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

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318

Enter your authorization code:
··········
Mounted at /content/drive
```

```
!unzip '/content/drive/My Drive/Flickr-dog.zip' -d '/content/sample_data/'
```

```
!apt-get install p7zip-full
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
p7zip-full is already the newest version (16.02+dfsg-6).
0 upgraded, 0 newly installed, 0 to remove and 28 not upgraded.
```

```
!tar -xvf DogsImages.tar -C /content/sample_data/
```

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import zipfile
import cv2
# import unzip

# Set random seem for reproducibility
manualSeed = 999
```

```
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

```
Random Seed:  999
<torch._C.Generator at 0x7f71b7b4d0b0>
```

```
# drive = '/content/sample_data/square_data/squares/'
# for i in range(5000):
#     x = random.randrange(0,5)
#     img = np.zeros((64,64,3), np.uint8)
#     color = (random.randrange(1,255), random.randrange(1,255), random.randrange(1,255))

#     img[42-x:42+x, 42-x:42+x, :] = tuple(reversed(color))
#     rotationMat = cv2.getRotationMatrix2D((32,32), random.randrange(0,90), 1)
#     img = cv2.warpAffine(img, rotationMat,(img.shape[0],img.shape[1]))
#     trans_mat = np.float32([[1,0,random.randrange(-20,20)], [0,1,random.randrange(-20,20)]]
#     img = cv2.warpAffine(img, trans_mat, (img.shape[0],img.shape[1]))
#     # cv2.war
#     file = drive + 'square'+ str(i) + '.png'
#     plt.imsave(file, img)
```

Inputs

Let's define some inputs for the run:

- **dataroot** - the path to the root of the dataset folder. We will talk more about the dataset in the n
- **workers** - the number of worker threads for loading the data with the DataLoader
- **batch_size** - the batch size used in training. The DCGAN paper uses a batch size of 128
- **image_size** - the spatial size of the images used for training. This implementation defaults to 6 structures of D and G must be changed. See `here <https://github.com/pytorch/examples/is`
- **nc** - number of color channels in the input images. For color images this is 3
- **nz** - length of latent vector
- **ngf** - relates to the depth of feature maps carried through the generator
- **ndf** - sets the depth of feature maps propagated through the discriminator
- **num_epochs** - number of training epochs to run. Training for longer will probably lead to better
- **lr** - learning rate for training. As described in the DCGAN paper, this number should be 0.0002
- **beta1** - beta1 hyperparameter for Adam optimizers. As described in paper, this number should
- **ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greate

```
# Root directory for dataset
```

```
dataroot = "/content/sample_data/Flickr-dog"

# Number of workers for dataloader
workers = 4

# Batch size during training
batch_size = 32

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 100

# Learning rate for optimizers
lr = 0.0005

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1


# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

```
# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=
```

```
<matplotlib.image.AxesImage at 0x7f71ab5702b0>
```

Training Images

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
```

```
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

Now, we can instantiate the generator and apply the `weights_init` function. Check out the printed m structured.

```
# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
    (13): Tanh()
  )
)
```

