Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says YOUR CODE HERE or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = "Nopphawan Nurnuansuwan"
ID = "122410"
```

Lab 12: Generative Adversarial Networks

We have seen that in the unsupervised learning setting, we are given a training dataset $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})$, $\mathbf{x}^{(i)} \in \mathcal{X}$. and our goal is to output a model that expresses the structure of the dataset.

Many unsupervised learning algorithms such as the EM algorithm for Gaussian mixture models estimate a probability density model $h_\theta(\mathbf{x}) = p(\mathbf{X} = \mathbf{x}; \theta)$. The GMM is an explicit parametric probability density estimator.

A completely different approach to probability density modeling is represented by the recently introduced Generative Adversarial Network (GAN) density model. GANs were introduced in 2014 by Ian Goodfellow and colleagues (https://arxiv.org/abs/1406.2661).

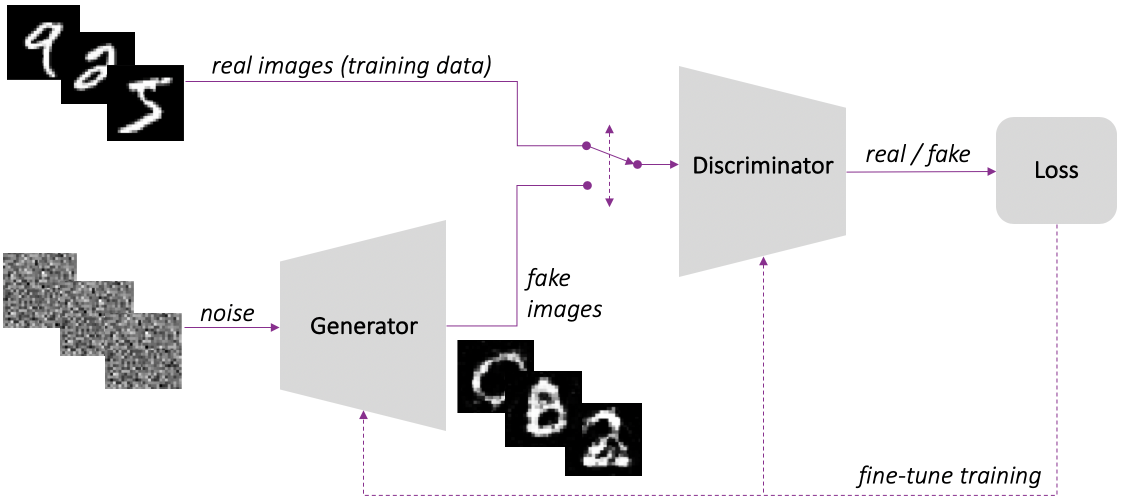
The idea is simple. We assume that the training data were sampled i.i.d. from a distribution $p_{\text{data}}(\mathbf{x})$ over $\mathcal{X}$. We define a prior density $p_{\mathbf{z}}(\mathbf{z})$ over a set of possible "noise vectors" $\mathcal{Z}$, (usually, $\mathcal{Z} = \mathbb{R}^d$), then we try to come up with a neural network *generator* $G(\mathbf{z}; \theta_g)$ whose goal is to transform noise inputs $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$ into outputs $\mathbf{x} = G(\mathbf{z}; \theta_g)$ distributed (as closely as possible) according to $p_{\text{data}}(\mathbf{x})$. We also create and train a neural network *discriminator* $D(\mathbf{x}; \theta_d)$ whose goal is to distinguish "real" samples from the training set from "fake" samples from G.

Formally, G and D play a minimax game with value function $V(D, G)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} \left[\log(1 - D(G(\mathbf{z}))\right].$$

Again, the generator's goal is to "trick" the discriminator into thinking its outputs are samples from $p_{\text{data}}(\mathbf{x})$. The generator *implicitly* defines a probability density $p_g(\mathbf{x})$ over $\mathcal{X}$. The generator wins the minimax game when $p_g = p_{\text{data}}$.

One type of application of the GAN is to generate images according to a target distribution. Goodfellow's original GAN paper (https://arxiv.org/abs/1406.2661) demonstrates the ability of their GANs to generate images similar to those in the MNIST handwritten digit dataset and the CIFAR-10 common object image dataset. Here is the basic GAN model described by Goodfellow et al. (2014):



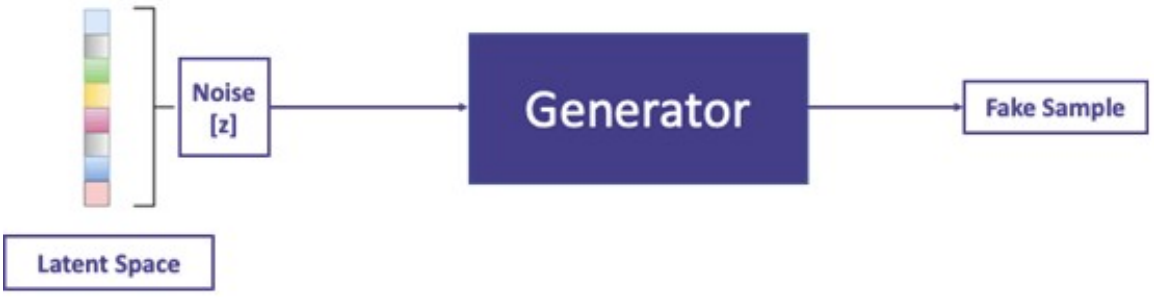
Since 2014, GANs and their cousins have led to some of the most startling advances artificial intelligence has seen during this time. See, for example, our favorite GAN variant, the CycleGAN (https://arxiv.org/abs/1703.10593), which combines two generators and two discriminators and can do incredibly amazing things like turn horses into zebras (https://junyanz.github.io/CycleGAN/) or super-resolve images of peoples' faces (https://link.springer.com/article/10.1007%2Fs00521-021-05973-0).

In this lab, we'll develop several basic GANs and experiment with them. Some of the material here is derived from Coursera's *Build Basic Generative Adversarial Networks*.

After this lab, you may be interested in 6 GAN Architectures You Really Should Know (https://neptune.ai/blog/6-gan-architectures).

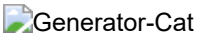
Generator

The generator in a GAN is driven by a noise vector sampled from the "latent space" $\mathcal{Z}$ according to $p_{\mathbf{z}}$ and transforms that noise sample into an element of $\mathcal{X}$, the domain of p_{data}.



