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

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

```
%matplotlib inline
```

```
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

# Set random seem 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 0x7f403a2e52d0>
```

```
drive = '/content/sample_data/square_data/squares/'
for i in range(5000):
    x = random.randrange(0,10)
    img = np.zeros((64,64,3), np.uint8)
    color = (random.randrange(1,255), random.randrange(1,255), random.randrange(1,255))

    img[32-x:32+x, 32-x:32+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 size is desired, the structures of D and G must be changed. See `here <https://github.com/pytorch/examples/issues/70>`__ for more details
- **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 also take much longer
- **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 great run on that number of GPUs

```
# Root directory for dataset
dataroot = "/content/sample_data/square_data"

# Number of workers for dataloader
workers = 4

# Batch size during training
batch_size = 64

# 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 = 60

# Learning rate for optimizers
lr = 0.001

# Beta1 hyperparam for Adam optimizers
```

```
beta1 = 0.5

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

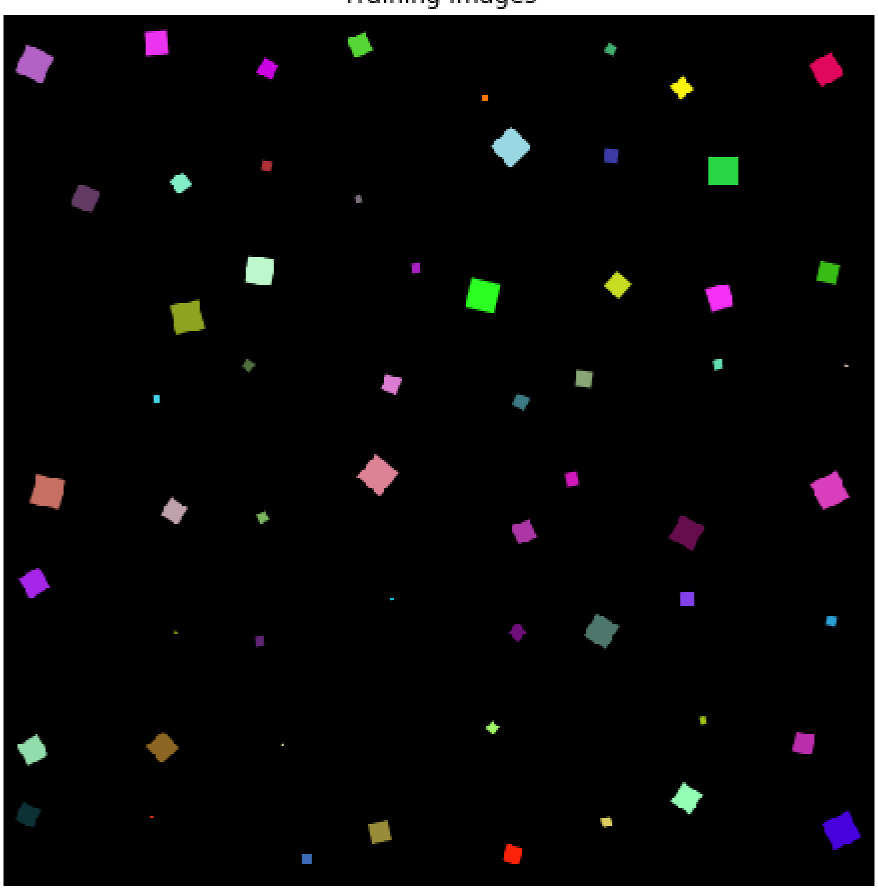
```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               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=True).c
```

```
<matplotlib.image.AxesImage at 0x7f4084e20898>
```

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 the generator object is 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 structure.

