# Connecting with google drive

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

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

```
datadir = '/content/drive/My Drive/CS444/assignment4_materials/assignment4_materials/'
import os
if os.path.exists(datadir):
  os.chdir(datadir)
else:
  print(datadir, 'path not found')
```

```
!pwd
```

```
/content/drive/My Drive/CS444/assignment4_materials/assignment4_materials
```

```
# run this cell only once
# !unzip cats.zip
```

# 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 torch.nn as nn
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
```

```
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.

```
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`).

```
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.

```
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)
```

## Visualize dataset

```
from gan.utils import show_images

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

```
/usr/local/lib/python3.9/dist-packages/torchvision/transforms/functional.py:1603:
  warnings.warn(
```

# Training

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

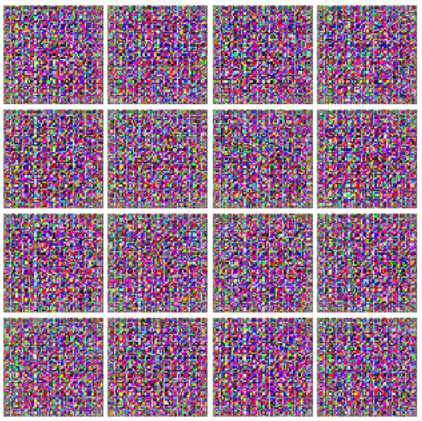
```
NOISE_DIM = 100
NUM_EPOCHS = 50
learning_rate = 1e-3
```

## Train GAN

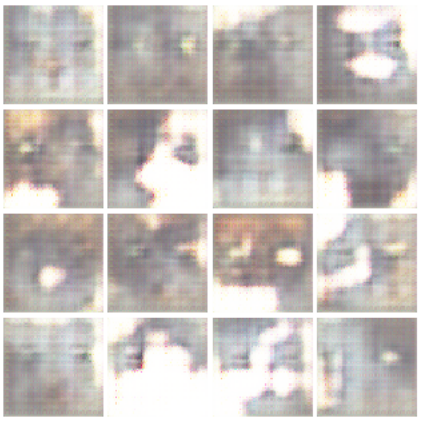
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)


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)
```

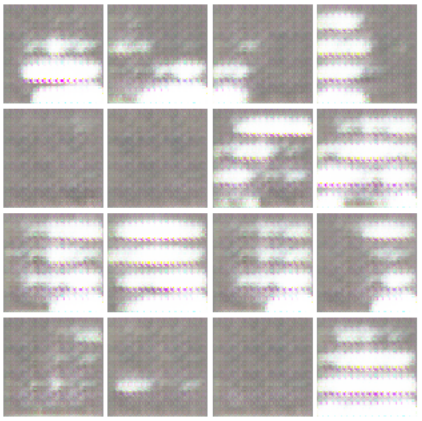
```
EPOCH:  1
/usr/local/lib/python3.9/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter
  warnings.warn(
Iter: 0, D: 1.404, G:3.864
```



```
Iter: 250, D: 1.82, G:0.5296
```

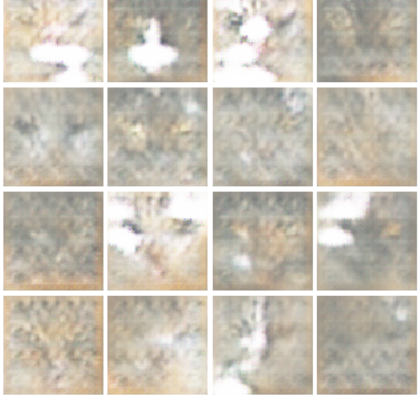


```
EPOCH:  2
Iter: 500, D: 0.9001, G:1.798
```

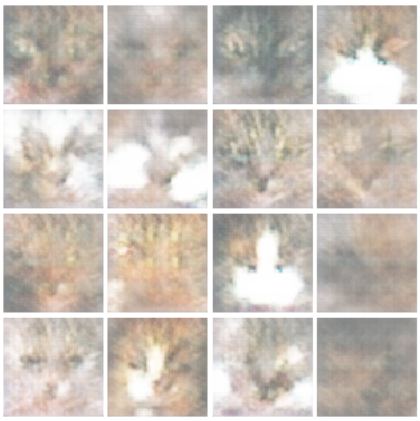


```
Iter: 750, D: 1.015, G:1.361
```

```
EPOCH:  3
Iter: 1000, D: 0.7664, G:3.154
```