The generator in a GAN is used to generate examples and is the model we're invested in and helping to achieve high performance at the end of the training process.

The generator's final goal is to produce examples from a certain class. So if you trained it from the class of a cat, then the generator will do some computations and output a representation of a cat that looks as real as possible.



Clearly, yhe generator should not output the same cat every time it runs. To ensure that it is able to produce different examples each time it is invoked, we actually input different samples from the noise distribution.

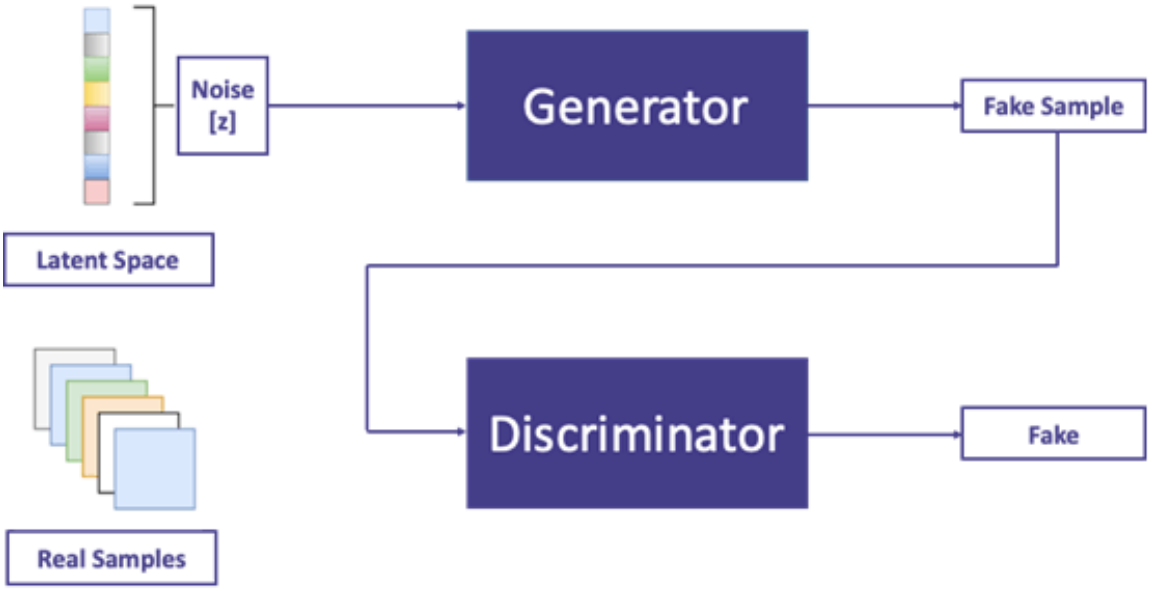
Noise vector is actually just a set of values where these differently shaded cells are just different values. So you can think of this as 1, 2, 5, 1.5, 5, 5, 2. Then this noise vector is fed in as input, sometimes with additional information such as a class y for the class "cat" into the generator's neural network. This means

that these features, x_0, x_1, x_2, all the way up to x_n, include the class, as well as, the numbers in this noise vector. Then the generator in this neural network will compute a series of nonlinearities from those inputs and return some variables that look link an image.

In another run, it may generate a cat or a dog or even a horse. These are all with a different noise vectors and each noise vector can be red nose, short hair and other. These things can be learned from Discriminator model.

Discriminator

The discriminator has the responsibity to classify its input as real or fake. When a fake sample from the generator is given, it should ouptut 0 for fake:



On the other hand, if the input is real, it shoudl output 1 for real:

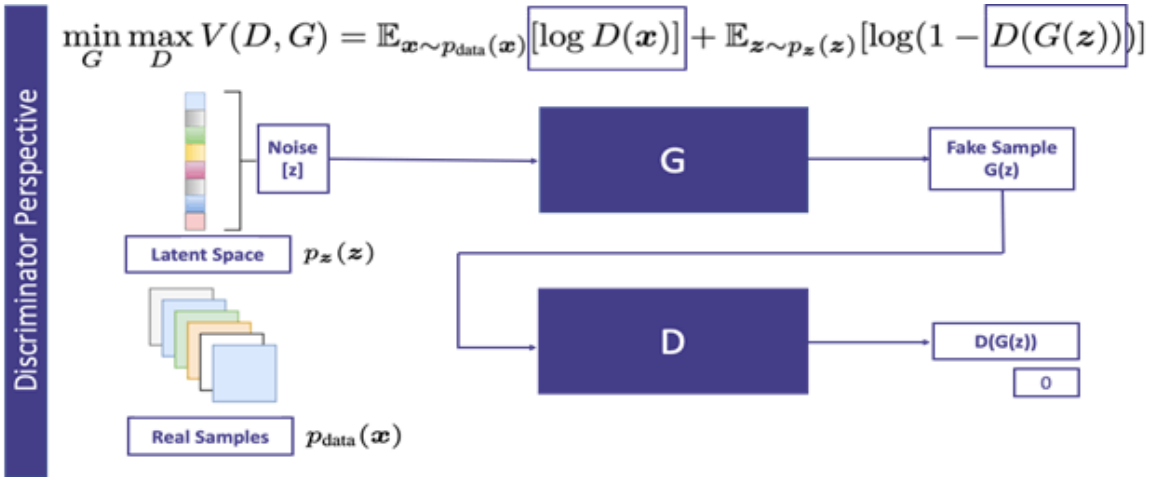
Discriminator models is the probability of an example being fake given a set of input X. It will look at the image of fake cats and determined that they are about 80% probability it isn't the real one, so it will classify as **FAKE**.

On the other hand, it will look at another cat image and determined that they are about 5% probability it isn't the real one, so it will classify as **REAL**.

It uses the probability to feedback to the generator models, then generator model will learn from the punishment.

The optimizer

As explained above, the optimization is a minimax game. The generator wants to minimize the objective function, whereas the discriminator wants to maximize the same objective function.



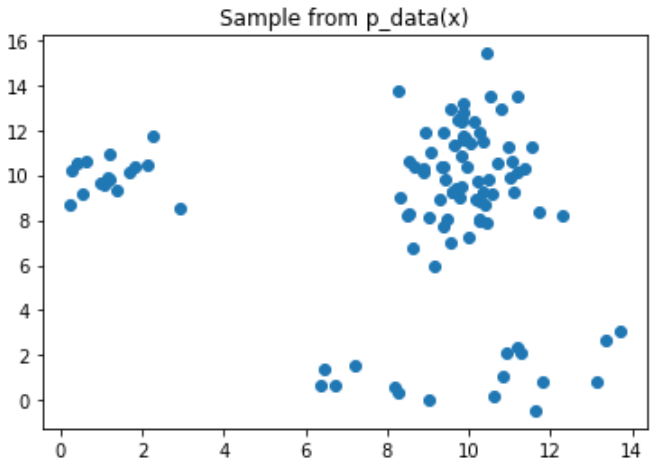
Example 1: Generate a mixture of Gaussians

Suppose we have an unknwon distribution $p_{\text{data}}(\mathbf{x})$ that is in fact a mixture of three Gaussian distributions:

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

# Sample from p_data(x):
def sample_pdata(m):
    means_gt = [ [1,10], [10,1], [10,10] ]
    sigmas_gt = [ np.matrix([[1, 0],[0, 1]]), np.matrix([[4,0],[0,1]]),
                  np.matrix([[1,0],[0,4]]) ]
    phi_gt = [ 0.2, 0.2, 0.6 ]
    n = len(means_gt[0])
    k = len(phi_gt)
    Z = [0]*m
    X = np.zeros((m,n))
    # Generate m samples from multinomial distribution using phi_gt
    z_vectors = np.random.multinomial(1, phi_gt, size=m)  # Result: binary matrix of size (m x k)
    for i in range(m):
        # Convert one-hot representation z_vectors[i,:] to an index
        Z[i] = np.where(z_vectors[i,:] == 1)[0][0]
        # Grab ground truth mean mu_{z^i}
        mu = means_gt[Z[i]]
        # Grab ground truth covariance Sigma_{z^i}
        sigma = sigmas_gt[Z[i]]
        # Sample a 2D point from mu, sigma
        X[i,:] = np.random.multivariate_normal(mu,sigma,1)
    return X
```