Discriminator Code

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Now, as with the generator, we can create the discriminator, apply the `weights_init` function, and pri

```
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
```

```
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
```

```
        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 100 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
Starting Training Loop...
[0/100][0/12]   Loss_D: 0.4606  Loss_G: 7.8754  D(x): 0.9659    D(G(z)): 0.2564 / 0.0009
[1/100][0/12]   Loss_D: 0.1862  Loss_G: 4.5960  D(x): 0.9394    D(G(z)): 0.0992 / 0.0185
[2/100][0/12]   Loss_D: 0.2264  Loss_G: 4.9598  D(x): 0.9258    D(G(z)): 0.1178 / 0.0131
[3/100][0/12]   Loss_D: 0.2165  Loss_G: 5.4288  D(x): 0.9006    D(G(z)): 0.0883 / 0.0105
[4/100][0/12]   Loss_D: 0.2408  Loss_G: 5.7874  D(x): 0.9047    D(G(z)): 0.0839 / 0.0071
[5/100][0/12]   Loss_D: 0.1308  Loss_G: 5.3680  D(x): 0.9438    D(G(z)): 0.0644 / 0.0115
[6/100][0/12]   Loss_D: 0.4179  Loss_G: 8.1568  D(x): 0.9725    D(G(z)): 0.2502 / 0.0007
[7/100][0/12]   Loss_D: 0.6443  Loss_G: 5.4085  D(x): 0.7318    D(G(z)): 0.1180 / 0.0136
[8/100][0/12]   Loss_D: 0.3162  Loss_G: 5.6046  D(x): 0.9264    D(G(z)): 0.1086 / 0.0074
[9/100][0/12]   Loss_D: 0.1338  Loss_G: 5.2713  D(x): 0.9542    D(G(z)): 0.0676 / 0.0095
[10/100][0/12]  Loss_D: 0.4878  Loss_G: 8.0307  D(x): 0.9371    D(G(z)): 0.1855 / 0.0062
[11/100][0/12]  Loss_D: 0.2537  Loss_G: 5.4491  D(x): 0.9483    D(G(z)): 0.1642 / 0.0129
[12/100][0/12]  Loss_D: 0.1785  Loss_G: 5.3023  D(x): 0.9253    D(G(z)): 0.0688 / 0.0164
[13/100][0/12]  Loss_D: 0.1382  Loss_G: 5.9818  D(x): 0.9234    D(G(z)): 0.0445 / 0.0057
[14/100][0/12]  Loss_D: 0.1763  Loss_G: 5.0628  D(x): 0.9072    D(G(z)): 0.0539 / 0.0135
[15/100][0/12]  Loss_D: 0.2831  Loss_G: 5.3494  D(x): 0.8712    D(G(z)): 0.0888 / 0.0117
[16/100][0/12]  Loss_D: 0.7262  Loss_G: 10.4547 D(x): 0.9827    D(G(z)): 0.3724 / 0.0001
[17/100][0/12]  Loss_D: 0.3887  Loss_G: 5.2466  D(x): 0.8774    D(G(z)): 0.1070 / 0.0220
[18/100][0/12]  Loss_D: 0.4538  Loss_G: 8.5613  D(x): 0.9486    D(G(z)): 0.2296 / 0.0014
[19/100][0/12]  Loss_D: 0.1833  Loss_G: 7.4461  D(x): 0.9591    D(G(z)): 0.0969 / 0.0018
[20/100][0/12]  Loss_D: 0.1336  Loss_G: 5.1001  D(x): 0.9409    D(G(z)): 0.0578 / 0.0111
