0607_boolgan

September 27, 2020

[ ]:
```
# Don't run again

# !unzip 'thecarconnectionpicturedataset.zip' -d 'dataset/'
```

[ ]:
```
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
from PIL import Image
from fid_score import calculate_fid_given_paths
import time

# Set random seed for reproducibility
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 0x7f62c586f230>
```

[ ]:
```python
dataroot = "dataset/"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# 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 = 50

# Learning rate for optimizers
lr = 0.00075

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

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

[ ]:
```python
# 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
```

```python
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")
print(real_batch[0].to(device)[:64].shape)
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],␣
 ↪padding=2, normalize=True).cpu(),(1,2,0)))
```

```
torch.Size([64, 3, 64, 64])
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f62c17dde80>
```

Training Images

[ ]:
```
for i in range(64):
  vutils.save_image(real_batch[0][i], \
                        os.path.join('sample_real_images_0607',"foo"+str(i)+".
↪jpeg"), \
                        normalize=True)
```

[ ]:
```
print(torch.cuda.is_available())
print(torch.cuda.device_count())
```

```
True
1
```

[ ]:
```python
# 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)
```

[ ]:
```python
# 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 * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.LeakyReLU(True),
            # state size. (ngf*16) x 4 x 4
            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 8 x 8
            nn.ConvTranspose2d( ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 16 x 16
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 32 x 32
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

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

            # state size. (nc) x 128 x 128
            nn.Conv2d(nc, nc * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(nc * 2),
            nn.ReLU(True),

```

```
            # state size. (nc * 2) x 64 x 64
            nn.Conv2d(nc * 2, nc, 1, 1, 0, bias=False),
            nn.Tanh()
        )

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

[ ]:
```
# 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, 1024, kernel_size=(4, 4), stride=(1, 1),
bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): LeakyReLU(negative_slope=True)
    (3): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (10): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (13): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
    (14): ReLU(inplace=True)
    (15): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (16): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(3, 6, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (19): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): Conv2d(6, 3, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (22): Tanh()
  )
)
```

[ ]:
```python
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.Dropout(0.2),
        )

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

[ ]:
```python
# 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): Dropout(p=0.2, inplace=False)
  )
)
```

[ ]:
```python
# 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 = []
fid_values = []
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)
        D_x = output.mean()

        ## 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)
        D_G_z1 = output.mean()#.item()
        errD = D_G_z1 - D_x
        errD.backward()
        # Update D
```

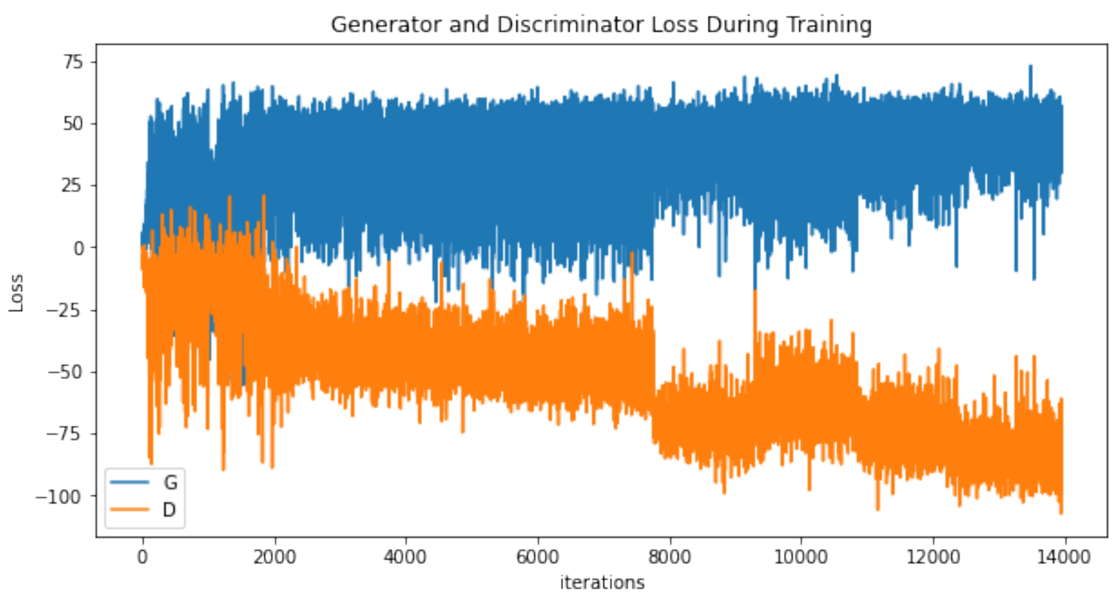
```
        optimizerD.step()
        # Wasserstein weight clipping
        for p in netD.parameters():
          p.data.clamp_(-0.1, 0.1)

