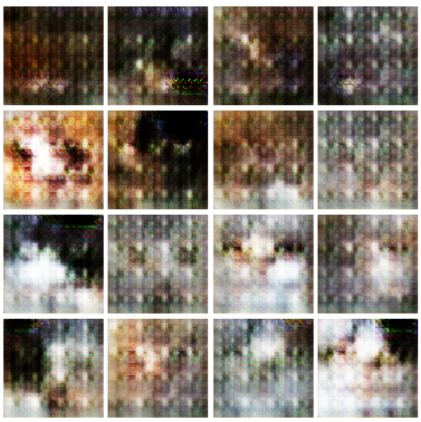
EPOCH: 25
Iter: 6000, D: 0.281, G:0.1825



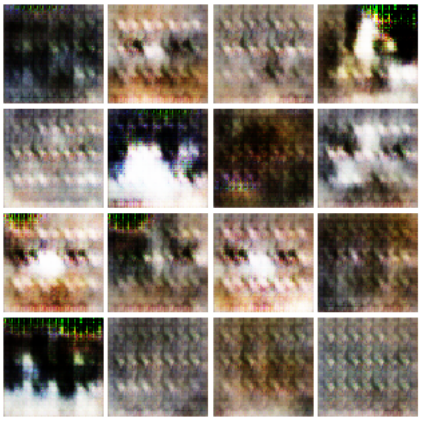
EPOCH: 26
Iter: 6250, D: 0.4961, G:0.3849



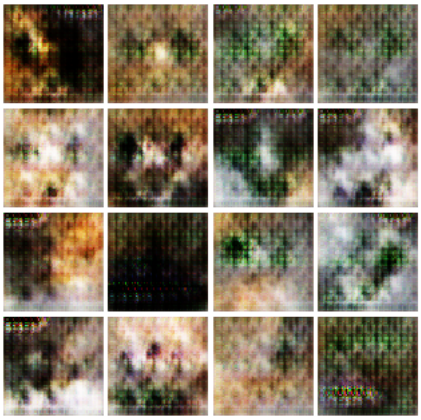
EPOCH: 27
Iter: 6500, D: 0.5924, G:0.2854



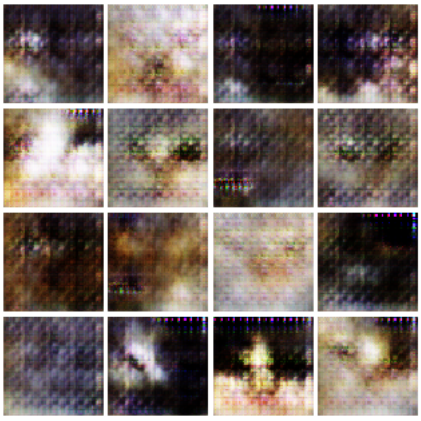
EPOCH: 28
Iter: 6750, D: 0.2474, G:0.1602



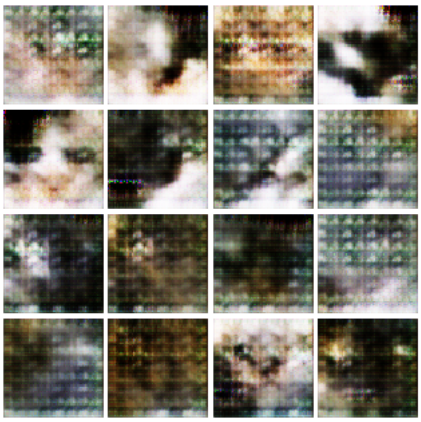
EPOCH: 29
Iter: 7000, D: 0.2561, G:0.1579



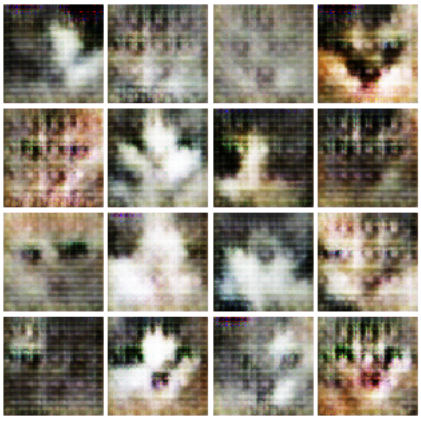
EPOCH: 30
Iter: 7250, D: 0.3258, G:0.2041



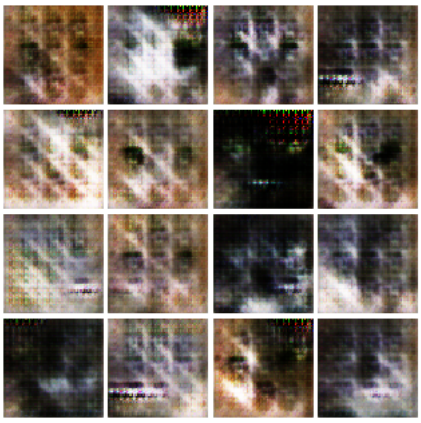
EPOCH: 31
Iter: 7500, D: 0.2492, G:0.2434