Let's generate a sample from this ground truth distribution:

In [3]:
```
X = sample_pdata(100)

plt.scatter(X[:,0],X[:,1])
plt.title('Sample from p_data(x)')
plt.show()
```

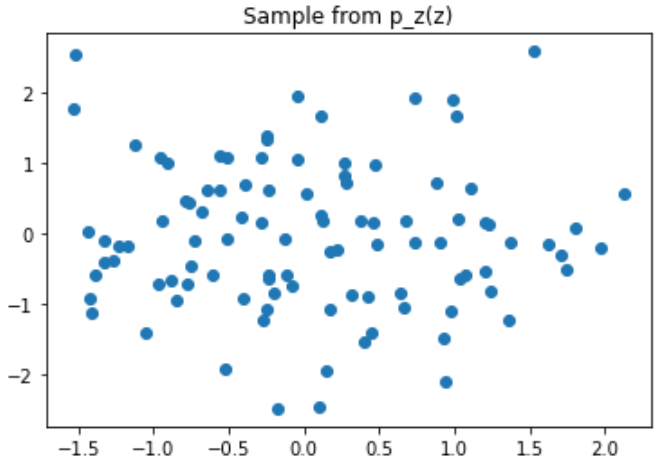


Next we need a function to sample from the noise distribution:

In [4]:
```
def sample_noise(m, n):
    return np.random.multivariate_normal([0,0],[[1, 0],[0, 1]], m)
```

Let's get a sample from the noise distribution:

In [5]:
```
Z = sample_noise(100, 2)

plt.scatter(Z[:,0],Z[:,1])
plt.title('Sample from p_z(z)')
plt.show()
```



Next, let's define a discriminator and generator:

In [6]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class GeneratorNet(nn.Module):
    def __init__(self):
        super(GeneratorNet, self).__init__()
        # First fully connected layer
        self.fc1 = nn.Linear(2, 20)
        # Second fully connected layer
        self.fc2 = nn.Linear(20, 20)
        self.output = nn.Linear(20, 2)

    def forward(self, x):
        # Pass data through fc1
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        output = self.output(x)
        return output

class DiscriminatorNet(nn.Module):
    def __init__(self):
        super(DiscriminatorNet, self).__init__()
        # First fully connected layer
        self.fc1 = nn.Linear(2, 20)
        # Second fully connected layer
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20,20)
        self.output = nn.Linear(20, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.output(x)
        return F.sigmoid(x)
```

Let's create instances of the generator and discriminator and test that G() can process a sample from the noise distribution and that D() can process a sample from the data distribution or the output of the generator:

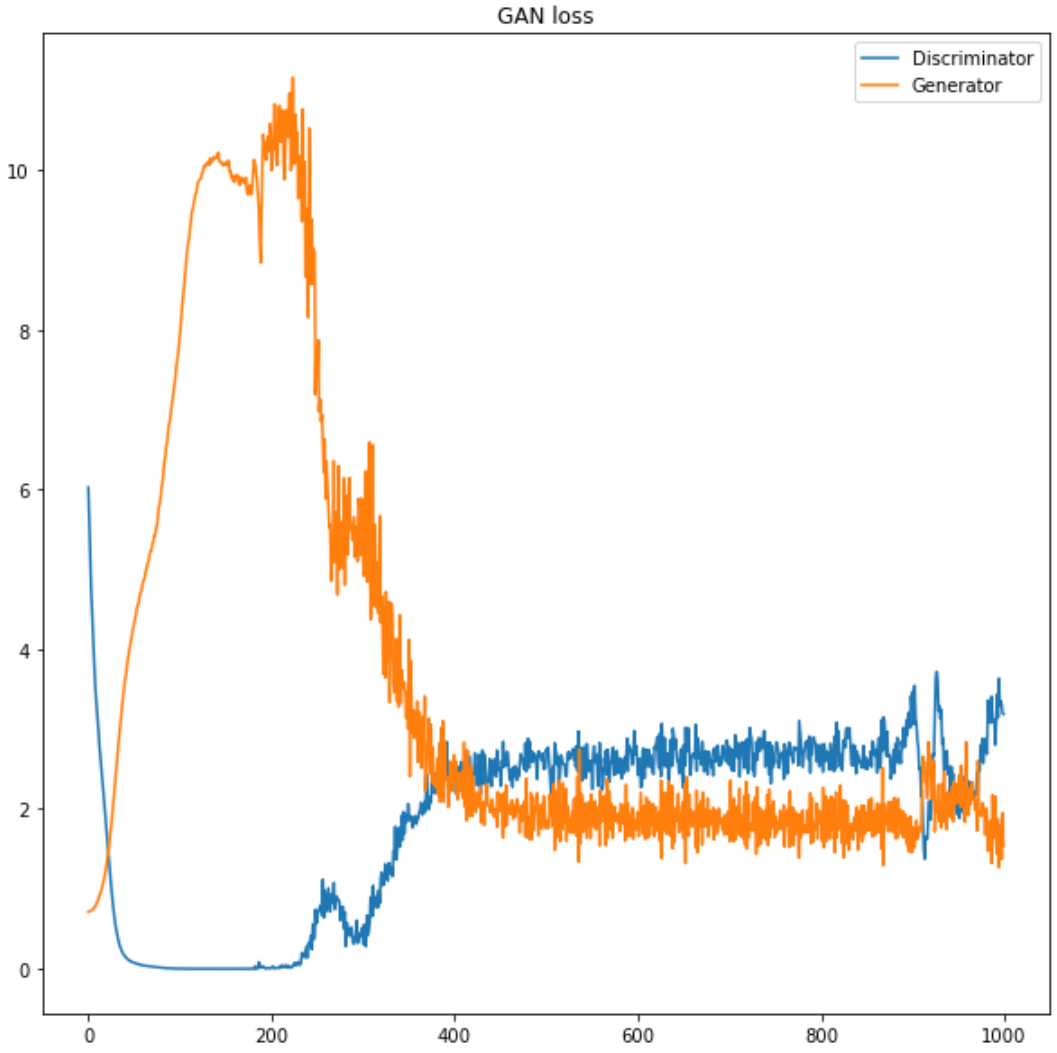
In [7]:
```
# Instantiate the generator and discriminator