        ############################
        # (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
        D_G_z2 = output.mean()
        errG = -D_G_z2
        errG.backward()
        # 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 / %.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 % 500 == 0) or ((epoch == num_epochs-1) and (i ==␣
↪len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
                for i in range(len(fake)):
                  vutils.save_image(fake[i], \
                      os.path.join('sample_fake_images_0607',"fake"+str(i)+".
↪jpeg"), \
                      normalize=True)
                time.sleep(30)
                fid_value = calculate_fid_given_paths(['sample_real_images_0607/
↪', 'sample_fake_images_0607/'], 50, False, 2048)
                fid_values.append(fid_value)
                print("FID Value:", fid_value)
```

```
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
Starting Training Loop…
[0/50][0/155]   Loss_D: -71.5861        Loss_G: 58.2906 D(x): 42.3939   D(G(z)):
-29.1921 / -58.2906
[0/50][50/155]  Loss_D: -91.0381        Loss_G: 57.1893 D(x): 47.8883   D(G(z)):
-43.1498 / -57.1893
[0/50][100/155] Loss_D: -85.4512        Loss_G: 37.8022 D(x): 35.3447   D(G(z)):
-50.1064 / -37.8022

  0%|          | 0/2 [00:00<?, ?it/s]

['sample_real_images_0607/', 'sample_fake_images_0607/']