EPOCH: 32
Iter: 7750, D: 0.2388, G:0.1699

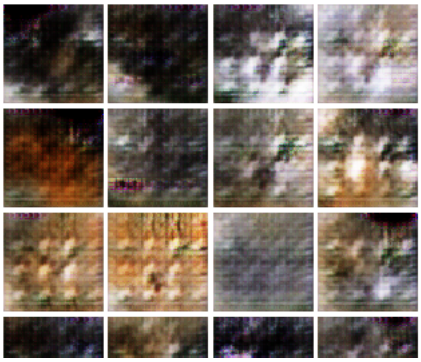


EPOCH: 33
Iter: 8000, D: 0.2549, G:0.2432

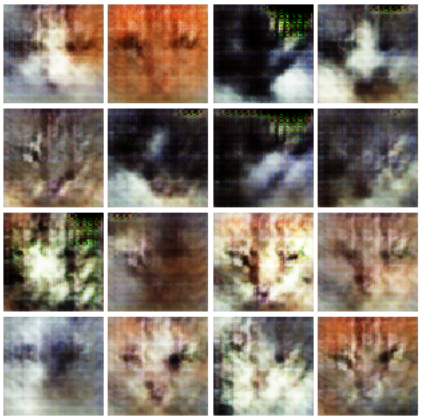


EPOCH: 34
Iter: 8250, D: 0.2484, G:0.1507

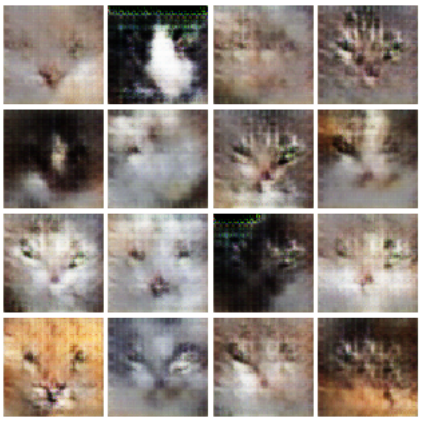
EPOCH: 35
Iter: 8500, D: 0.2662, G:0.17



EPOCH: 36
Iter: 8750, D: 0.2502, G:0.1812

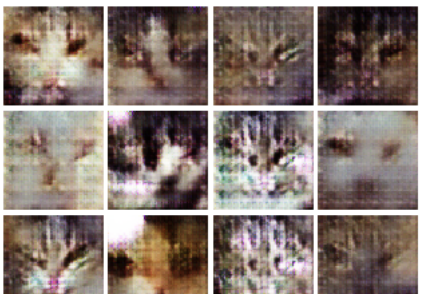


EPOCH: 37
Iter: 9000, D: 0.2415, G:0.1559



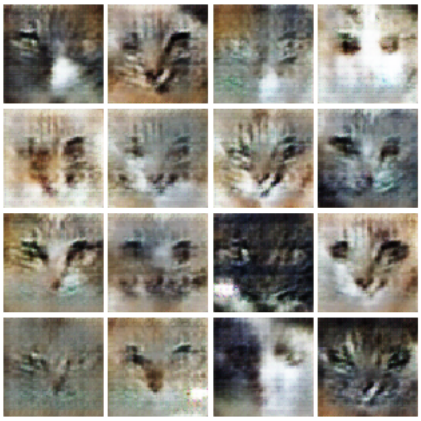
EPOCH: 38
Iter: 9250, D: 0.2814, G:0.2005

EPOCH: 39
Iter: 9500, D: 0.2627, G:0.2152

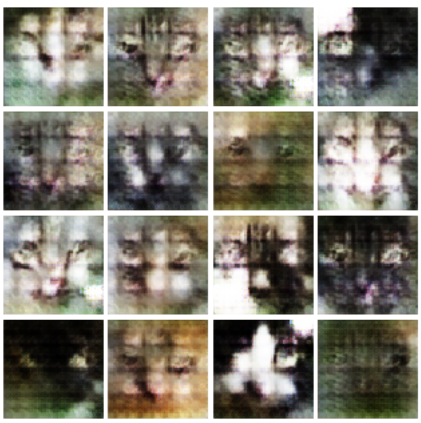


EPOCH: 40
Iter: 9750, D: 0.2402, G:0.2164

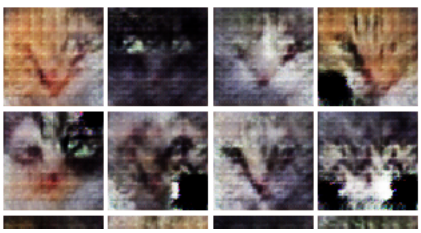
EPOCH: 41
Iter: 10000, D: 0.2854, G:0.1277

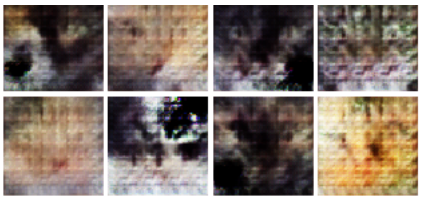


EPOCH: 42
Iter: 10250, D: 0.2473, G:0.2101



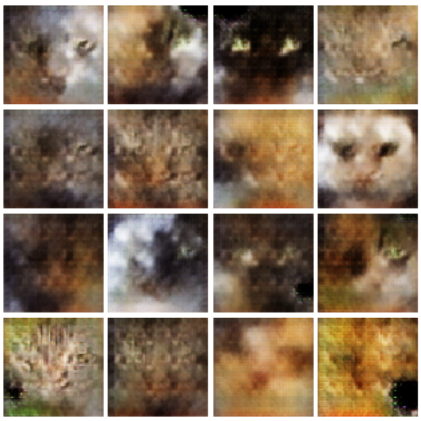
EPOCH: 43
Iter: 10500, D: 0.368, G:0.1819





EPOCH: 44
Iter: 10750, D: 0.2888, G:0.2008

EPOCH: 45
Iter: 11000, D: 0.2471, G:0.248



EPOCH: 46
Iter: 11250, D: 0.2249, G:0.2138