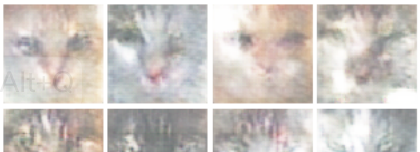
Iter: 1250, D: 0.8841, G:1.086

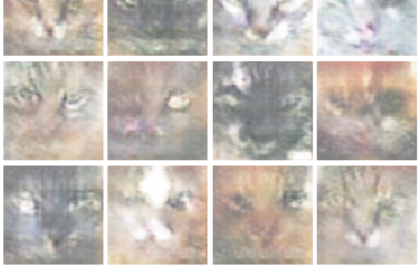


EPOCH:  4
Iter: 1500, D: 0.8442, G:2.22

Iter: 1750, D: 0.8579, G:1.805

EPOCH:  5
Iter: 2000, D: 0.8601, G:1.994



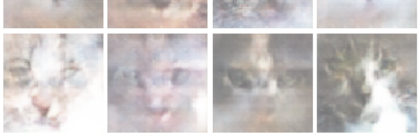


Iter: 2250, D: 0.349, G:2.903

EPOCH:  6
Iter: 2500, D: 1.063, G:3.722

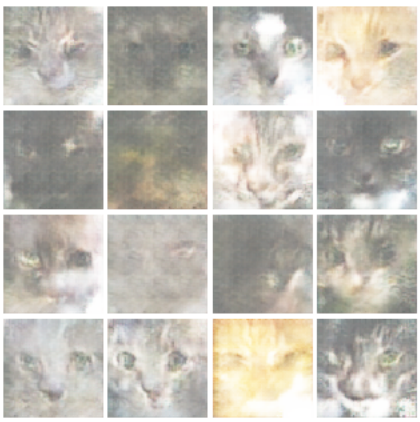
Iter: 2750, D: 1.334, G:4.081

EPOCH:  7
Iter: 3000, D: 0.7145, G:3.047



Iter: 3250, D: 0.5861, G:3.654

EPOCH:  8
Iter: 3500, D: 0.6522, G:2.149



Iter: 3750, D: 0.2513, G:2.74

EPOCH:  9
Iter: 4000, D: 0.4427, G:2.123

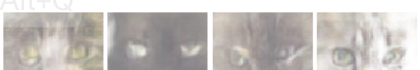
Iter: 4250, D: 0.3313, G:2.422

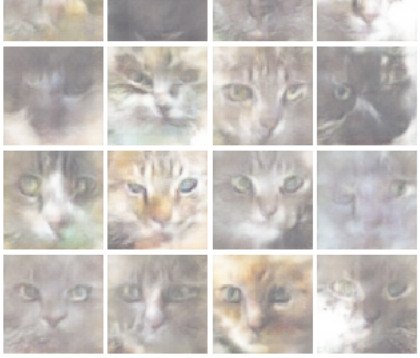
EPOCH:  10
Iter: 4500, D: 0.2112, G:4.82

Iter: 4750, D: 0.4704, G:5.033

EPOCH:  11
Iter: 5000, D: 0.3216, G:6.3

Iter: 5250, D: 0.2058, G:4.732



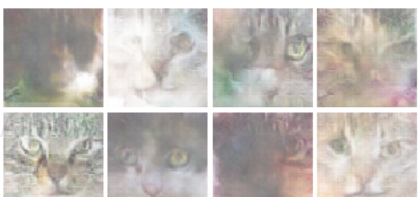


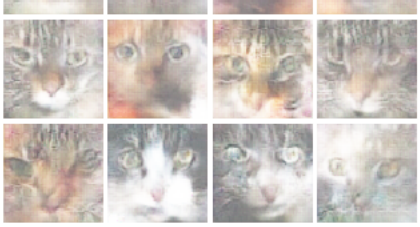
EPOCH:  12
Iter: 5500, D: 0.3856, G:7.39

Iter: 5750, D: 0.1506, G:3.392

EPOCH:  13
Iter: 6000, D: 0.2137, G:4.73

Iter: 6250, D: 0.08691, G:5.7



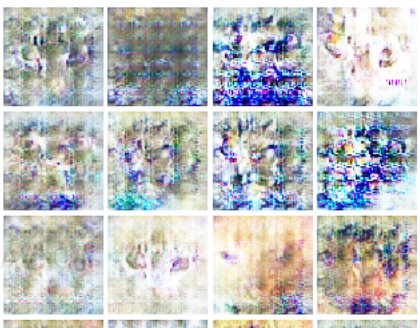


EPOCH:  14
Iter: 6500, D: 0.04791, G:5.405

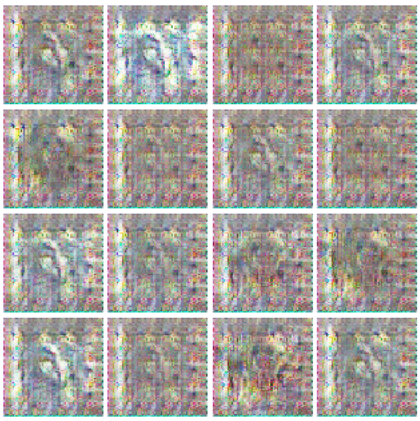
Iter: 6750, D: 0.7524, G:9.153

EPOCH:  15
Iter: 7000, D: 0.02584, G:5.369

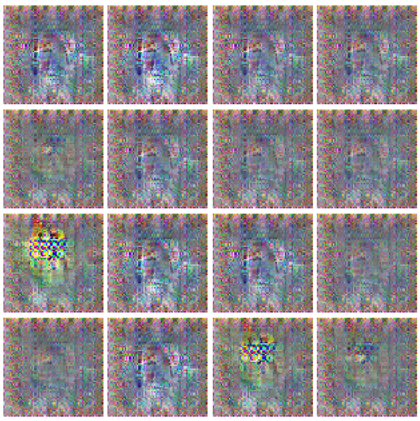
Iter: 7250, D: 0.02313, G:6.916



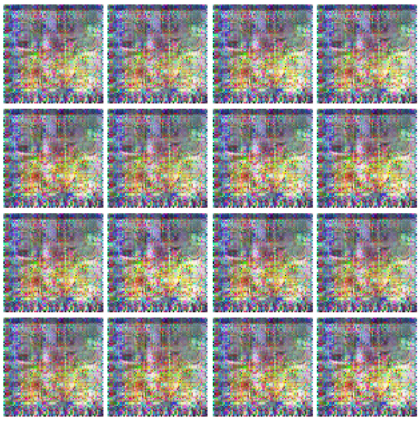
EPOCH:  16
Iter: 7500, D: 0.007139, G:7.551



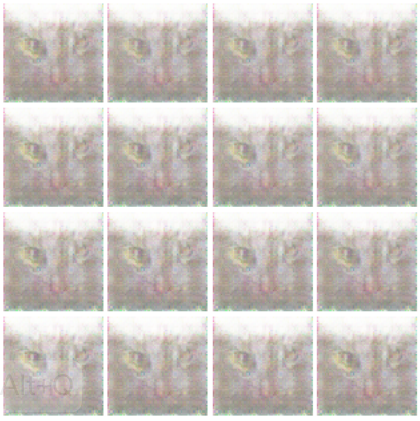
Iter: 7750, D: 0.0005296, G:8.18



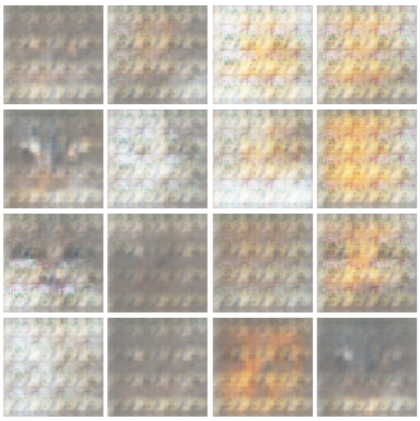
EPOCH:  17
Iter: 8000, D: 1.976e-05, G:15.48



