

In [1]:

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

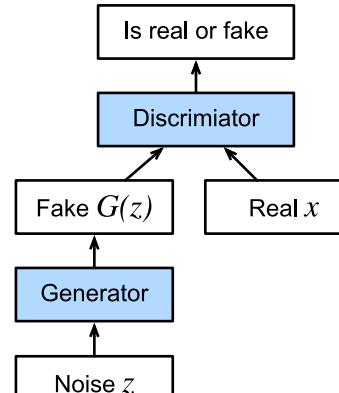
# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

In [2]:

```
import torch
from torch import nn
from d2l import torch as d2l
```



Generative Adversarial Networks

Generate some "real" data

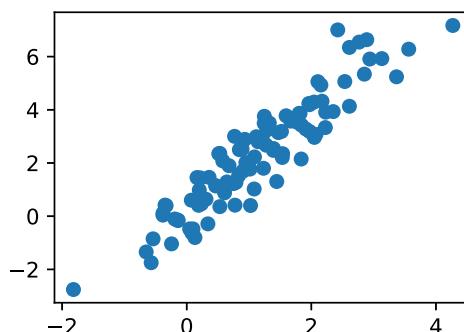
simply generate data drawn from a Gaussian

In [3]:

```
X = torch.normal(0.0, 1, (1000, 2)) # mean = 0.0, standard deviation = 1, shape = (1000, 2)
A = torch.tensor([[1, 2], [-0.1, 0.5]])
b = torch.tensor([1, 2])
data = torch.matmul(X, A) + b

d2l.set figsize()
d2l.plt.scatter(data[:, 0], data[:, 1])
```

Out[3]: <matplotlib.collections.PathCollection at 0x1e12ada7688>



Generator & Discriminator

Generator: single linear model

Discriminator: MLP with 3 layers

In [4]:

```
net_G = nn.Sequential(nn.Linear(2, 2)) # input_features = 2, output_features = 2

net_D = nn.Sequential(nn.Linear(2, 5),
                     nn.Tanh(),
                     nn.Linear(5, 3),
                     nn.Tanh(),
                     nn.Linear(3, 1))
```

Training

In [5]:

```
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    batch_size = X.shape[0]
    ones = torch.ones((batch_size, ), device=X.device)
    zeros = torch.zeros((batch_size, ), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)

    fake_Y = net_D(fake_X.detach()) # do not need to compute gradient
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2

    loss_D.backward()
    trainer_D.step()

    return loss_D
```

In [6]:

```
def update_G(Z, net_D, net_G, loss, trainer_G):
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size, ), device=Z.device)
    trainer_G.zero_grad()

    fake_X = net_G(Z)
    fake_Y = net_D(fake_X)

    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()

    return loss_G
```

In [7]:

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    # Binary Cross Entropy Loss with Logistic Loss, (binary logistic regression + cross entropy loss)
    # BCELoss + sigmoid
    loss = nn.BCEWithLogitsLoss(reduction='sum')

    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)

    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, num_epochs],
                             nrows=2, figsize=(5, 5),
                             legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)

    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(3) # loss_D, loss_G, num_examples

        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim)) # (8, 2)
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)

            # generate numbers
            Z = torch.normal(0, 1, size=(100, latent_dim)) # (100, 2)
            fake_X = net_G(Z).detach().numpy()

            animator.axes[1].cla()
            animator.axes[1].scatter(data[:, 0], data[:, 1])
            animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
            animator.axes[1].legend(['real', 'generated'])

            loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
            animator.add(epoch + 1, (loss_D, loss_G))

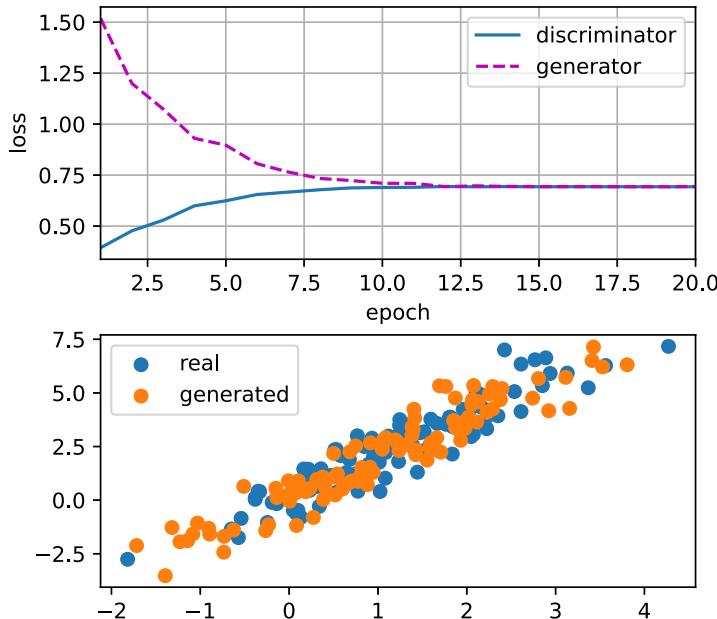
        print("loss_D: {:.3f}, loss_G: {:.3f}, {:.1f} examples/sec".
              format(loss_D, loss_G, metric[2] / timer.stop()))
```

In [8]:

```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

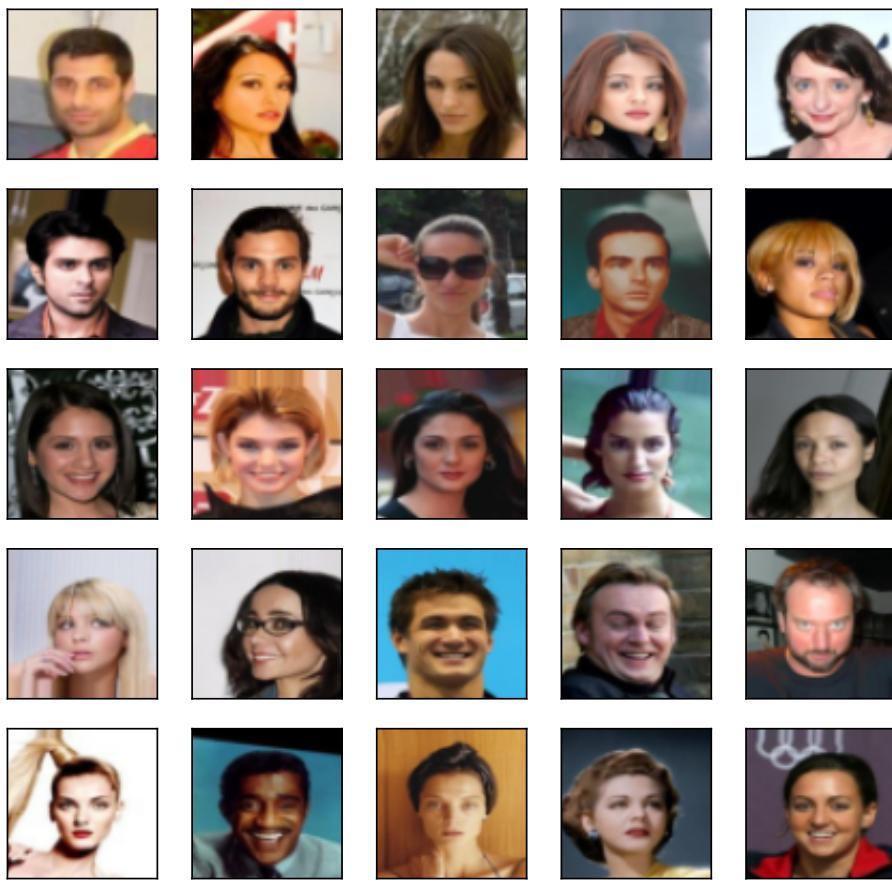
```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data[:100].detach().numpy())
```

loss_D: 0.693, loss_G: 0.694, 2861.9 examples/sec



Deep Convolutional GAN - celebrity

```
In [10]:  
import torchvision  
import warnings, os  
  
data_dir = 'D:\\Project\\WBM\\DeepLearning\\data\\archive'  
celebrity = torchvision.datasets.ImageFolder(data_dir)  
  
In [11]:  
batch_size = 256  
transformer = torchvision.transforms.Compose([  
    torchvision.transforms.Resize((64, 64)),  
    torchvision.transforms.ToTensor(),  
    torchvision.transforms.Normalize(0.5, 0.5)  
)  
celebrity.transform = transformer  
data_iter = torch.utils.data.DataLoader(celebrity, batch_size = batch_size,  
                                         shuffle = True, num_workers = d2l.get_dataloader_workers())  
  
In [12]:  
warnings.filterwarnings('ignore')  
d2l.set_figsize((5, 5))  
for X, y in data_iter:  
    imgs = X[0:25, :, :, :].permute(0, 2, 3, 1) / 2 + 0.5  
    # torch.Size([25, 3, 64, 64]) -> torch.Size([25, 64, 64, 3])  
    print(imgs.shape)  
    d2l.show_images(imgs, num_rows = 5, num_cols = 5)  
    break  
  
torch.Size([25, 64, 64, 3])
```