G = GeneratorNet()
D = DiscriminatorNet()

# xhat = G(noise sample)

z = torch.tensor(sample_noise(10, 2)).float()
print('Generator input:', z)
xhat = G(z)
print('Generator output:', xhat)

# decisions on fake data = D(G(noise sample))

decisions_fake = D(xhat)
print('Discriminator output for generated data:', decisions_fake)

# decisions on real data = D(data sample)

x = torch.tensor(sample_pdata(10)).float()
decisions_real = D(x)
print('Discriminator output for real data:', decisions_real)
```

```
Generator input: tensor([[ 0.0494, -0.3717],
        [-0.7428,  2.3857],
        [-0.1837, -1.1883],
        [-2.0065, -0.6708],
        [-0.2898, -1.5977],
        [-1.0663,  0.2929],
        [ 2.1540,  0.0948],
        [-0.6652, -1.7583],
        [-0.0791, -0.7941],
        [ 1.8723,  0.0476]])
Generator output: tensor([[ 0.0230, -0.3400],
        [ 0.0109, -0.3135],
        [-0.0140, -0.4628],
        [-0.1634, -0.3867],
        [-0.0368, -0.5133],
        [-0.0359, -0.2628],
        [-0.0935, -0.3890],
        [-0.0717, -0.5581],
        [ 0.0064, -0.4081],
        [-0.0767, -0.3759]], grad_fn=<AddmmBackward0>)
Discriminator output for generated data: tensor([[0.4518],
        [0.4518],
        [0.4510],
        [0.4508],
        [0.4503],
        [0.4518],
        [0.4513],
        [0.4494],
        [0.4519],
        [0.4516]], grad_fn=<SigmoidBackward0>)
Discriminator output for real data: tensor([[0.2508],
        [0.2111],
        [0.2366],
        [0.2985],
        [0.3342],
        [0.2557],
        [0.3422],
        [0.3740],
        [0.2227],
        [0.2167]], grad_fn=<SigmoidBackward0>)
```

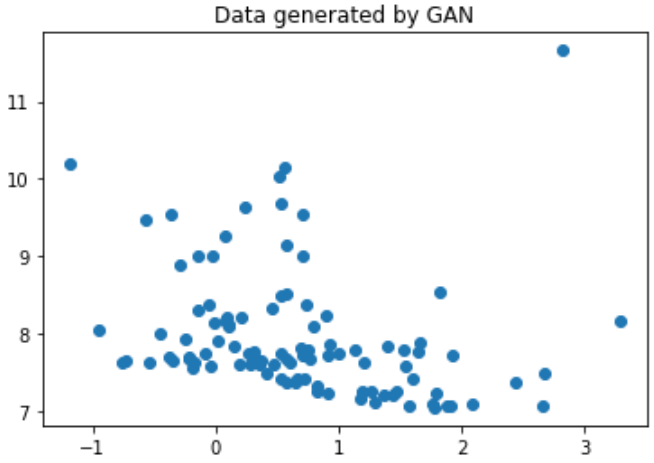
Let's write some code to train these models using the algorithm from Goodfellow et al. (2014):

In [8]:
```
from IPython.display import clear_output
from torch import optim
%matplotlib inline

num_iters = 1000
num_minibatches_discriminator = 5
minibatch_size = 100
n = 2

G = GeneratorNet()
D = DiscriminatorNet()

D_optimizer = optim.Adam(D.parameters(), lr=0.001)
G_optimizer = optim.Adam(G.parameters(), lr=0.001)
loss = nn.BCELoss()

# for number of training iterations

d_losses = []
g_losses = []

def do_plot(d_losses, g_losses):
    plt.figure(figsize=(10,10))
    clear_output(wait=True)
    plt.plot(d_losses, label='Discriminator')
    plt.plot(g_losses, label='Generator')
    plt.title('GAN loss')
    plt.legend()
    plt.show()

G.train()
D.train()

for iter in range(num_iters):

    # Train discriminator for num_minibatches_discriminator minibatches

    d_loss = 0
    for discriminator_iter in range(num_minibatches_discriminator):
        D.zero_grad()
        D_optimizer.zero_grad()
        x = torch.tensor(sample_pdata(minibatch_size)).float()
        z = torch.tensor(sample_noise(minibatch_size, n)).float()
        xhat = G(z)
        decisions_real = D(x)
        real_targets = torch.ones(minibatch_size, 1)
        error_real = loss(decisions_real, real_targets)
        error_real.backward()
        decisions_fake = D(xhat)
        fake_targets = torch.zeros(minibatch_size, 1)
        error_fake = loss(decisions_fake, fake_targets)
        error_fake.backward()
        D_optimizer.step()
        d_loss += error_real + error_fake

    # Train generator on one minibatch

    G.zero_grad()
    D.zero_grad()
    G_optimizer.zero_grad()
    z = torch.tensor(sample_noise(minibatch_size, n)).float()
    xhat = G(z)
    decisions_fake = D(xhat)
    fake_targets = torch.ones(minibatch_size, 1)
    g_loss = loss(decisions_fake, fake_targets)
    g_loss.backward()
    G_optimizer.step()
```