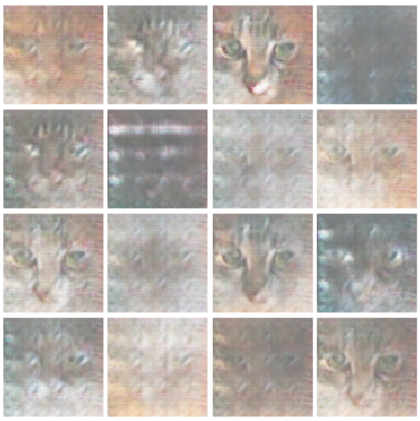
Iter: 8250, D: 0.0001648, G:39.14



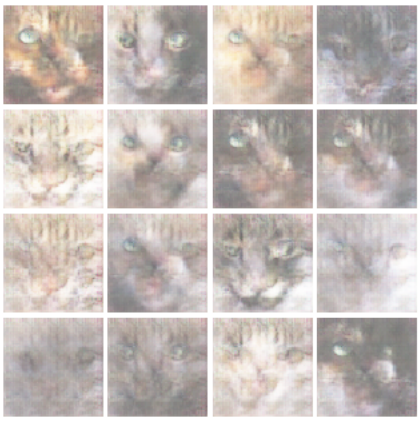
EPOCH:  18
Iter: 8500, D: 0.1004, G:6.548



Iter: 8750, D: 0.03448, G:7.636

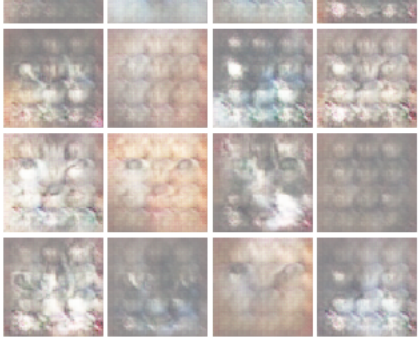


EPOCH:  19
Iter: 9000, D: 0.466, G:7.725



Iter: 9250, D: 0.1047, G:8.315

EPOCH:  20
Iter: 9500, D: 0.06076, G:6.453

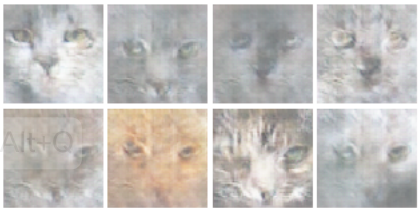


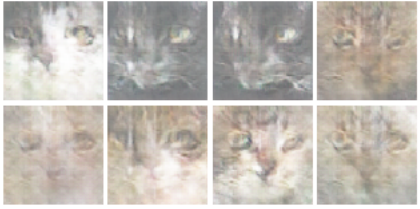
Iter: 9750, D: 0.1272, G:10.18

EPOCH:  21
Iter: 10000, D: 0.02186, G:8.294

Iter: 10250, D: 0.4661, G:7.568

EPOCH:  22
Iter: 10500, D: 0.03727, G:5.514

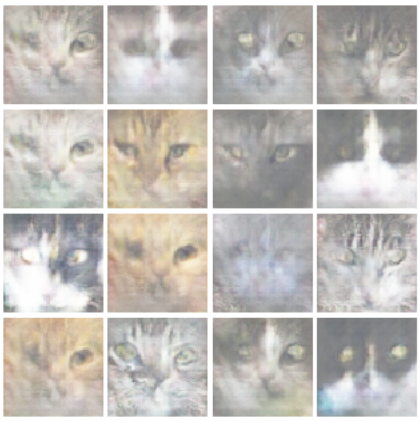




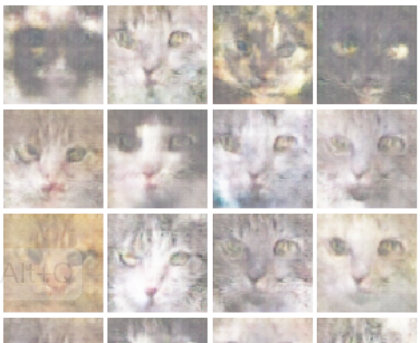
Iter: 10750, D: 3.177, G:23.89

EPOCH:  23
Iter: 11000, D: 0.0986, G:8.857

Iter: 11250, D: 0.0135, G:10.74



EPOCH:  24
Iter: 11500, D: 0.1785, G:6.648



Iter: 11750, D: 0.2865, G:11.29

EPOCH:  25
Iter: 12000, D: 0.05508, G:8.072

Iter: 12250, D: 0.02403, G:8.981

EPOCH:  26
Iter: 12500, D: 0.08988, G:5.659

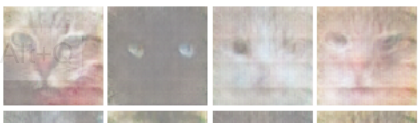
Iter: 12750, D: 2.436, G:11.41

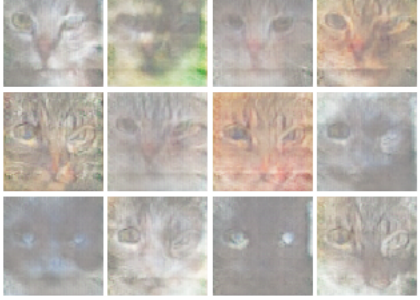
EPOCH:  27
Iter: 13000, D: 0.08051, G:8.076

Iter: 13250, D: 0.09018, G:5.813

EPOCH:  28
Iter: 13500, D: 0.4847, G:9.353

Iter: 13750, D: 0.1462, G:8.17



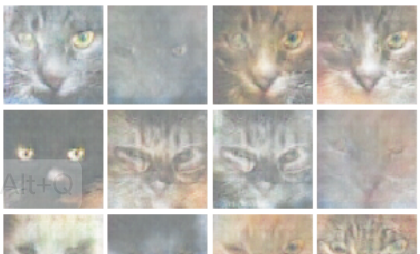


EPOCH:  29
Iter: 14000, D: 0.1872, G:10.31

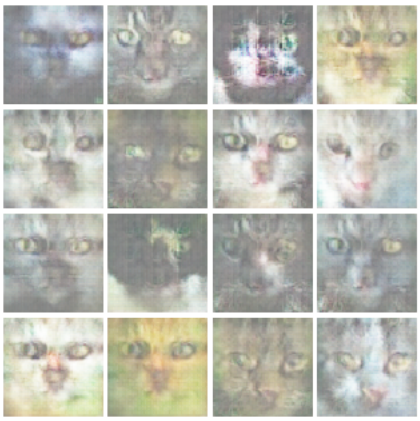
Iter: 14250, D: 0.04759, G:7.341

EPOCH:  30
Iter: 14500, D: 0.03389, G:6.373

Iter: 14750, D: 0.2461, G:9.222



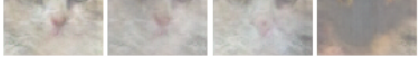
EPOCH:  31
Iter: 15000, D: 0.05453, G:6.652