[21/100][0/12]  Loss_D: 0.0978  Loss_G: 6.0094  D(x): 0.9593    D(G(z)): 0.0493 / 0.0053
[22/100][0/12]  Loss_D: 0.2283  Loss_G: 7.3405  D(x): 0.9914    D(G(z)): 0.1518 / 0.0009
[23/100][0/12]  Loss_D: 0.1869  Loss_G: 6.2461  D(x): 0.9394    D(G(z)): 0.1026 / 0.0039
[24/100][0/12]  Loss_D: 0.0542  Loss_G: 5.7488  D(x): 0.9682    D(G(z)): 0.0199 / 0.0066
[25/100][0/12]  Loss_D: 0.1576  Loss_G: 5.8005  D(x): 0.9781    D(G(z)): 0.1121 / 0.0058
[26/100][0/12]  Loss_D: 0.1305  Loss_G: 5.6543  D(x): 0.9553    D(G(z)): 0.0691 / 0.0062
[27/100][0/12]  Loss_D: 0.2468  Loss_G: 7.0703  D(x): 0.9923    D(G(z)): 0.1539 / 0.0030
[28/100][0/12]  Loss_D: 0.1727  Loss_G: 6.2242  D(x): 0.8831    D(G(z)): 0.0284 / 0.0099
[29/100][0/12]  Loss_D: 0.3905  Loss_G: 7.0902  D(x): 0.8420    D(G(z)): 0.0184 / 0.0238
[30/100][0/12]  Loss_D: 0.8110  Loss_G: 10.5747 D(x): 0.9417    D(G(z)): 0.3813 / 0.0007
[31/100][0/12]  Loss_D: 0.4732  Loss_G: 10.2280 D(x): 0.9777    D(G(z)): 0.2924 / 0.0012
[32/100][0/12]  Loss_D: 0.2470  Loss_G: 7.3577  D(x): 0.9327    D(G(z)): 0.1172 / 0.0018
[33/100][0/12]  Loss_D: 0.2783  Loss_G: 8.9674  D(x): 0.9152    D(G(z)): 0.1362 / 0.0025
[34/100][0/12]  Loss_D: 0.1655  Loss_G: 6.5588  D(x): 0.9478    D(G(z)): 0.0800 / 0.0045
[35/100][0/12]  Loss_D: 0.1825  Loss_G: 6.2660  D(x): 0.9869    D(G(z)): 0.1311 / 0.0068
[36/100][0/12]  Loss_D: 0.4751  Loss_G: 11.4205 D(x): 0.9865    D(G(z)): 0.2758 / 0.0004
[37/100][0/12]  Loss_D: 0.1478  Loss_G: 5.9893  D(x): 0.9504    D(G(z)): 0.0700 / 0.0059
[38/100][0/12]  Loss_D: 0.1601  Loss_G: 7.9928  D(x): 0.9878    D(G(z)): 0.1238 / 0.0016
[39/100][0/12]  Loss_D: 0.0710  Loss_G: 6.7695  D(x): 0.9727    D(G(z)): 0.0400 / 0.0039
[40/100][0/12]  Loss_D: 0.0584  Loss_G: 6.2230  D(x): 0.9856    D(G(z)): 0.0409 / 0.0045
[41/100][0/12]  Loss_D: 0.0463  Loss_G: 5.8976  D(x): 0.9792    D(G(z)): 0.0236 / 0.0068
[42/100][0/12]  Loss_D: 0.0396  Loss_G: 6.2987  D(x): 0.9795    D(G(z)): 0.0180 / 0.0046
[43/100][0/12]  Loss_D: 0.0446  Loss_G: 8.3428  D(x): 0.9674    D(G(z)): 0.0093 / 0.0017
[44/100][0/12]  Loss_D: 0.0683  Loss_G: 6.2304  D(x): 0.9888    D(G(z)): 0.0475 / 0.0042
[45/100][0/12]  Loss_D: 0.0580  Loss_G: 6.6429  D(x): 0.9860    D(G(z)): 0.0404 / 0.0044
[46/100][0/12]  Loss_D: 0.0932  Loss_G: 6.5549  D(x): 0.9841    D(G(z)): 0.0680 / 0.0031
[47/100][0/12]  Loss_D: 0.6402  Loss_G: 10.1137 D(x): 0.9381    D(G(z)): 0.2609 / 0.0013