```
d_losses.append(d_loss.item())
g_losses.append(g_loss.item())

do_plot(d_losses, g_losses)
```



In [9]:
```
G.eval()
z = torch.tensor(sample_noise(100, 2)).float()
xhat = G(z).detach().numpy()

plt.scatter(xhat[:,0],xhat[:,1])
plt.title('Data generated by GAN')
plt.show()
```



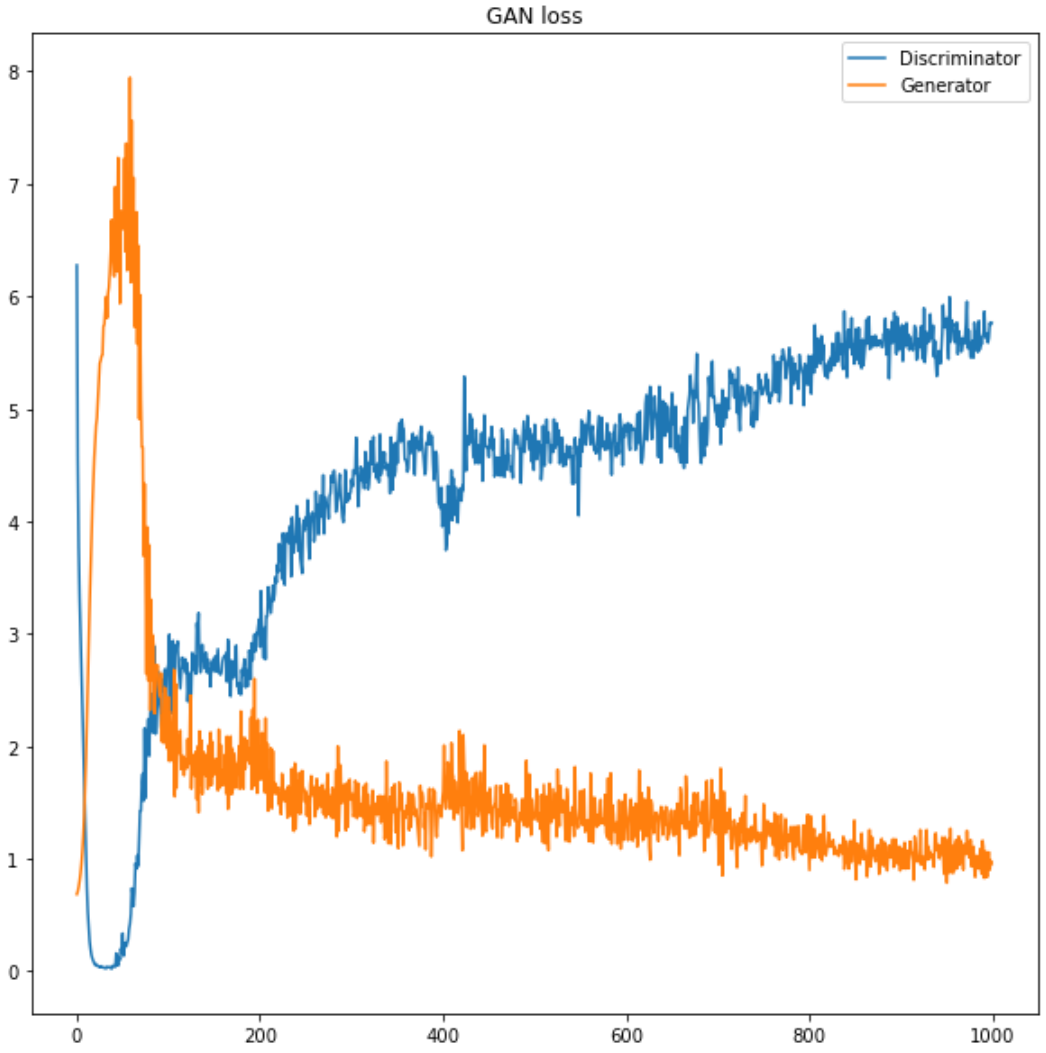
Exercise 1 (50 points)

As the results are not yet convincing, perform some further experiments with bigger noise vectors, larger networks, and different hyperparameters to improve the results. In your report, describe your experiments and demonstrate the results.

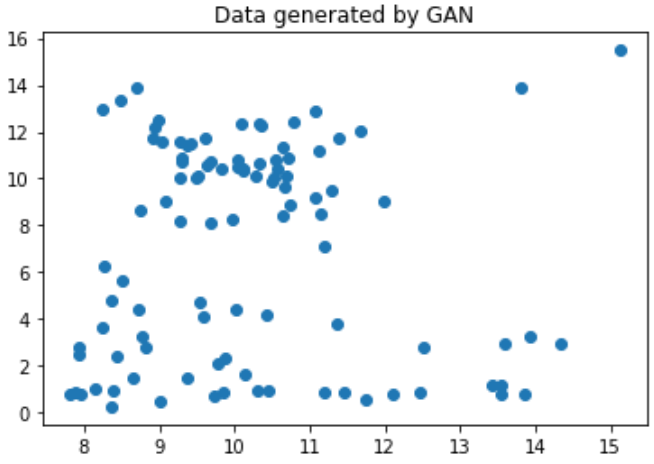
In [10]:
```
class GeneratorNet(nn.Module):
    def __init__(self):
        super(GeneratorNet, self).__init__()
        # First fully connected layer
        self.fc1 = nn.Linear(2, 50)
        # Second fully connected layer
        self.fc2 = nn.Linear(50, 50)
        self.output = nn.Linear(50, 2)

    def forward(self, x):
        # Pass data through fc1
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        output = self.output(x)
        return output

class DiscriminatorNet(nn.Module):
    def __init__(self):
        super(DiscriminatorNet, self).__init__()
        # First fully connected layer
        self.fc1 = nn.Linear(2, 50)
        # Second fully connected layer
        self.fc2 = nn.Linear(50, 50)
        self.fc3 = nn.Linear(50,50)
        self.output = nn.Linear(50, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.output(x)
        return F.sigmoid(x)

# Instantiate the generator and discriminator

G = GeneratorNet()
D = DiscriminatorNet()

# xhat = G(noise sample)

z = torch.tensor(sample_noise(200, 2)).float()
xhat = G(z)

# decisions on fake data = D(G(noise sample))

decisions_fake = D(xhat)

# decisions on real data = D(data sample)

x = torch.tensor(sample_pdata(10)).float()
decisions_real = D(x)
```

In [11]:
```
num_iters = 1000
num_minibatches_discriminator = 5
minibatch_size = 100
n = 2

G = GeneratorNet()
D = DiscriminatorNet()

D_optimizer = optim.Adam(D.parameters(), lr=0.001)
G_optimizer = optim.Adam(G.parameters(), lr=0.001)
loss = nn.BCELoss()

# for number of training iterations

d_losses = []
g_losses = []

def do_plot(d_losses, g_losses):
    plt.figure(figsize=(10,10))
    clear_output(wait=True)
    plt.plot(d_losses, label='Discriminator')
    plt.plot(g_losses, label='Generator')
    plt.title('GAN loss')
    plt.legend()
    plt.show()