Iter: 15250, D: 0.04684, G:11.24

EPOCH:  32
Iter: 15500, D: 0.1645, G:8.869

EPOCH:  33
Iter: 15750, D: 0.02804, G:8.139



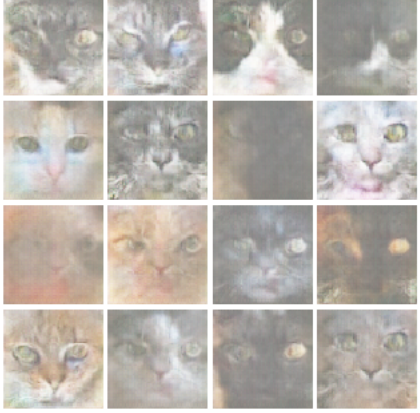
Iter: 16000, D: 0.03255, G:8.812

EPOCH:  34
Iter: 16250, D: 0.0445, G:8.608

Iter: 16500, D: 0.03176, G:9.594

EPOCH:  35
Iter: 16750, D: 0.01037, G:6.301

Iter: 17000, D: 0.003622, G:9.171

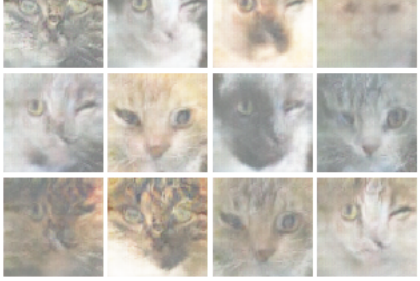


EPOCH:  36
Iter: 17250, D: 6.267, G:22.55

Iter: 17500, D: 0.027, G:6.338

EPOCH:  37
Iter: 17750, D: 0.2946, G:10.88

Iter: 18000, D: 0.02904, G:6.504

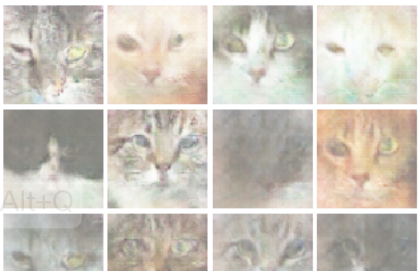


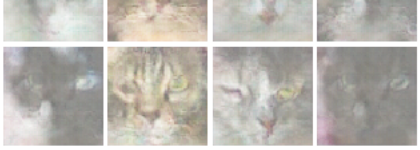
EPOCH:  38
Iter: 18250, D: 0.1327, G:5.132

Iter: 18500, D: 0.04387, G:7.312

EPOCH:  39
Iter: 18750, D: 0.01074, G:5.997

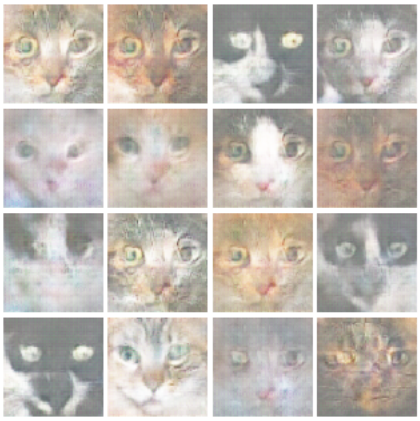
Iter: 19000, D: 0.1913, G:9.741





EPOCH:  40
Iter: 19250, D: 0.004206, G:7.149

Iter: 19500, D: 0.9517, G:17.4



EPOCH:  41
Iter: 19750, D: 0.01317, G:8.667

Iter: 20000, D: 0.2456, G:11.78

EPOCH:  42
Iter: 20250, D: 0.1777, G:6.409

Iter: 20500, D: 0.3138, G:9.189

EPOCH:  43
Iter: 20750, D: 0.3407, G:8.388

Iter: 21000, D: 0.1599, G:8.553

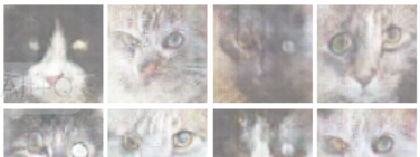
EPOCH:  44
Iter: 21250, D: 0.1078, G:4.975

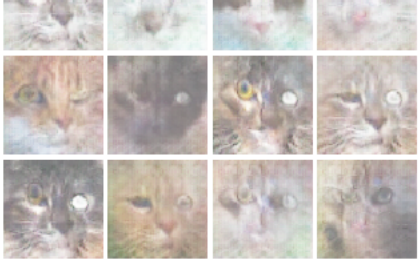
Iter: 21500, D: 0.1653, G:9.83

EPOCH:  45
Iter: 21750, D: 0.002121, G:6.347

Iter: 22000, D: 0.1816, G:12.32

EPOCH:  46
Iter: 22250, D: 0.01723, G:7.671



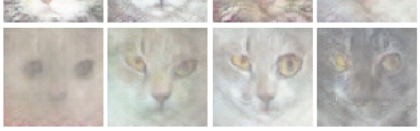


Iter: 22500, D: 0.02963, G:19.37

EPOCH:  47
Iter: 22750, D: 0.005308, G:11.33

Iter: 23000, D: 0.08054, G:10.02

EPOCH:  48
Iter: 23250, D: 0.05585, G:9.492



```
Iter: 23500, D: 0.005678, G:5.476
```

```
EPOCH:  49
Iter: 23750, D: 0.01358, G:9.314
```