[PosixPath('sample_real_images_0607/foo0.jpeg'),
PosixPath('sample_real_images_0607/foo1.jpeg'),
PosixPath('sample_real_images_0607/foo2.jpeg'),
PosixPath('sample_real_images_0607/foo3.jpeg'),
PosixPath('sample_real_images_0607/foo4.jpeg'),
PosixPath('sample_real_images_0607/foo5.jpeg'),
PosixPath('sample_real_images_0607/foo6.jpeg'),
PosixPath('sample_real_images_0607/foo7.jpeg'),
PosixPath('sample_real_images_0607/foo8.jpeg'),
PosixPath('sample_real_images_0607/foo9.jpeg'),
PosixPath('sample_real_images_0607/foo10.jpeg'),
PosixPath('sample_real_images_0607/foo11.jpeg'),
PosixPath('sample_real_images_0607/foo12.jpeg'),
PosixPath('sample_real_images_0607/foo13.jpeg'),
PosixPath('sample_real_images_0607/foo14.jpeg'),
PosixPath('sample_real_images_0607/foo15.jpeg'),
PosixPath('sample_real_images_0607/foo16.jpeg'),
PosixPath('sample_real_images_0607/foo17.jpeg'),
PosixPath('sample_real_images_0607/foo18.jpeg'),
PosixPath('sample_real_images_0607/foo19.jpeg'),
PosixPath('sample_real_images_0607/foo20.jpeg'),
PosixPath('sample_real_images_0607/foo21.jpeg'),
PosixPath('sample_real_images_0607/foo22.jpeg'),
PosixPath('sample_real_images_0607/foo23.jpeg'),
PosixPath('sample_real_images_0607/foo24.jpeg'),
PosixPath('sample_real_images_0607/foo25.jpeg'),
PosixPath('sample_real_images_0607/foo26.jpeg'),