G.train()
D.train()

for iter in range(num_iters):

    # Train discriminator for num_minibatches_discriminator minibatches

    d_loss = 0
    for discriminator_iter in range(num_minibatches_discriminator):
        D.zero_grad()
        D_optimizer.zero_grad()
        x = torch.tensor(sample_pdata(minibatch_size)).float()
        z = torch.tensor(sample_noise(minibatch_size, n)).float()
        xhat = G(z)
        decisions_real = D(x)
        real_targets = torch.ones(minibatch_size, 1)
        error_real = loss(decisions_real, real_targets)
        error_real.backward()
        decisions_fake = D(xhat)
        fake_targets = torch.zeros(minibatch_size, 1)
        error_fake = loss(decisions_fake, fake_targets)
        error_fake.backward()
        D_optimizer.step()
        d_loss += error_real + error_fake

    # Train generator on one minibatch

    G.zero_grad()
    D.zero_grad()
    G_optimizer.zero_grad()
    z = torch.tensor(sample_noise(minibatch_size, n)).float()
    xhat = G(z)
    decisions_fake = D(xhat)
    fake_targets = torch.ones(minibatch_size, 1)
    g_loss = loss(decisions_fake, fake_targets)
    g_loss.backward()
    G_optimizer.step()

    d_losses.append(d_loss.item())
    g_losses.append(g_loss.item())
```

```
do_plot(d_losses, g_losses)
```



In [12]:
```
G.eval()
z = torch.tensor(sample_noise(100, 10)).float()
xhat = G(z).detach().numpy()

plt.scatter(xhat[:,0],xhat[:,1])
plt.title('Data generated by GAN')
plt.show()
```



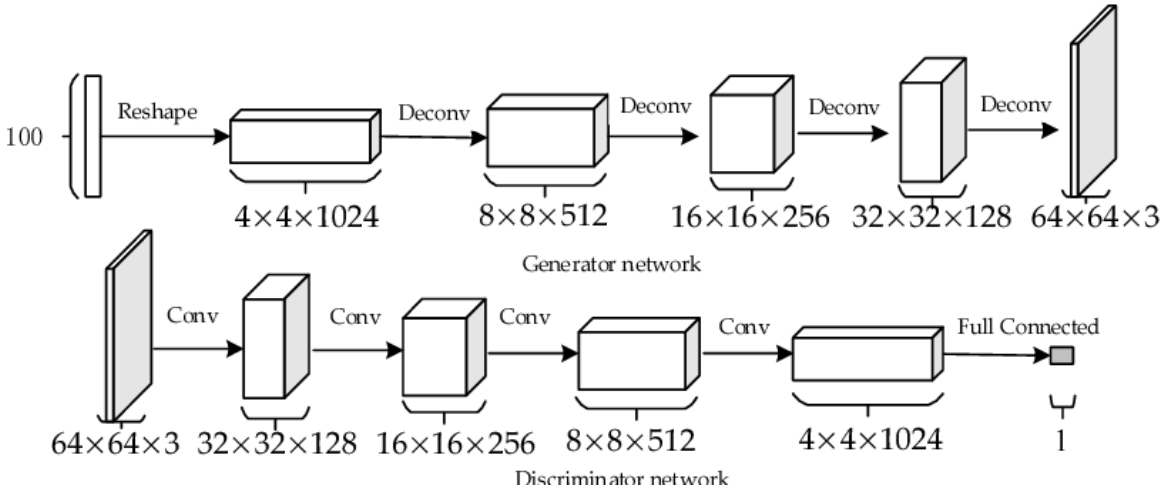
Example 2: Deep Convolutional GAN

This DCGAN tutorial is from the DCGAN Tutorial (https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html) and the data is from Mckinsey666 (https://github.com/Mckinsey666/Anime-Face-Dataset)

The goal of this paper is to generate fake images like as shown below. For this lab we will use 2000 of the original 60,000 images.

The DCGAN is a GAN with a generator designed to do for generate larger RGB images using convolutional layers.

Here is the DCGAN architecture:



First, we import some additional required libraries.

In [13]:
```
%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

# Set random seed for reproducibility
seed = 6969
# seed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", seed)
random.seed(seed)
torch.manual_seed(seed)
```

```
Random Seed:  6969
```

Out[13]:
```
<torch._C.Generator at 0x7f946d675110>
```

Next, we define some important variables.

In [14]:
```
# Root directory for dataset
dataroot = '/home/Datasets/anime-faces/'

# Number of workers for dataloader
workers = 0

# 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 = 10 # Original is 5 on a dataset of 1 million

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

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

Next, let's create and preview the dataset.

In [19]:
```
# 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:1" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
#real_batch = next(dataloader.__iter__())
real_batch_i, real_batch = next(enumerate(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).cpu
```

Out[19]: <matplotlib.image.AxesImage at 0x7f944c678f70>

Weight initialization

Here is a weight initialization function:

In [20]:
```
# 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

In [21]:
```
# 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)
```

In [22]:
```
# 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), bias=False)
    (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), bias=False)
    (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), bias=False)
    (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=False)
    (13): Tanh()
  )
)
```

Discriminator

In [23]:
```
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)
```

In [24]:
```
# 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()
  )
)
```

Loss functions and optimizers

In [25]:
```
# 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

You will need to follow these conceptual instructions.

1. Create noise
2. Input noise to generator network to get fake images
3. Input fake images to discriminator and detect it that true or false. Calculate $loss_{fake}$ with **True** probability
4. Input real images to discriminator and detect it that true or false. Calculate $loss_{real}$ with **True** probability
5. $loss_d = (loss_{fake} + loss_{real})$
6. back propagation discriminator network.

7. Input fake images to discriminator and detect it that true or false. Calculate $loss_{gan}$ with **Fake** probability
8. back propagation generator network.
9. loop it!