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

        iters += 1
```

```
[20/60][50/79]  Loss_D: 1.4951  Loss_G: 0.7221  D(x): 0.3653   D(G(z)): 0.2892 / 0.5438
[21/60][0/79]   Loss_D: 2.0199  Loss_G: 1.5116  D(x): 0.9083   D(G(z)): 0.7975 / 0.2521
[21/60][50/79]  Loss_D: 1.2511  Loss_G: 1.4097  D(x): 0.8450   D(G(z)): 0.6402 / 0.2637
[22/60][0/79]   Loss_D: 2.0815  Loss_G: 2.3464  D(x): 0.8709   D(G(z)): 0.8087 / 0.1175
[22/60][50/79]  Loss_D: 1.2457  Loss_G: 1.5257  D(x): 0.6614   D(G(z)): 0.5218 / 0.2515
[23/60][0/79]   Loss_D: 1.4509  Loss_G: 1.4848  D(x): 0.8800   D(G(z)): 0.6792 / 0.2593
[23/60][50/79]  Loss_D: 0.9505  Loss_G: 1.1233  D(x): 0.5841   D(G(z)): 0.3002 / 0.3478
[24/60][0/79]   Loss_D: 1.5079  Loss_G: 1.7092  D(x): 0.9072   D(G(z)): 0.7195 / 0.2453
[24/60][50/79]  Loss_D: 1.1446  Loss_G: 0.6574  D(x): 0.4374   D(G(z)): 0.2374 / 0.5424
[25/60][0/79]   Loss_D: 2.7688  Loss_G: 0.9737  D(x): 0.9293   D(G(z)): 0.8896 / 0.4313
[25/60][50/79]  Loss_D: 1.2217  Loss_G: 1.3049  D(x): 0.4208   D(G(z)): 0.2370 / 0.2869
[26/60][0/79]   Loss_D: 3.3645  Loss_G: 0.6042  D(x): 0.9731   D(G(z)): 0.9400 / 0.6066
[26/60][50/79]  Loss_D: 1.2905  Loss_G: 1.1599  D(x): 0.5157   D(G(z)): 0.4129 / 0.3615
[27/60][0/79]   Loss_D: 1.5698  Loss_G: 1.4661  D(x): 0.8260   D(G(z)): 0.6743 / 0.2817
[27/60][50/79]  Loss_D: 1.3573  Loss_G: 2.9691  D(x): 0.8115   D(G(z)): 0.6475 / 0.0610
[28/60][0/79]   Loss_D: 2.6460  Loss_G: 1.8834  D(x): 0.9850   D(G(z)): 0.8910 / 0.1909
[28/60][50/79]  Loss_D: 1.6447  Loss_G: 3.1307  D(x): 0.9376   D(G(z)): 0.7692 / 0.0492
[29/60][0/79]   Loss_D: 1.6191  Loss_G: 2.7252  D(x): 0.7974   D(G(z)): 0.6507 / 0.1024
[29/60][50/79]  Loss_D: 1.3014  Loss_G: 0.9880  D(x): 0.3335   D(G(z)): 0.0545 / 0.4108
[30/60][0/79]   Loss_D: 0.8916  Loss_G: 4.9544  D(x): 0.9233   D(G(z)): 0.5092 / 0.0087
[30/60][50/79]  Loss_D: 0.4139  Loss_G: 2.6281  D(x): 0.8747   D(G(z)): 0.2275 / 0.0873
[31/60][0/79]   Loss_D: 1.7637  Loss_G: 2.2514  D(x): 0.8812   D(G(z)): 0.7436 / 0.1495
[31/60][50/79]  Loss_D: 0.4079  Loss_G: 2.4651  D(x): 0.7282   D(G(z)): 0.0678 / 0.1001
[32/60][0/79]   Loss_D: 0.6146  Loss_G: 0.7553  D(x): 0.6417   D(G(z)): 0.1217 / 0.5311
[32/60][50/79]  Loss_D: 0.1896  Loss_G: 3.1153  D(x): 0.9523   D(G(z)): 0.1264 / 0.0528
[33/60][0/79]   Loss_D: 1.6854  Loss_G: 3.0538  D(x): 0.8678   D(G(z)): 0.7134 / 0.0717
[33/60][50/79]  Loss_D: 0.5339  Loss_G: 5.1051  D(x): 0.9929   D(G(z)): 0.3859 / 0.0072
[34/60][0/79]   Loss_D: 0.4652  Loss_G: 3.1988  D(x): 0.8242   D(G(z)): 0.1858 / 0.0570
[34/60][50/79]  Loss_D: 0.2115  Loss_G: 3.2222  D(x): 0.9098   D(G(z)): 0.0680 / 0.0538