[48/100][0/12]  Loss_D: 0.3478  Loss_G: 9.6045  D(x): 0.9248    D(G(z)): 0.1520 / 0.0016
[49/100][0/12]  Loss_D: 1.1717  Loss_G: 7.3010  D(x): 0.6630    D(G(z)): 0.1406 / 0.0204
[50/100][0/12]  Loss_D: 0.4913  Loss_G: 9.2208  D(x): 0.7613    D(G(z)): 0.0229 / 0.0415
[51/100][0/12]  Loss_D: 1.1828  Loss_G: 7.0112  D(x): 0.4549    D(G(z)): 0.0048 / 0.0451
[52/100][0/12]  Loss_D: 0.2182  Loss_G: 7.2926  D(x): 0.9361    D(G(z)): 0.1070 / 0.0063
[53/100][0/12]  Loss_D: 0.1773  Loss_G: 7.4166  D(x): 0.8988    D(G(z)): 0.0156 / 0.0048
[54/100][0/12]  Loss_D: 0.1286  Loss_G: 6.2247  D(x): 0.9754    D(G(z)): 0.0864 / 0.0039
[55/100][0/12]  Loss_D: 0.0595  Loss_G: 5.3710  D(x): 0.9690    D(G(z)): 0.0241 / 0.0091
```

```
[56/100][0/12]  Loss_D: 0.0567  Loss_G: 6.3720  D(x): 0.9758    D(G(z)): 0.0298 / 0.0041
[57/100][0/12]  Loss_D: 0.1755  Loss_G: 8.0688  D(x): 0.9960    D(G(z)): 0.1343 / 0.0005
[58/100][0/12]  Loss_D: 0.0416  Loss_G: 6.2708  D(x): 0.9742    D(G(z)): 0.0144 / 0.0040
[59/100][0/12]  Loss_D: 0.1344  Loss_G: 8.8964  D(x): 0.9893    D(G(z)): 0.1003 / 0.0013
[60/100][0/12]  Loss_D: 0.1411  Loss_G: 7.2474  D(x): 0.9821    D(G(z)): 0.0655 / 0.0085
[61/100][0/12]  Loss_D: 0.0497  Loss_G: 5.1717  D(x): 0.9716    D(G(z)): 0.0183 / 0.0129
[62/100][0/12]  Loss_D: 0.0815  Loss_G: 6.1618  D(x): 0.9904    D(G(z)): 0.0606 / 0.0037
[63/100][0/12]  Loss_D: 0.0411  Loss_G: 6.3632  D(x): 0.9831    D(G(z)): 0.0227 / 0.0052
[64/100][0/12]  Loss_D: 0.0206  Loss_G: 6.2924  D(x): 0.9909    D(G(z)): 0.0112 / 0.0039
[65/100][0/12]  Loss_D: 0.0843  Loss_G: 7.3455  D(x): 0.9899    D(G(z)): 0.0595 / 0.0013
[66/100][0/12]  Loss_D: 0.0479  Loss_G: 5.8734  D(x): 0.9886    D(G(z)): 0.0346 / 0.0048
[67/100][0/12]  Loss_D: 0.0868  Loss_G: 6.5758  D(x): 0.9943    D(G(z)): 0.0719 / 0.0022
[68/100][0/12]  Loss_D: 0.0796  Loss_G: 6.8328  D(x): 0.9972    D(G(z)): 0.0679 / 0.0026
[69/100][0/12]  Loss_D: 0.0514  Loss_G: 5.2122  D(x): 0.9711    D(G(z)): 0.0201 / 0.0113
[70/100][0/12]  Loss_D: 0.0253  Loss_G: 5.4775  D(x): 0.9871    D(G(z)): 0.0119 / 0.0087
[71/100][0/12]  Loss_D: 0.0325  Loss_G: 6.1019  D(x): 0.9913    D(G(z)): 0.0219 / 0.0051
[72/100][0/12]  Loss_D: 0.0318  Loss_G: 6.5252  D(x): 0.9810    D(G(z)): 0.0118 / 0.0035
[73/100][0/12]  Loss_D: 0.0251  Loss_G: 6.0494  D(x): 0.9939    D(G(z)): 0.0184 / 0.0062