## Train LS-GAN

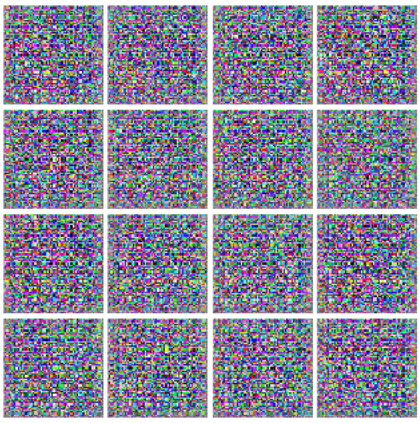
```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```



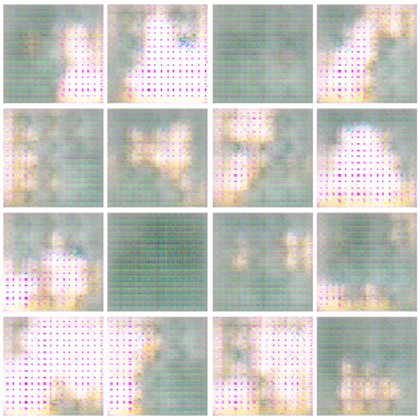
```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.999))

# default 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: 0.9051, G:31.82

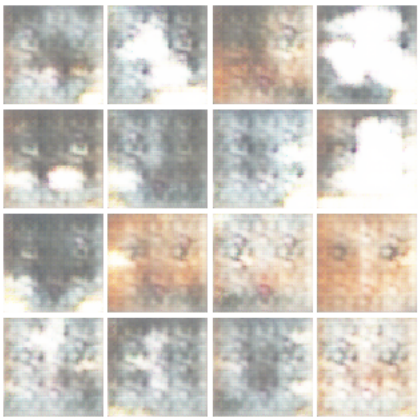


Iter: 250, D: 0.494, G:0.6876



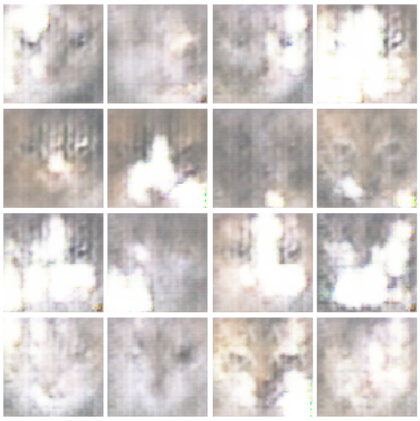
EPOCH:  2
Iter: 500, D: 0.4639, G:0.2442