[35/60][0/79]   Loss_D: 0.5568  Loss_G: 2.0906  D(x): 0.9084   D(G(z)): 0.3163 / 0.1545
[35/60][50/79]  Loss_D: 5.1111  Loss_G: 2.8137  D(x): 0.0195   D(G(z)): 0.0003 / 0.1043
[36/60][0/79]   Loss_D: 0.2795  Loss_G: 4.4470  D(x): 0.9940   D(G(z)): 0.2042 / 0.0178
[36/60][50/79]  Loss_D: 0.1687  Loss_G: 3.6446  D(x): 0.8880   D(G(z)): 0.0350 / 0.0466
[37/60][0/79]   Loss_D: 0.4623  Loss_G: 2.6764  D(x): 0.7381   D(G(z)): 0.0827 / 0.1034
[37/60][50/79]  Loss_D: 0.2594  Loss_G: 4.6265  D(x): 0.7939   D(G(z)): 0.0019 / 0.0150
[38/60][0/79]   Loss_D: 0.0417  Loss_G: 4.1340  D(x): 0.9848   D(G(z)): 0.0257 / 0.0207
[38/60][50/79]  Loss_D: 0.0613  Loss_G: 4.7990  D(x): 0.9868   D(G(z)): 0.0423 / 0.0128
[39/60][0/79]   Loss_D: 0.0126  Loss_G: 4.5422  D(x): 0.9990   D(G(z)): 0.0114 / 0.0161
[39/60][50/79]  Loss_D: 0.5382  Loss_G: 2.0068  D(x): 0.6838   D(G(z)): 0.0692 / 0.1815
[40/60][0/79]   Loss_D: 0.1843  Loss_G: 4.9301  D(x): 0.9713   D(G(z)): 0.1123 / 0.0147
[40/60][50/79]  Loss_D: 0.0114  Loss_G: 6.9595  D(x): 0.9983   D(G(z)): 0.0095 / 0.0026
[41/60][0/79]   Loss_D: 2.7421  Loss_G: 5.6773  D(x): 0.9543   D(G(z)): 0.8748 / 0.0198
[41/60][50/79]  Loss_D: 0.1682  Loss_G: 5.2652  D(x): 0.8612   D(G(z)): 0.0027 / 0.0082
[42/60][0/79]   Loss_D: 0.1054  Loss_G: 2.9872  D(x): 0.9965   D(G(z)): 0.0946 / 0.0626
[42/60][50/79]  Loss_D: 0.2524  Loss_G: 3.1311  D(x): 0.8583   D(G(z)): 0.0189 / 0.0583
[43/60][0/79]   Loss_D: 4.2940  Loss_G: 9.1384  D(x): 0.9991   D(G(z)): 0.9634 / 0.0003
[43/60][50/79]  Loss_D: 0.9087  Loss_G: 4.4272  D(x): 0.9725   D(G(z)): 0.4980 / 0.0178
[44/60][0/79]   Loss_D: 0.2963  Loss_G: 3.1727  D(x): 0.9064   D(G(z)): 0.1285 / 0.0690
[44/60][50/79]  Loss_D: 0.2790  Loss_G: 3.5738  D(x): 0.8128   D(G(z)): 0.0558 / 0.0438
[45/60][0/79]   Loss_D: 0.1481  Loss_G: 3.9010  D(x): 0.9132   D(G(z)): 0.0422 / 0.0468
[45/60][50/79]  Loss_D: 0.0852  Loss_G: 4.6652  D(x): 0.9938   D(G(z)): 0.0742 / 0.0132
[46/60][0/79]   Loss_D: 1.6991  Loss_G: 8.7159  D(x): 0.9984   D(G(z)): 0.7377 / 0.0004
[46/60][50/79]  Loss_D: 0.8390  Loss_G: 4.4246  D(x): 0.9862   D(G(z)): 0.5092 / 0.0215
[47/60][0/79]   Loss_D: 4.9910  Loss_G: 4.1055  D(x): 0.9999   D(G(z)): 0.9751 / 0.0324
[47/60][50/79]  Loss_D: 0.6508  Loss_G: 3.2114  D(x): 0.9895   D(G(z)): 0.4469 / 0.0477
[48/60][0/79]   Loss_D: 2.8913  Loss_G: 7.9553  D(x): 0.9998   D(G(z)): 0.9029 / 0.0007
[48/60][50/79]  Loss_D: 0.1558  Loss_G: 3.5601  D(x): 0.9295   D(G(z)): 0.0748 / 0.0406
[49/60][0/79]   Loss_D: 0.5910  Loss_G: 4.2077  D(x): 1.0000   D(G(z)): 0.3580 / 0.0241
```