PosixPath('sample_real_images_0607/foo27.jpeg'),
PosixPath('sample_real_images_0607/foo28.jpeg'),
PosixPath('sample_real_images_0607/foo29.jpeg'),
PosixPath('sample_real_images_0607/foo30.jpeg'),
PosixPath('sample_real_images_0607/foo31.jpeg'),
PosixPath('sample_real_images_0607/foo32.jpeg'),
```

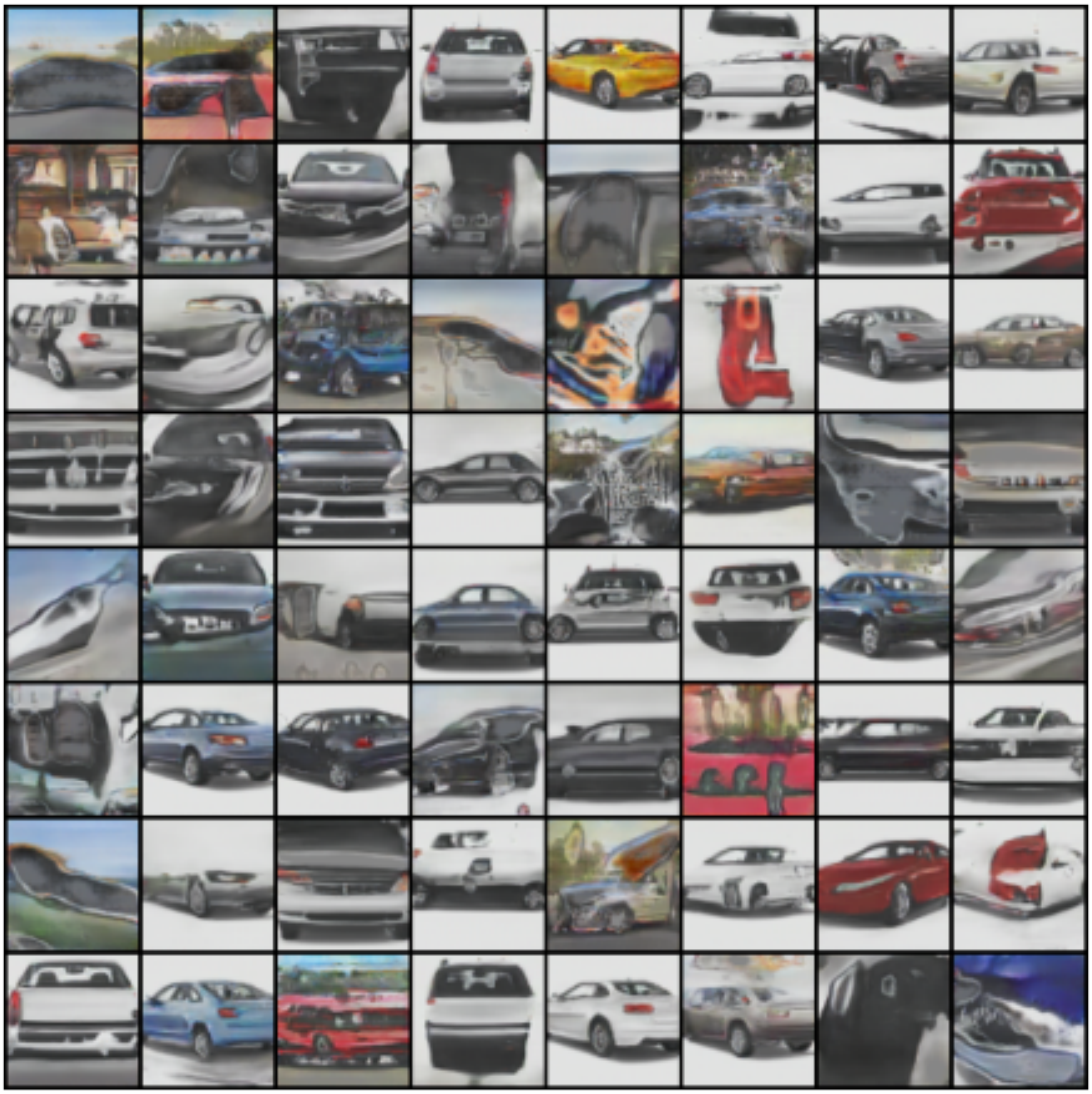
```
-52.4274 / -40.9825
[8/50][0/155]   Loss_D: -68.9025        Loss_G: 55.1833 D(x): 57.6392   D(G(z)):
-11.2632 / -55.1833
[8/50][50/155]  Loss_D: -84.7135        Loss_G: 59.3480 D(x): 61.8538   D(G(z)):
-22.8597 / -59.3480
[8/50][100/155] Loss_D: -72.1132        Loss_G: 51.5793 D(x): 60.2030   D(G(z)):
-11.9102 / -51.5793
[8/50][150/155] Loss_D: -73.7943        Loss_G: 27.5061 D(x): 19.4245   D(G(z)):
-54.3699 / -27.5061
[9/50][0/155]   Loss_D: -64.5925        Loss_G: 48.1603 D(x): 49.0829   D(G(z)):
-15.5096 / -48.1603
[9/50][50/155]  Loss_D: -83.8324        Loss_G: 37.4948 D(x): 36.9237   D(G(z)):
-46.9087 / -37.4948
[9/50][100/155] Loss_D: -69.9681        Loss_G: 58.1766 D(x): 58.7819   D(G(z)):
-11.1862 / -58.1766
[9/50][150/155] Loss_D: -60.9230        Loss_G: 40.0866 D(x): 10.3662   D(G(z)):
-50.5568 / -40.0866
```

[ ]:
```
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()
```



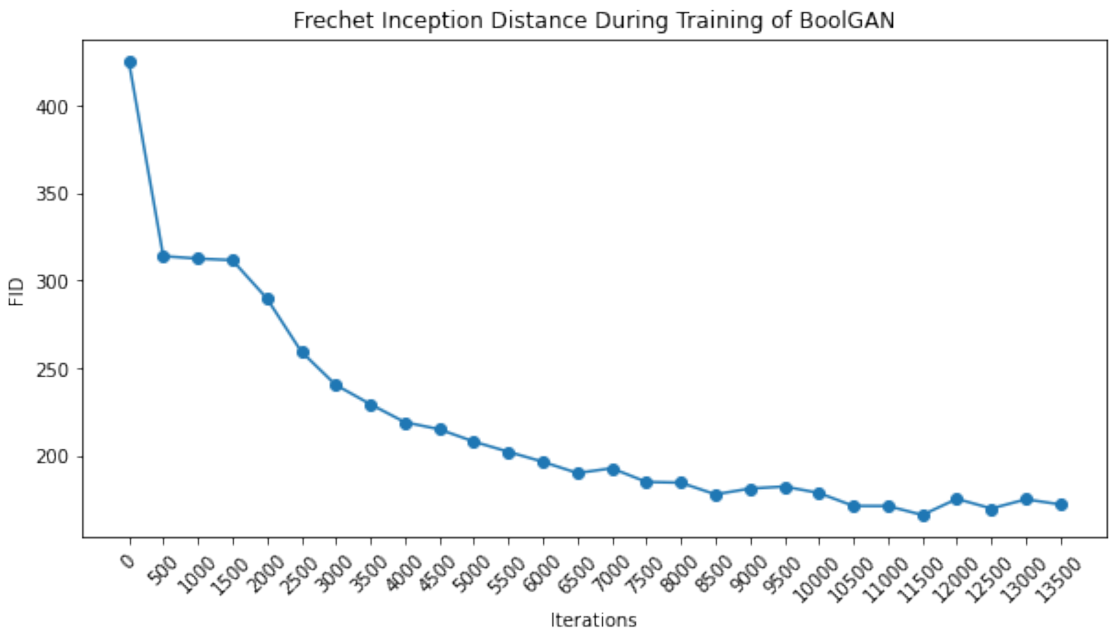
[ ]:
```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,␣
↪blit=True)