Iter: 750, D: 0.3506, G:0.6265

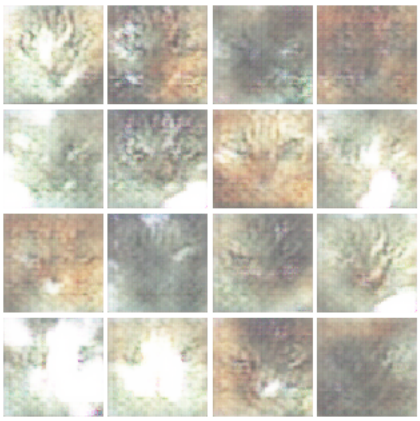


EPOCH:  3
Iter: 1000, D: 0.2665, G:0.3978

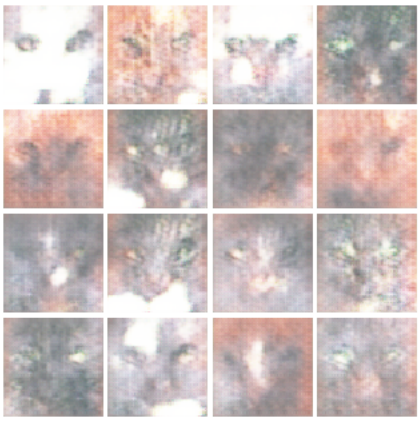
Iter: 1250, D: 0.3603, G:0.4334



EPOCH:  4
Iter: 1500, D: 0.3042, G:1.345



Iter: 1750, D: 0.8207, G:0.1738

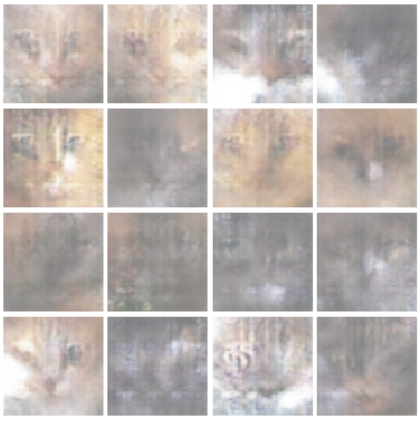


EPOCH:  5
Iter: 2000, D: 0.5936, G:0.4628

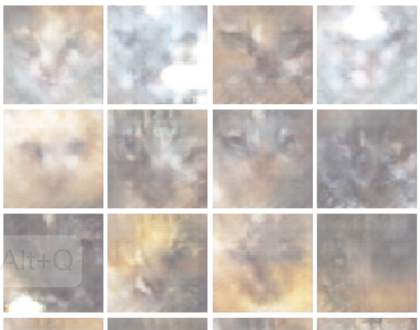
Iter: 2250, D: 0.3505, G:0.4596

EPOCH:  6
Iter: 2500, D: 0.3537, G:0.7555

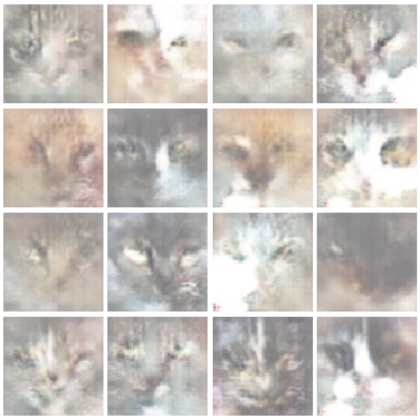
Iter: 2750, D: 0.1911, G:1.006



EPOCH:  7
Iter: 3000, D: 0.2527, G:0.6386



Iter: 3250, D: 0.2866, G:0.6723



EPOCH:  8
Iter: 3500, D: 0.36, G:0.445

Iter: 3750, D: 0.2878, G:0.525

EPOCH:  9
Iter: 4000, D: 0.1532, G:0.5567

Iter: 4250, D: 0.09432, G:0.8286

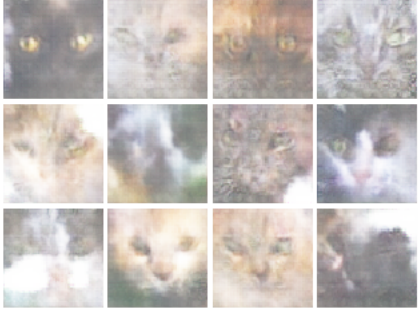
Iter: 4250, D: 0.09432, G:0.8286

EPOCH:  10
Iter: 4500, D: 0.1983, G:1.146

Iter: 4750, D: 0.254, G:0.6202

EPOCH:  11
Iter: 5000, D: 0.07278, G:1.055

Iter: 5250, D: 0.2014, G:0.7008

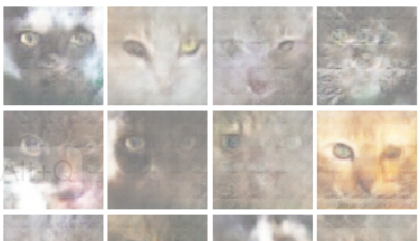


EPOCH:  12
Iter: 5500, D: 0.1222, G:0.8312

Iter: 5750, D: 0.08405, G:0.9092

EPOCH:  13
Iter: 6000, D: 0.3684, G:0.5358

Iter: 6250, D: 0.1844, G:1.079

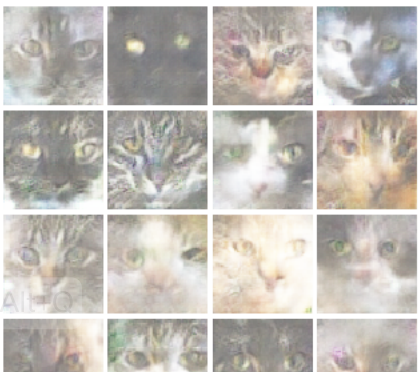


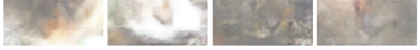
EPOCH:  14
Iter: 6500, D: 0.1286, G:0.802