```
[49/60][50/79]  Loss_D: 0.1235  Loss_G: 3.4171  D(x): 0.9291    D(G(z)): 0.0383 / 0.0412
[50/60][0/79]   Loss_D: 4.3362  Loss_G: 3.9284  D(x): 0.9998    D(G(z)): 0.9172 / 0.0920
[50/60][50/79]  Loss_D: 0.1021  Loss_G: 4.4575  D(x): 0.9825    D(G(z)): 0.0763 / 0.0172
[51/60][0/79]   Loss_D: 0.6642  Loss_G: 3.2043  D(x): 0.8651    D(G(z)): 0.3584 / 0.0626
[51/60][50/79]  Loss_D: 2.3933  Loss_G: 0.7965  D(x): 0.1250    D(G(z)): 0.0004 / 0.5122
[52/60][0/79]   Loss_D: 0.1220  Loss_G: 4.1970  D(x): 0.9808    D(G(z)): 0.0951 / 0.0202
[52/60][50/79]  Loss_D: 0.1891  Loss_G: 8.3906  D(x): 0.8488    D(G(z)): 0.0001 / 0.0005
[53/60][0/79]   Loss_D: 0.0065  Loss_G: 7.2400  D(x): 0.9949    D(G(z)): 0.0013 / 0.0012
[53/60][50/79]  Loss_D: 0.0918  Loss_G: 4.4401  D(x): 0.9776    D(G(z)): 0.0592 / 0.0192
[54/60][0/79]   Loss_D: 4.1998  Loss_G: 6.7730  D(x): 1.0000    D(G(z)): 0.8662 / 0.0017
[54/60][50/79]  Loss_D: 0.0901  Loss_G: 5.4064  D(x): 0.9909    D(G(z)): 0.0679 / 0.0081
[55/60][0/79]   Loss_D: 0.3984  Loss_G: 5.8680  D(x): 0.9924    D(G(z)): 0.2578 / 0.0041
[55/60][50/79]  Loss_D: 0.3112  Loss_G: 4.1508  D(x): 0.7506    D(G(z)): 0.0031 / 0.0313