The Generator

```
In [13]: class g_block(nn.Module):
    def __init__(self, out_channels, in_channels = 3, kernel_size = 4, strides = 2,
                 padding = 1, **kwargs):
        super(g_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
                                             strides, padding, bias = False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU()

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

In default, the transposed convolution layer uses a $k_h = k_w = 4$ kernel, a $s_h = s_w = 2$ strides, and a $p_h = p_w = 1$ padding. With a input shape of $n'_h \times n'_w = 16 \times 16$

The generator block will double input's width and height.

$$\begin{aligned} n'_h \times n'_w &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h) \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w)] \\ &= [(k_h + s_h(n_h - 1) - 2p_h) \times [(k_w + s_w(n_w - 1) - 2p_w)] \\ &= [(4 + 2 \times (16 - 1) - 2 \times 1) \times [(4 + 2 \times (16 - 1) - 2 \times 1)] \\ &= 32 \times 32. \end{aligned}$$

```
In [14]: x = torch.zeros((2, 3, 16, 16))
g_bk = g_block(20)
g_bk(x).shape
```

```
Out[14]: torch.Size([2, 20, 32, 32])
```

```
In [15]: n_g = 64
net_g = nn.Sequential(
    g_block(in_channels = 100, out_channels = n_g * 8, strides = 1, padding = 0),
    # output: (64 * 8, 4, 4)
    g_block(in_channels = n_g * 8, out_channels = n_g * 4),
    # output: (64 * 4, 8, 8)
    g_block(in_channels = n_g * 4, out_channels = n_g * 2),
    # output: (64 * 2, 16, 16)
    g_block(in_channels = n_g * 2, out_channels = n_g),
    # output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels = n_g, out_channels = 3, kernel_size = 4, stride = 2,
                      padding = 1, bias = False),
    # output: (3, 64, 64)
```

```
nn.Tanh()  
)
```

```
In [16]:  
x = torch.zeros((1, 100, 1, 1))  
net_g(x).shape
```

```
Out[16]: torch.Size([1, 3, 64, 64])
```

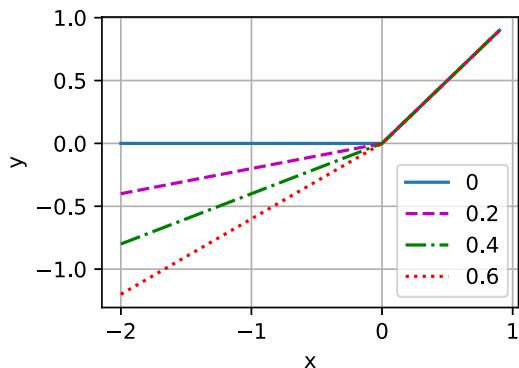
Discriminator

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

pytorch default alpha 0.01

keras default alpha 0.3

```
In [17]:  
alphas = [0, .2, .4, .6, .8, 1]  
x = torch.arange(-2, 1, 0.1)  
y = [nn.LeakyReLU(alpha)(x) for alpha in alphas]  
d2l.plot(x.detach().numpy(), y, 'x', 'y', alphas)
```



```
In [18]:  
class d_block(nn.Module):  
    def __init__(self, out_channels, in_channels = 3, kernel_size = 4, strides = 2, padding = 1,  
                 alpha = 0.2, **kwargs):  
        super(d_block, self).__init__(**kwargs)  
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size, strides,  
                             padding, bias = False)  
        self.batch_norm = nn.BatchNorm2d(out_channels)  
        self.activation = nn.LeakyReLU(alpha, inplace = True)  
  