Iter: 6750, D: 0.1936, G:0.8203

EPOCH:  15
Iter: 7000, D: 0.2907, G:1.259

Iter: 7250, D: 0.159, G:0.6063





EPOCH:  16
Iter: 7500, D: 0.169, G:1.102

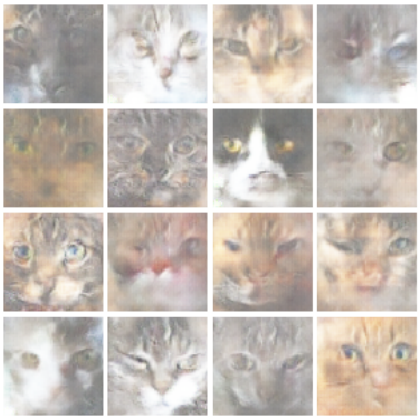
Iter: 7750, D: 0.1915, G:0.7845

EPOCH:  17
Iter: 8000, D: 0.1903, G:1.657

Iter: 8250, D: 0.1658, G:1.045



EPOCH:  18
Iter: 8500, D: 0.1273, G:0.9983

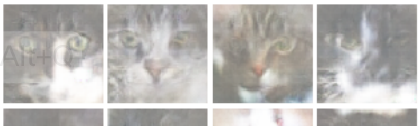


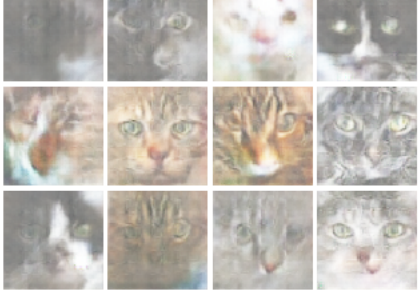
Iter: 8750, D: 0.1297, G:0.9305

EPOCH:  19
Iter: 9000, D: 0.09855, G:0.7985

Iter: 9250, D: 0.04403, G:0.8487

EPOCH:  20
Iter: 9500, D: 0.1306, G:0.7928



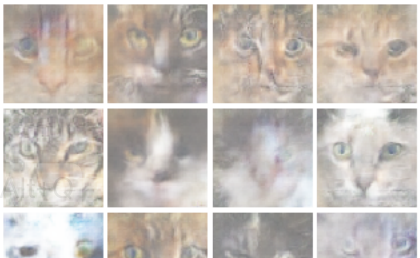


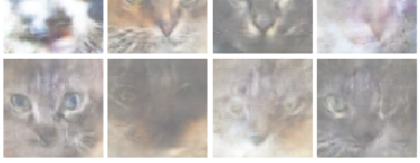
Iter: 9750, D: 0.1545, G:1.442

EPOCH:  21
Iter: 10000, D: 0.07784, G:1.216

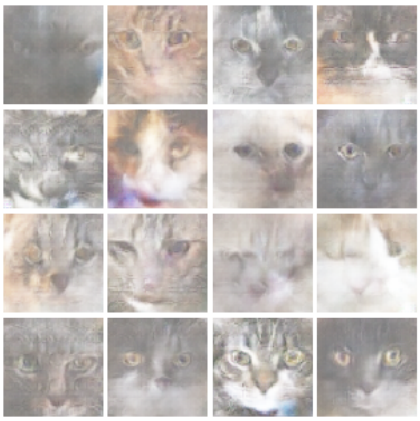
Iter: 10250, D: 0.06578, G:0.7414

EPOCH:  22
Iter: 10500, D: 0.09487, G:0.831





Iter: 10750, D: 0.059, G:1.202



EPOCH:  23
Iter: 11000, D: 0.1423, G:0.8791

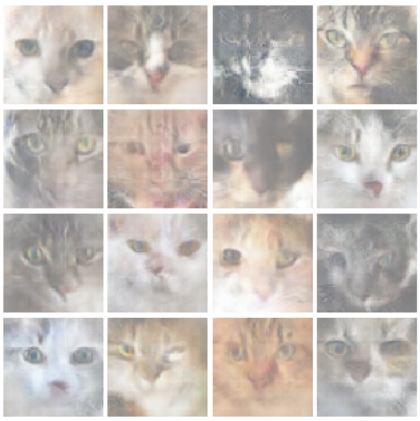
Iter: 11250, D: 0.1439, G:1.763

EPOCH:  24
Iter: 11500, D: 0.1507, G:0.9113

Iter: 11750, D: 0.107, G:0.6437

EPOCH:  25
Iter: 12000, D: 0.1261, G:1.438

Iter: 12250, D: 0.06329, G:0.9867



EPOCH:  26
Iter: 12500, D: 0.04847, G:0.9894

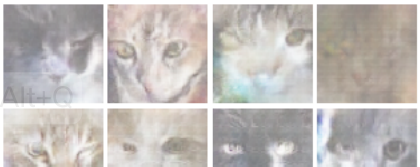
Iter: 12750, D: 0.07956, G:1.033

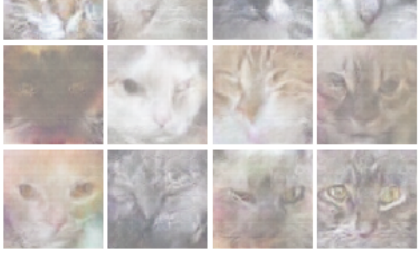
EPOCH:  27
Iter: 13000, D: 0.03513, G:1.349

Iter: 13250, D: 0.09473, G:0.6015

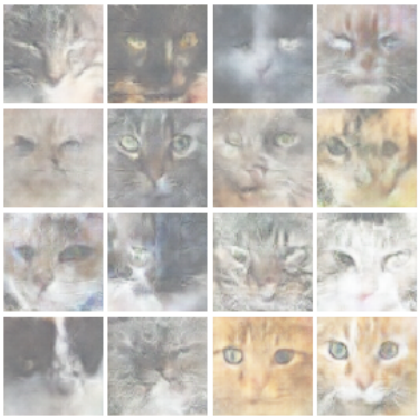
EPOCH:  28
Iter: 13500, D: 0.1331, G:0.5

Iter: 13750, D: 0.04539, G:0.7968





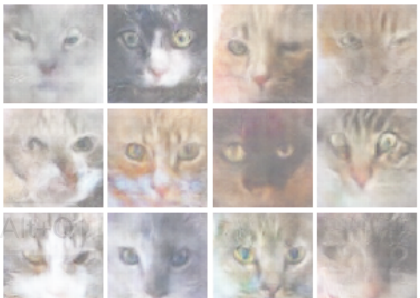
EPOCH:  29
Iter: 14000, D: 0.1089, G:0.7395

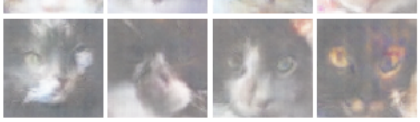


Iter: 14250, D: 0.1057, G:0.8082

EPOCH:  30
Iter: 14500, D: 0.05947, G:1.212

Iter: 14750, D: 0.0768, G:1.13



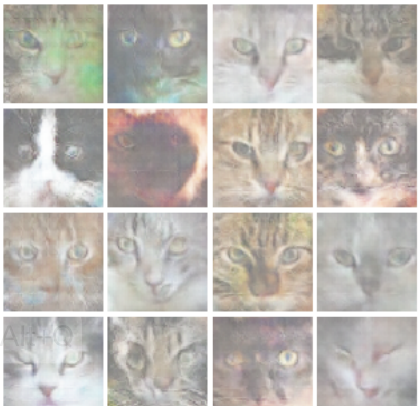


EPOCH:  31
Iter: 15000, D: 0.09253, G:1.106

Iter: 15250, D: 0.03511, G:1.012

EPOCH:  32
Iter: 15500, D: 0.1081, G:1.48

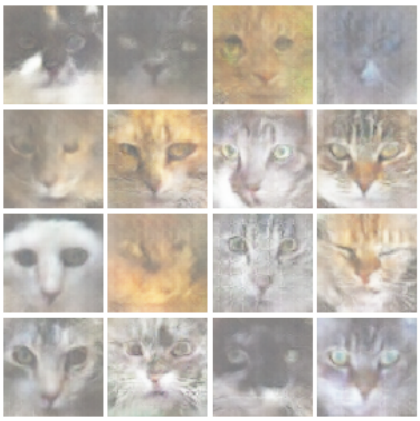
EPOCH:  33
Iter: 15750, D: 0.05206, G:1.446



Iter: 16000, D: 0.1769, G:1.498

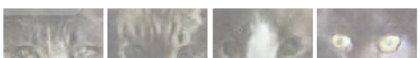
EPOCH:  34
Iter: 16250, D: 0.03723, G:1.288

Iter: 16500, D: 0.0335, G:0.813

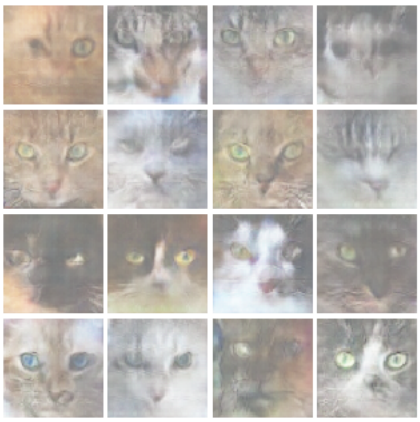


EPOCH:  35
Iter: 16750, D: 0.05509, G:0.8695

Iter: 17000, D: 0.1509, G:0.8049



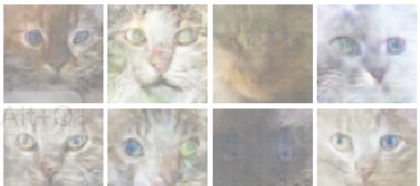
EPOCH:  36
Iter: 17250, D: 0.1419, G:1.125