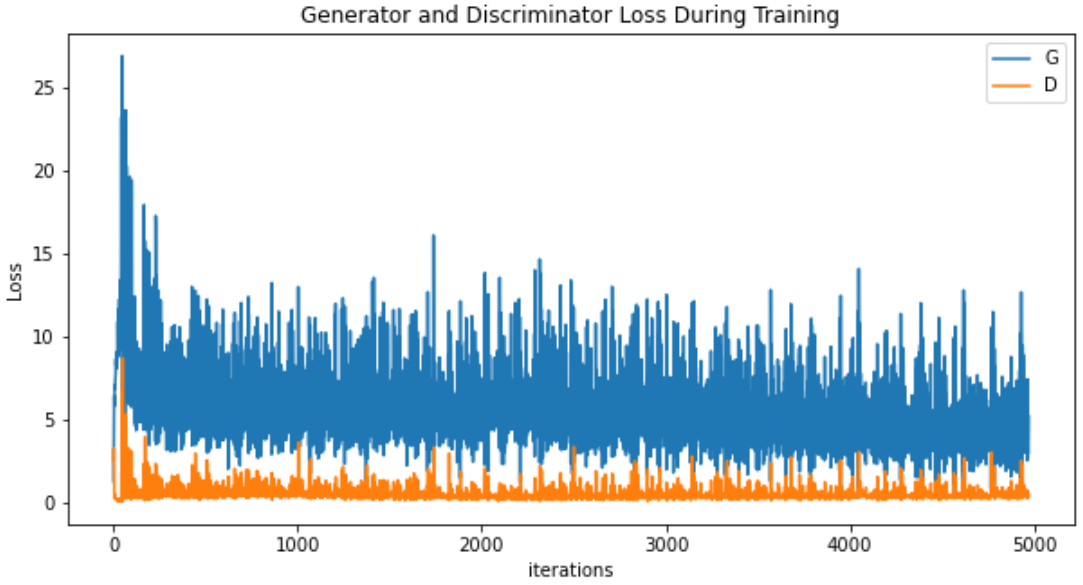
In [26]:
```
# Training Loop

# 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, dtype=torch.float, 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, accumulated (summed) with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real 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%200 == 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
```

```
Starting Training Loop...
[0/10][0/497]    Loss_D: 2.5167  Loss_G: 1.2817  D(x): 0.1340    D(G(z)): 0.1608 / 0.3492
[0/10][200/497]  Loss_D: 0.2687  Loss_G: 7.3043  D(x): 0.8523    D(G(z)): 0.0027 / 0.0015
[0/10][400/497]  Loss_D: 0.3111  Loss_G: 6.8362  D(x): 0.8153    D(G(z)): 0.0188 / 0.0020
[1/10][0/497]    Loss_D: 0.7591  Loss_G: 9.9992  D(x): 0.9723    D(G(z)): 0.4667 / 0.0001
[1/10][200/497]  Loss_D: 0.6509  Loss_G: 8.0601  D(x): 0.6637    D(G(z)): 0.0077 / 0.0008
[1/10][400/497]  Loss_D: 1.3619  Loss_G: 8.5397  D(x): 0.4101    D(G(z)): 0.0005 / 0.0006
[2/10][0/497]    Loss_D: 0.4993  Loss_G: 4.6211  D(x): 0.7624    D(G(z)): 0.1112 / 0.0172
[2/10][200/497]  Loss_D: 0.4472  Loss_G: 6.6328  D(x): 0.9157    D(G(z)): 0.2714 / 0.0020
[2/10][400/497]  Loss_D: 0.4769  Loss_G: 6.3119  D(x): 0.8631    D(G(z)): 0.2359 / 0.0030
[3/10][0/497]    Loss_D: 0.3582  Loss_G: 9.5153  D(x): 0.7717    D(G(z)): 0.0067 / 0.0005
[3/10][200/497]  Loss_D: 0.5252  Loss_G: 7.6284  D(x): 0.6844    D(G(z)): 0.0023 / 0.0009
[3/10][400/497]  Loss_D: 0.3967  Loss_G: 4.6066  D(x): 0.8647    D(G(z)): 0.1804 / 0.0188
[4/10][0/497]    Loss_D: 0.3019  Loss_G: 5.0309  D(x): 0.8934    D(G(z)): 0.1326 / 0.0114
[4/10][200/497]  Loss_D: 0.3838  Loss_G: 6.1939  D(x): 0.9218    D(G(z)): 0.2056 / 0.0038
[4/10][400/497]  Loss_D: 0.7065  Loss_G: 10.5739  D(x): 0.9475    D(G(z)): 0.4159 / 0.0001
[5/10][0/497]    Loss_D: 0.1159  Loss_G: 4.6605  D(x): 0.9147    D(G(z)): 0.0049 / 0.0258
[5/10][200/497]  Loss_D: 0.4251  Loss_G: 3.4109  D(x): 0.7295    D(G(z)): 0.0273 / 0.0540
[5/10][400/497]  Loss_D: 0.2753  Loss_G: 5.4465  D(x): 0.8996    D(G(z)): 0.1310 / 0.0089
[6/10][0/497]    Loss_D: 0.3821  Loss_G: 6.8792  D(x): 0.9339    D(G(z)): 0.2315 / 0.0023
[6/10][200/497]  Loss_D: 0.3058  Loss_G: 4.5226  D(x): 0.8607    D(G(z)): 0.1112 / 0.0181
[6/10][400/497]  Loss_D: 0.5739  Loss_G: 7.2162  D(x): 0.9830    D(G(z)): 0.3537 / 0.0019
[7/10][0/497]    Loss_D: 0.2180  Loss_G: 6.3770  D(x): 0.9706    D(G(z)): 0.1513 / 0.0043
[7/10][200/497]  Loss_D: 2.7023  Loss_G: 2.9689  D(x): 0.1282    D(G(z)): 0.0003 / 0.1175
[7/10][400/497]  Loss_D: 0.3742  Loss_G: 6.2056  D(x): 0.9367    D(G(z)): 0.2354 / 0.0037
[8/10][0/497]    Loss_D: 0.4787  Loss_G: 2.7975  D(x): 0.7364    D(G(z)): 0.0540 / 0.1000
[8/10][200/497]  Loss_D: 0.5043  Loss_G: 7.5225  D(x): 0.9643    D(G(z)): 0.3342 / 0.0010
[8/10][400/497]  Loss_D: 0.3107  Loss_G: 5.0497  D(x): 0.9851    D(G(z)): 0.2232 / 0.0124
[9/10][0/497]    Loss_D: 0.9230  Loss_G: 8.0419  D(x): 0.9718    D(G(z)): 0.5043 / 0.0009
[9/10][200/497]  Loss_D: 0.4068  Loss_G: 5.4708  D(x): 0.9726    D(G(z)): 0.2746 / 0.0080
[9/10][400/497]  Loss_D: 0.4533  Loss_G: 1.5289  D(x): 0.7144    D(G(z)): 0.0451 / 0.2883
```

Plot losses

In [27]:
```
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()
```



Visualize the training process

In [28]:
```
#%%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())
```

Out[28]:

Display final results

In [29]:
```
# Grab a batch of real images from the dataloader
real_batch = next(dataloader.__iter__())

# 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

# 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()
```

Take-home exercise (50 points)

Find another interesting image generation application of the DCGAN and implement it. Demonstate your results in your report.