HTML(ani.to_jshtml())
```

[ ]: <IPython.core.display.HTML object>



[ ]:
```
plt.figure(figsize=(10,5))
plt.title("Frechet Inception Distance During Training of BoolGAN")
```

```python
plt.plot(fid_values, marker='o')
plt.xlabel("Iterations")
labelList = range(0, len(fid_values)*500, 500)
plt.xticks(ticks=range(0, len(fid_values)), labels=labelList, rotation=45)
plt.ylabel("FID")
plt.show()
```

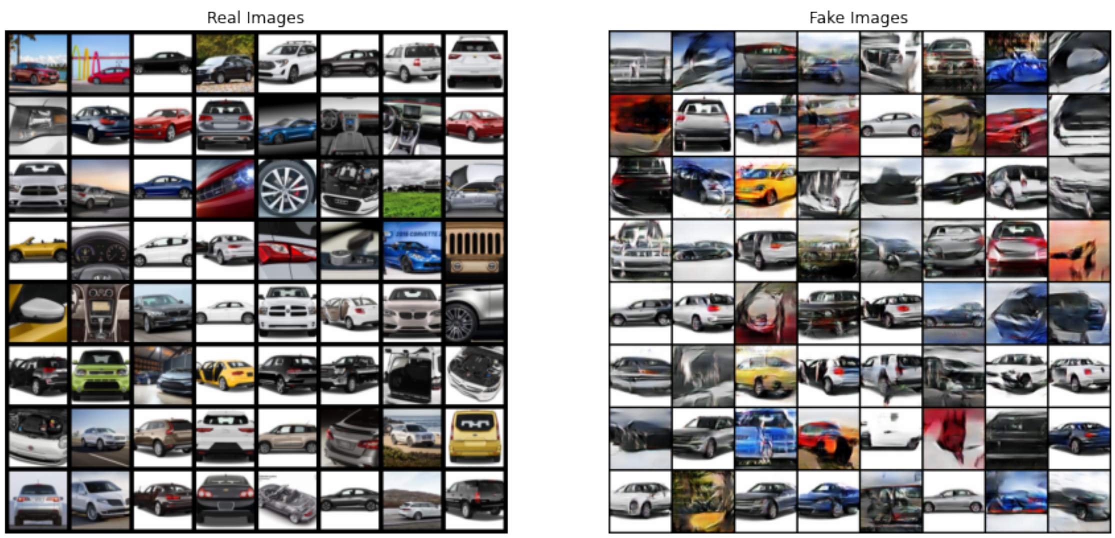


[ ]:
```python
print(fid_values)
```

```
[425.4186359162443, 313.9360441648118, 312.52982531308646, 311.64497167846264,
289.8353060933684, 259.374964764839, 240.29331157594982, 229.3073117959616,
219.0035415450701, 214.94912749317803, 207.9029070554729, 202.0819681655023,
196.3733133334875, 189.92914427573209, 192.65490340057005, 184.8520162324376,
184.41251005806075, 177.73243766373858, 181.0522812208194, 182.22659900846674,
178.5358250389574, 171.12284094889984, 171.0830520343335, 165.9662036438666,
175.17854331246605, 169.64499489759885, 174.91007214849043, 171.98485570346128]
```

[ ]:
```python
# 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=True).cpu(),(1,2,0)))

# 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[-1],(1,2,0)))
plt.show()
```



[ ]:

```

```