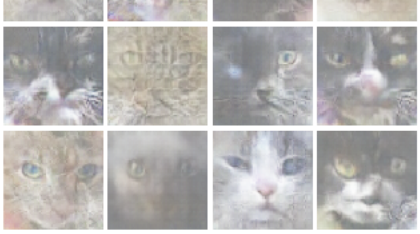


Iter: 17500, D: 0.08103, G:0.7166

EPOCH:  37
Iter: 17750, D: 0.07883, G:1.496

Iter: 18000, D: 0.08668, G:1.157



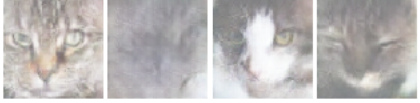


EPOCH:  38
Iter: 18250, D: 0.151, G:0.5944

Iter: 18500, D: 0.0334, G:1.16

EPOCH:  39
Iter: 18750, D: 0.04385, G:1.244

Iter: 19000, D: 0.03921, G:0.9263



EPOCH:  40
Iter: 19250, D: 0.09095, G:1.682

Iter: 19500, D: 0.04769, G:0.8927

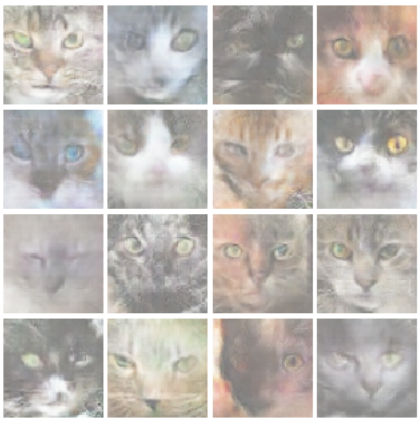
EPOCH:  41
Iter: 19750, D: 0.06901, G:0.6549

Iter: 20000, D: 0.2912, G:0.6517

EPOCH:  42
Iter: 20250, D: 0.04952, G:0.9108

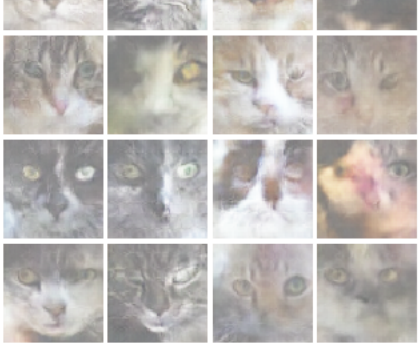
Iter: 20500, D: 0.06106, G:1.261

EPOCH:  43
Iter: 20750, D: 0.04882, G:1.073



Iter: 21000, D: 0.03985, G:0.8783

EPOCH:  44
Iter: 21250, D: 0.06278, G:0.8178

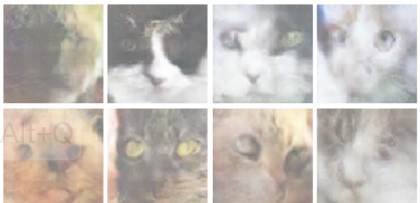


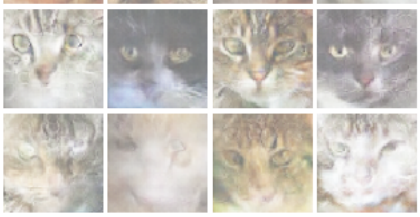
Iter: 21500, D: 0.05479, G:0.8499

EPOCH:  45
Iter: 21750, D: 0.03921, G:0.8772

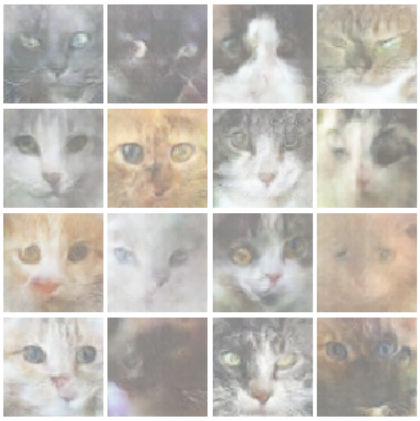
Iter: 22000, D: 0.09923, G:0.8288

EPOCH:  46
Iter: 22250, D: 0.07719, G:0.6311



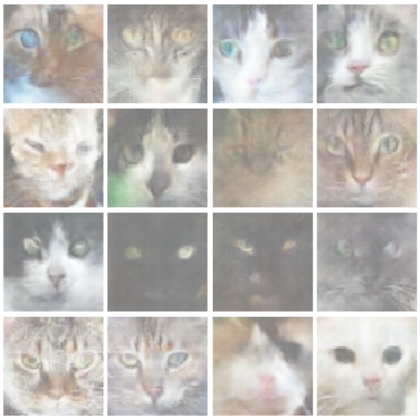


Iter: 22500, D: 0.09058, G:1.017

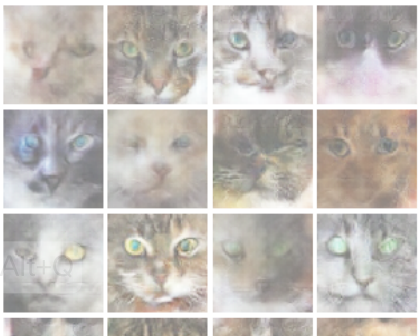


EPOCH:  47
Iter: 22750, D: 0.0205, G:1.088

Iter: 23000, D: 0.03951, G:1.003

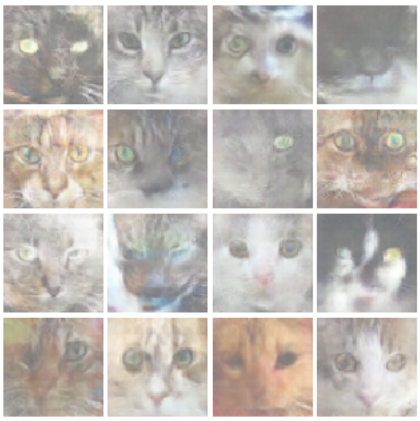


EPOCH:  48
Iter: 23250, D: 0.06864, G:0.9358



Iter: 23500, D: 0.2035, G:0.5662

EPOCH:  49
Iter: 23750, D: 0.03067, G:0.9909



Iter: 24000, D: 0.09764, G:0.6439

EPOCH:  50
Iter: 24250, D: 0.03836, G:0.908

Iter: 24500, D: 0.1092, G:0.7194

Iter: 24500, D: 0.1052, G: 3.077194