In [38]:
```
# Root directory for dataset
dataroot = '/home/st122410/work/AT82.03/12-GANs/Dataset/'

# Number of workers for dataloader
workers = 0

# Batch size during training
batch_size = 16

# 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 = 1000 # Original is 5 on a dataset of 1 million

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

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

In [39]:
```
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
#                                transforms.Grayscale(),
                               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:1" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

print(type(dataset))
# Plot some training images
real_batch = next(dataloader.__iter__())
# real_batch_i, real_batch = next(enumerate(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).cpu
```

```
<class 'torchvision.datasets.folder.ImageFolder'>
```

Out[39]: <matplotlib.image.AxesImage at 0x7f94640d5490>

In [40]:
```
# 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), bias=False)
    (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), bias=False)
    (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), bias=False)
    (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=False)
    (13): Tanh()
  )
)
```

In [41]:
```
# 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()
  )
)
```

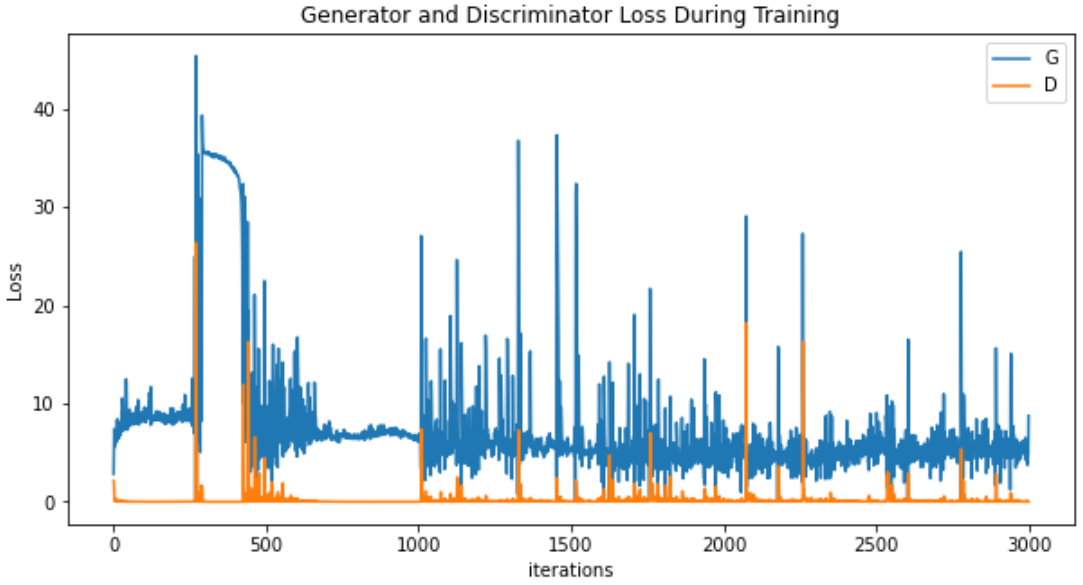
In [42]:
```
# Initialize BCELoss function
criterion = nn.BCELoss()

# 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))
```

In [43]:
```
# 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, dtype=torch.float, 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, accumulated (summed) with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real 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

        # 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

    if epoch%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))
```

```
Starting Training Loop...
[0/1000][2/3]    Loss_D: 0.9040  Loss_G: 7.4131  D(x): 1.0000      D(G(z)): 0.5064 / 0.0008
[50/1000][2/3]   Loss_D: 0.0056  Loss_G: 8.4002  D(x): 0.9984      D(G(z)): 0.0040 / 0.0003
[100/1000][2/3] Loss_D: 0.0002  Loss_G: 35.5641        D(x): 0.9998      D(G(z)): 0.0000 / 0.0000
[150/1000][2/3] Loss_D: 0.0246  Loss_G: 4.1182  D(x): 0.9871      D(G(z)): 0.0112 / 0.0187
[200/1000][2/3] Loss_D: 0.3790  Loss_G: 16.5711        D(x): 0.9957      D(G(z)): 0.2972 / 0.0000
[250/1000][2/3] Loss_D: 0.0073  Loss_G: 6.6028  D(x): 0.9982      D(G(z)): 0.0054 / 0.0014
[300/1000][2/3] Loss_D: 0.0029  Loss_G: 7.0790  D(x): 0.9991      D(G(z)): 0.0020 / 0.0010
[350/1000][2/3] Loss_D: 0.0654  Loss_G: 6.8289  D(x): 0.9473      D(G(z)): 0.0045 / 0.0011
[400/1000][2/3] Loss_D: 0.0842  Loss_G: 6.1749  D(x): 0.9991      D(G(z)): 0.0796 / 0.0022
[450/1000][2/3] Loss_D: 0.0459  Loss_G: 6.6100  D(x): 0.9828      D(G(z)): 0.0276 / 0.0016
[500/1000][2/3] Loss_D: 0.0241  Loss_G: 5.5273  D(x): 0.9931      D(G(z)): 0.0170 / 0.0050
[550/1000][2/3] Loss_D: 0.1898  Loss_G: 3.7862  D(x): 0.8505      D(G(z)): 0.0166 / 0.0236
[600/1000][2/3] Loss_D: 0.2846  Loss_G: 4.5879  D(x): 0.8337      D(G(z)): 0.0061 / 0.0169
[650/1000][2/3] Loss_D: 0.1300  Loss_G: 4.2790  D(x): 0.9916      D(G(z)): 0.1121 / 0.0157
[700/1000][2/3] Loss_D: 0.0750  Loss_G: 6.2626  D(x): 0.9396      D(G(z)): 0.0062 / 0.0040
[750/1000][2/3] Loss_D: 0.1835  Loss_G: 2.7638  D(x): 0.8654      D(G(z)): 0.0064 / 0.0761
[800/1000][2/3] Loss_D: 0.0841  Loss_G: 5.1636  D(x): 0.9790      D(G(z)): 0.0603 / 0.0065
[850/1000][2/3] Loss_D: 1.5019  Loss_G: 5.0606  D(x): 0.4089      D(G(z)): 0.0038 / 0.0441
[900/1000][2/3] Loss_D: 0.0188  Loss_G: 3.9822  D(x): 0.9955      D(G(z)): 0.0142 / 0.0200
[950/1000][2/3] Loss_D: 0.0571  Loss_G: 4.6283  D(x): 0.9805      D(G(z)): 0.0365 / 0.0115
```

In [44]:
```
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()
```



In [45]:
```
# Grab a batch of real images from the dataloader
real_batch = next(dataloader.__iter__())

# 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

# 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()
```