[74/100][0/12]  Loss_D: 0.0340  Loss_G: 6.8148  D(x): 0.9896    D(G(z)): 0.0217 / 0.0033
[75/100][0/12]  Loss_D: 0.0384  Loss_G: 6.5937  D(x): 0.9930    D(G(z)): 0.0291 / 0.0041
[76/100][0/12]  Loss_D: 0.0471  Loss_G: 7.7230  D(x): 0.9572    D(G(z)): 0.0017 / 0.0014
[77/100][0/12]  Loss_D: 0.0472  Loss_G: 6.9048  D(x): 0.9682    D(G(z)): 0.0119 / 0.0027
[78/100][0/12]  Loss_D: 0.0449  Loss_G: 6.3658  D(x): 0.9632    D(G(z)): 0.0063 / 0.0075
[79/100][0/12]  Loss_D: 0.0229  Loss_G: 5.7054  D(x): 0.9893    D(G(z)): 0.0117 / 0.0064
[80/100][0/12]  Loss_D: 0.1462  Loss_G: 8.9476  D(x): 0.9965    D(G(z)): 0.1089 / 0.0005
[81/100][0/12]  Loss_D: 0.0475  Loss_G: 5.7444  D(x): 0.9613    D(G(z)): 0.0069 / 0.0134
[82/100][0/12]  Loss_D: 0.0281  Loss_G: 5.8606  D(x): 0.9869    D(G(z)): 0.0144 / 0.0066
[83/100][0/12]  Loss_D: 0.0441  Loss_G: 6.7813  D(x): 0.9865    D(G(z)): 0.0285 / 0.0029
[84/100][0/12]  Loss_D: 0.0424  Loss_G: 6.1440  D(x): 0.9828    D(G(z)): 0.0236 / 0.0045
[85/100][0/12]  Loss_D: 0.0786  Loss_G: 6.3675  D(x): 0.9342    D(G(z)): 0.0028 / 0.0040
[86/100][0/12]  Loss_D: 0.1547  Loss_G: 6.4302  D(x): 0.8837    D(G(z)): 0.0007 / 0.0089
[87/100][0/12]  Loss_D: 4.7125  Loss_G: 0.8092  D(x): 0.4666    D(G(z)): 0.5499 / 0.5697
[88/100][0/12]  Loss_D: 1.7826  Loss_G: 2.0511  D(x): 0.5723    D(G(z)): 0.4396 / 0.2956
[89/100][0/12]  Loss_D: 1.7346  Loss_G: 1.9253  D(x): 0.7037    D(G(z)): 0.5625 / 0.2266
[90/100][0/12]  Loss_D: 1.1681  Loss_G: 1.6703  D(x): 0.7102    D(G(z)): 0.4111 / 0.2534
[91/100][0/12]  Loss_D: 1.2103  Loss_G: 2.3459  D(x): 0.7785    D(G(z)): 0.5191 / 0.1521
[92/100][0/12]  Loss_D: 0.9493  Loss_G: 2.1241  D(x): 0.5836    D(G(z)): 0.1954 / 0.1696
[93/100][0/12]  Loss_D: 1.6361  Loss_G: 2.6665  D(x): 0.7286    D(G(z)): 0.5965 / 0.1215
[94/100][0/12]  Loss_D: 1.2198  Loss_G: 2.6541  D(x): 0.6286    D(G(z)): 0.4099 / 0.1223
[95/100][0/12]  Loss_D: 1.3874  Loss_G: 3.7668  D(x): 0.7326    D(G(z)): 0.4602 / 0.1140
[96/100][0/12]  Loss_D: 1.0122  Loss_G: 3.9321  D(x): 0.8461    D(G(z)): 0.4158 / 0.0649
[97/100][0/12]  Loss_D: 0.9326  Loss_G: 3.2308  D(x): 0.7794    D(G(z)): 0.3694 / 0.0754
[98/100][0/12]  Loss_D: 1.3985  Loss_G: 5.5441  D(x): 0.7832    D(G(z)): 0.5211 / 0.0176
[99/100][0/12]  Loss_D: 0.4006  Loss_G: 4.6075  D(x): 0.9030    D(G(z)): 0.2066 / 0.0328
```