[56/60][0/79]   Loss_D: 2.0480  Loss_G: 8.0220  D(x): 0.9211    D(G(z)): 0.7479 / 0.0008
[56/60][50/79]  Loss_D: 0.2949  Loss_G: 3.0195  D(x): 0.9773    D(G(z)): 0.2043 / 0.0651
[57/60][0/79]   Loss_D: 0.1841  Loss_G: 3.6742  D(x): 0.9251    D(G(z)): 0.0599 / 0.0691
[57/60][50/79]  Loss_D: 0.1021  Loss_G: 4.3927  D(x): 0.9642    D(G(z)): 0.0615 / 0.0182
[58/60][0/79]   Loss_D: 0.0674  Loss_G: 4.5956  D(x): 0.9951    D(G(z)): 0.0568 / 0.0175
[58/60][50/79]  Loss_D: 0.0491  Loss_G: 5.4625  D(x): 0.9969    D(G(z)): 0.0440 / 0.0067
[59/60][0/79]   Loss_D: 2.1525  Loss_G: 7.6566  D(x): 0.9996    D(G(z)): 0.7589 / 0.0012
[59/60][50/79]  Loss_D: 0.0310  Loss_G: 4.2983  D(x): 0.9979    D(G(z)): 0.0280 / 0.0181
```

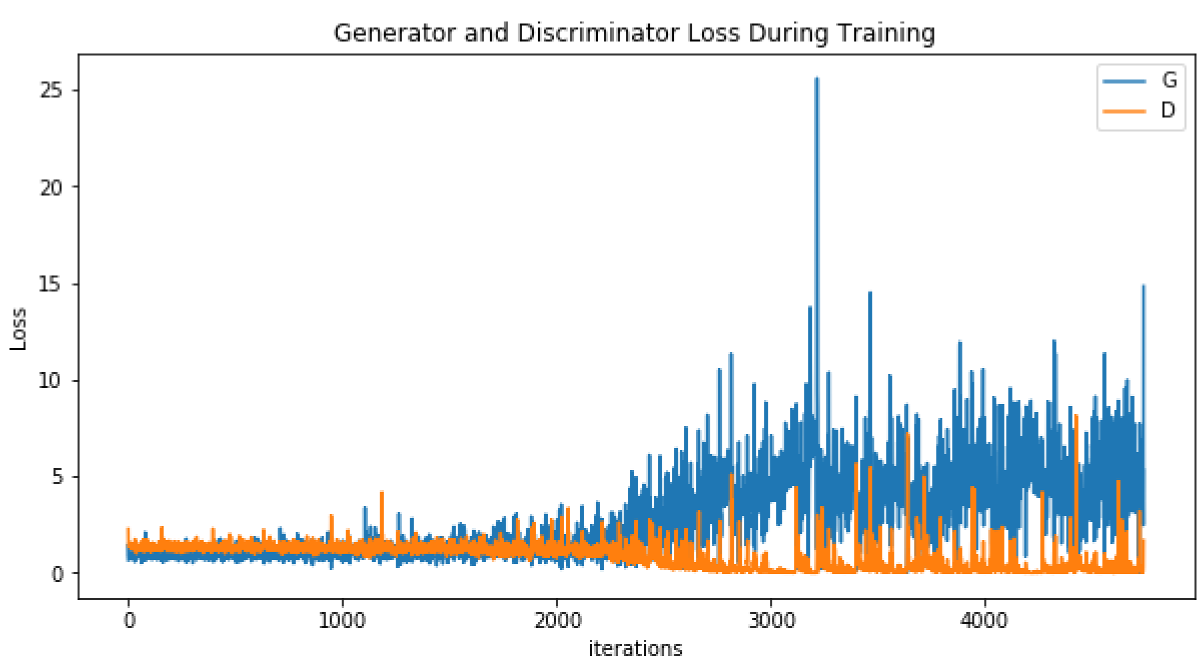
Results

Finally, lets check out how we did. Here, we will look at three different results. First, we will see how D
changed during training. Second, we will visualize G's output on the fixed_noise batch for every epoch
will look at a batch of real data next to a batch of 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()
```