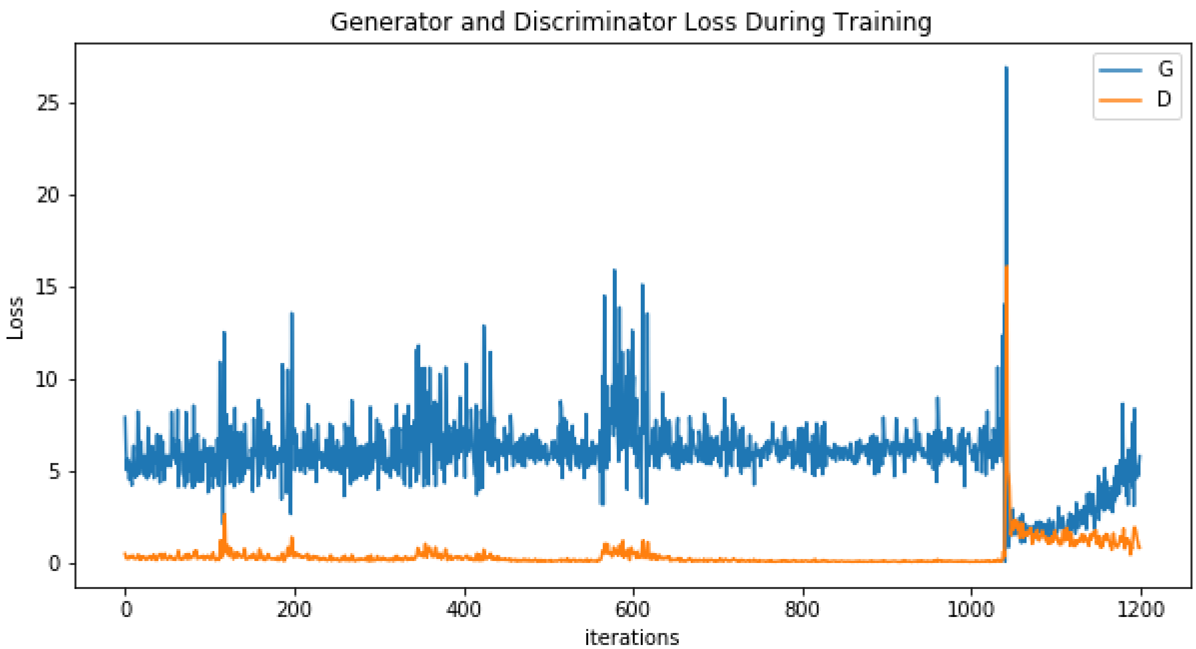
Results

Finally, lets check out how we did. Here, we will look at three different results. First, we will see how D
Second, we will visualize G's output on the fixed_noise batch for every epoch. And third, we will look a
fake data from G.

Loss versus training iteration

Below is a plot of D & G's losses versus training iterations.

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

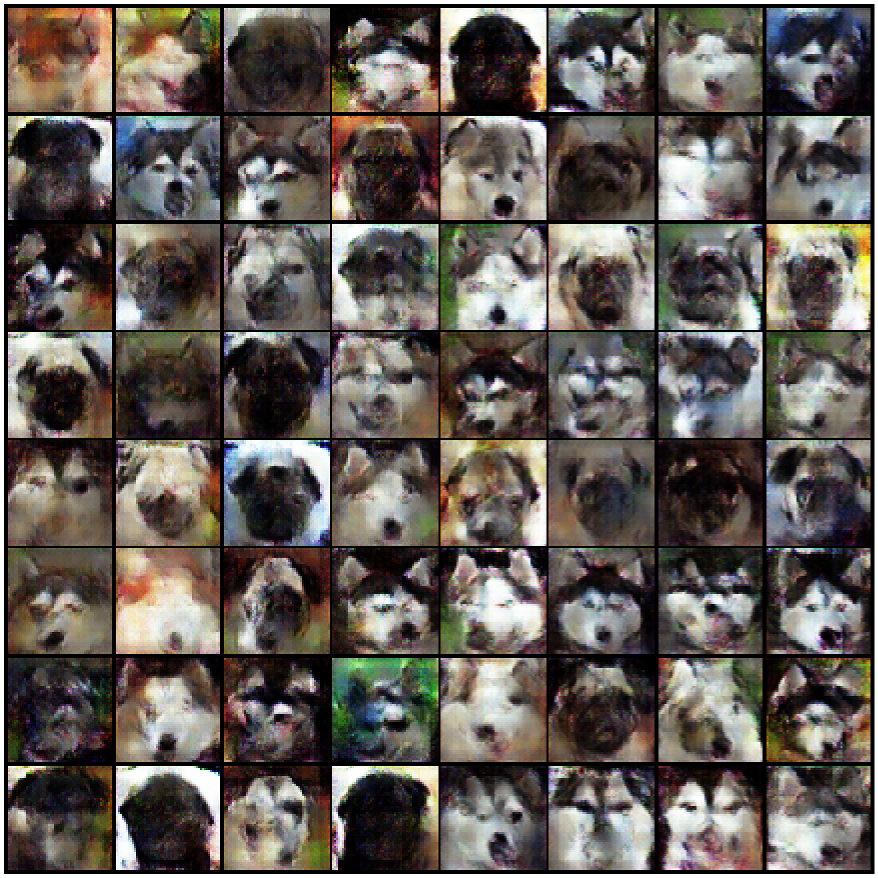


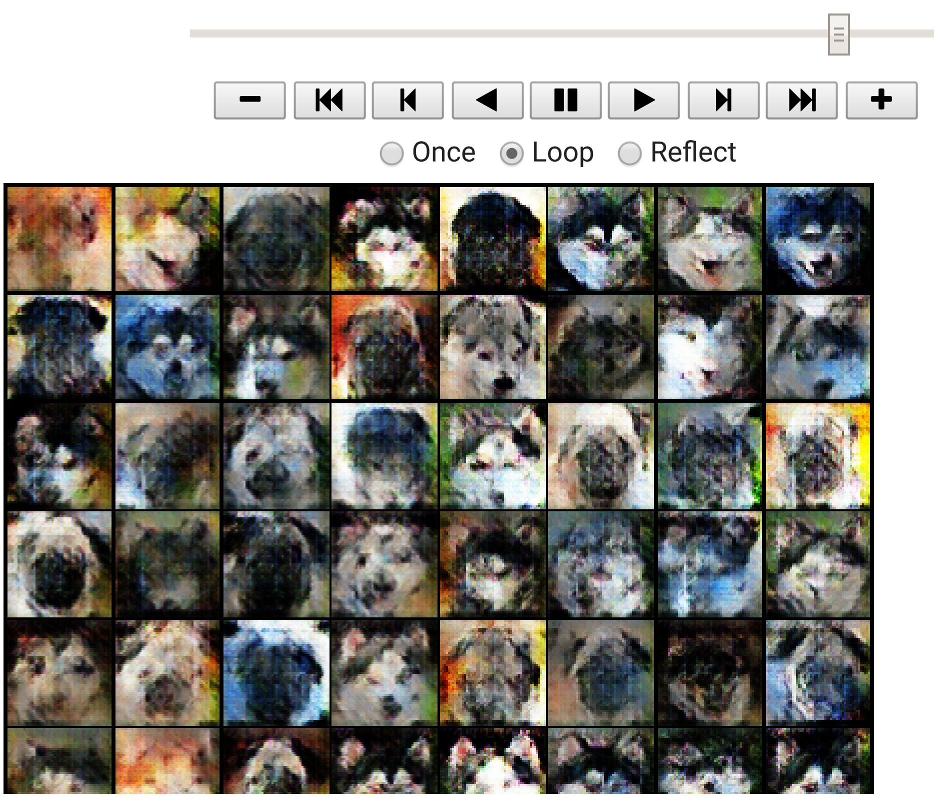
Visualization of G's progression

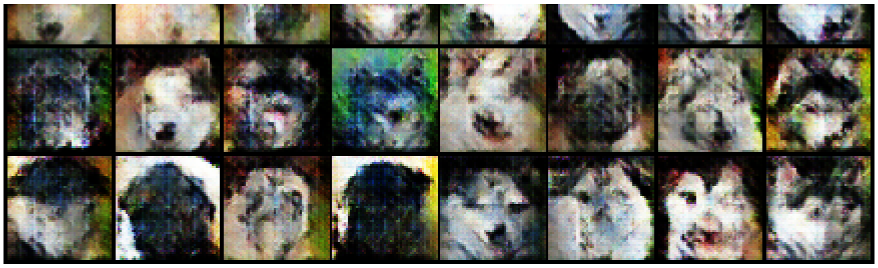
Remember how we saved the generator's output on the fixed_noise batch after every epoch of trainin progression of G with an animation. Press the play button to start the animation.

```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```




Once
Loop
Reflect



Real Images vs. Fake Images

Finally, lets take a look at some real images and fake images side by side.

```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=5, normalize=

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-4],(1,2,0)))
plt.show()
```

Where to Go Next

We have reached the end of our journey, but there are several places you could go from here. You cou

- Train for longer to see how good the results get
- Modify this model to take a different dataset and possibly change the size of the images and th
- Check out some other cool GAN projects `here <https://github.com/nashory/gans-awesome-a`
- Create GANs that generate `music <https://deepmind.com/blog/wavenet-generative-model-ra`