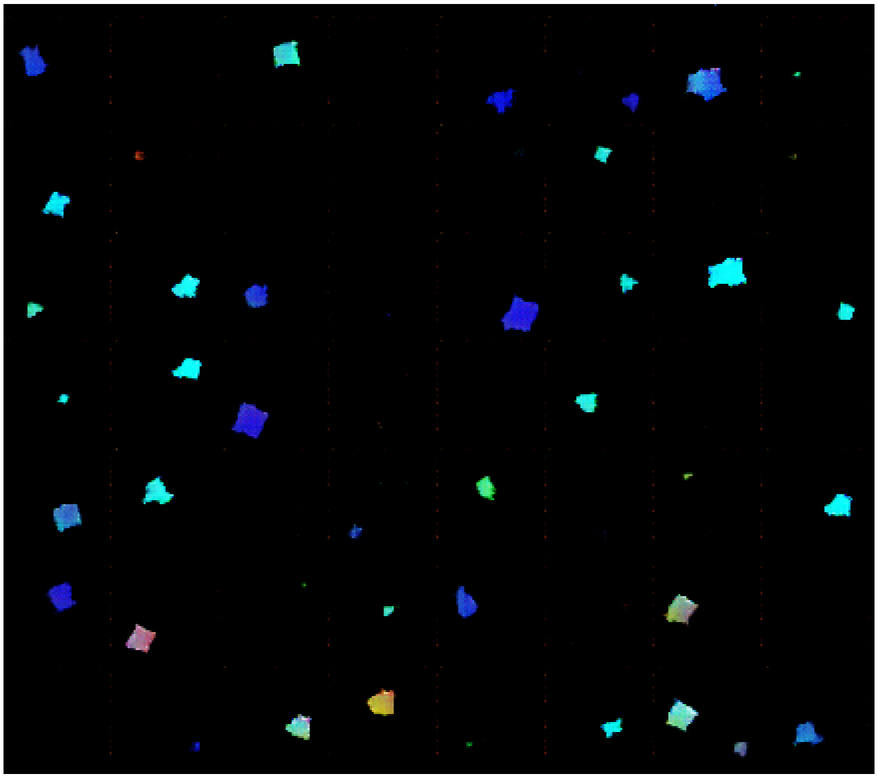


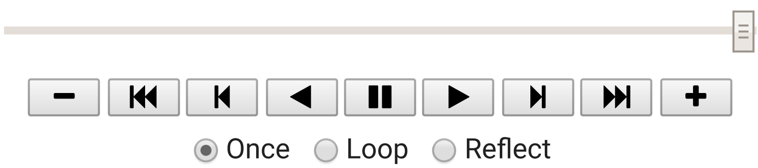
Visualization of G's progression

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

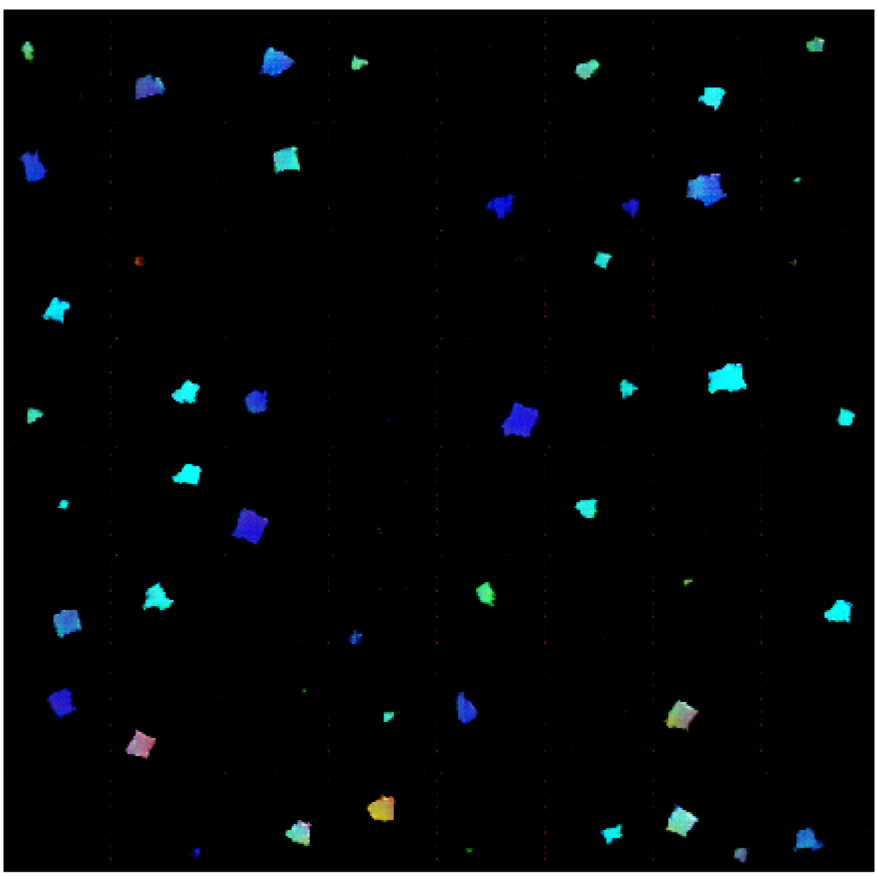
```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

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=True).c

# 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()
```

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 architecture
- 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`