    def forward(self, X):  
        return self.activation(self.batch_norm(self.conv2d(X)))
```

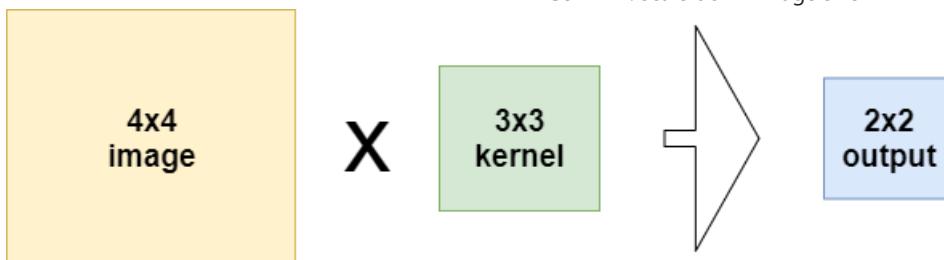
input shape $n_h = n_w = 16$, kernel shape $k_h = k_w = 4$, a stride shape $s_h = s_w = 2$, and a padding shpae $p_h = p_w = 1$.

The output shape

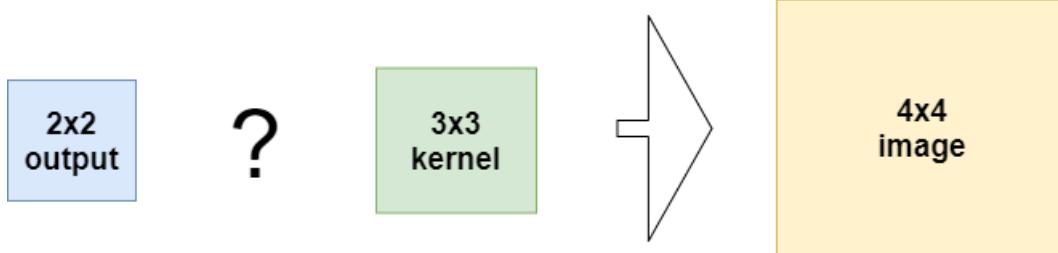
$$\begin{aligned}n'_h \times n'_w &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\&= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\&= 8 \times 8.\end{aligned}$$

</br> </br>

Conv2D: scale down image size



ConvTranspose2d: upsampling the image



<https://www.machinecurve.com/index.php/2019/12/10/conv2dtranspose-using-2d-transposed-convolutions-with-keras/>

```
In [19]: x = torch.zeros((2, 3, 16, 16))  
d_bk = d_block(20)  
d_bk(x).shape
```

```
Out[19]: torch.Size([2, 20, 8, 8])
```

```
In [20]: n_d = 64
net_d = nn.Sequential(
    d_block(n_d),
    # output: (64, 32, 32)
    d_block(in_channels = n_d, out_channels = n_d * 2),
    # output: (64 * 2, 16, 16)
    d_block(in_channels = n_d * 2, out_channels = n_d * 4),
    # output: (64, *4, 8, 8)
    d_block(in_channels = n_d * 4, out_channels = n_d * 8),
    # output: (64 * 8, 4, 4)
    nn.Conv2d(in_channels = n_d * 8, out_channels = 1, kernel_size = 4, bias= False)
)

```

```
In [21]: x = torch.zeros((1, 3, 64, 64))  
net_d(x).shape
```

```
Out[21]: torch.Size([1, 1, 1, 1])
```

Training

```

        for i in range(len(fake_x) // 5)], dim = 0)

animator.axes[1].cla()
animator.axes[1].imshow(imgs)

loss_d, loss_g = metric[0] / metric[2], metric[1] / metric[2]
animator.add(epoch, (loss_d, loss_g))

print("loss_d : {:.3f}, loss_g : {:.3f}".format(loss_d, loss_g))

```

In [23]:

```

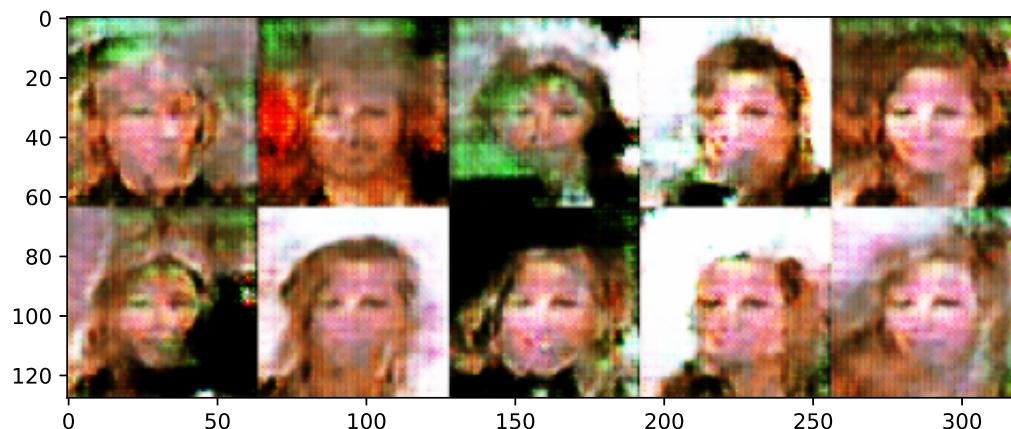
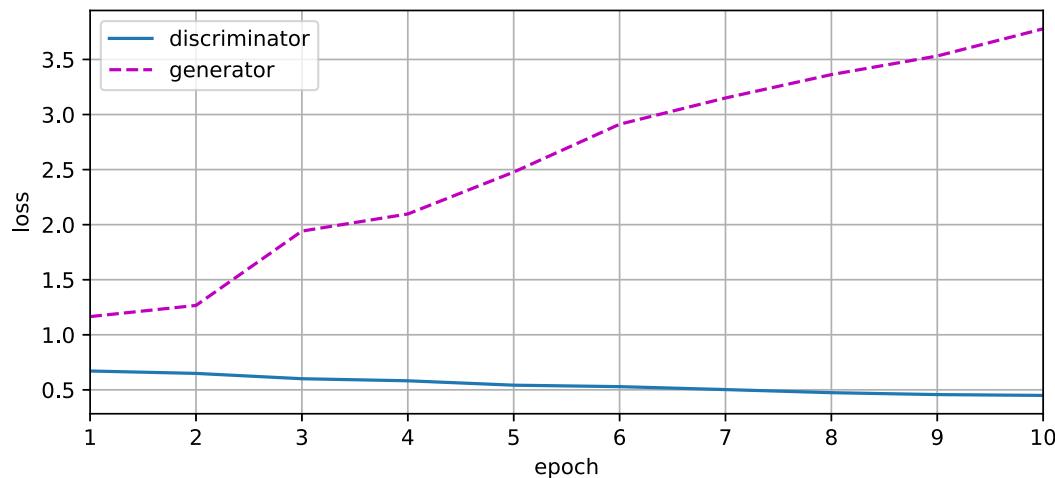
import time
start_time = time.time()

latent_dim, lr, num_epochs = 100, 0.005, 10
train(net_d, net_g, data_iter, num_epochs, lr, latent_dim)

print("run time: {:.2f} s".format(time.time() - start_time))

```

loss_d : 0.449, loss_g : 3.778
run time: 4262.84 s



Deep Convolutional GAN - Celeba

In [24]:

```

import argparse
import os
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

import torch
import torch.nn as nn
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

dataroot = 'D:\\WW2_Project\\WWBM\\WWDrive Into DeepLearning\\WWdata\\archive'
workers = 8
batch_size = 128
image_size = 64
nc = 3 # number of channels

```

```
nz = 100 # size of z latent vector
ngf = 64 # size of feature maps in generator
ndf = 64 # size of feature maps in discriminator
num_epochs = 10
lr = 0.0002
beta1 = 0.5
ngpu = 1 # number of GPUs available, 0: CPU
```

In [25]:

```
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))
                           ]))

dataloader = torch.utils.data.DataLoader(dataset, batch_size = batch_size,
                                         shuffle = True, num_workers = workers)

device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

real_batch = next(iter(dataloader))
plt.figure(figsize = (8, 8))
plt.axis("off")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device))[:64],
                       padding = 2, normalize=True).cpu(), (1, 2, 0)))
```

Out[25]: <matplotlib.image.AxesImage at 0x1e12ada7548>



In [26]:

```
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)
```

In [27]:

```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # nz = 100, ngf = 64, nc = 3
            nn.ConvTranspose2d(in_channels = nz, out_channels = ngf * 8,
                             kernel_size = 4, stride = 1, padding = 0, bias = False),
```

```

        nn.BatchNorm2d(ngf * 8),
        nn.ReLU(True),
        # output: (512, 4, 4)
        nn.ConvTranspose2d(in_channels = ngf * 8, out_channels = ngf * 4,
                           kernel_size = 4, stride = 2, padding = 1, bias = False),
        nn.BatchNorm2d(ngf * 4),
        nn.ReLU(True),
        # output: (256, 8, 8)
        nn.ConvTranspose2d(in_channels = ngf * 4, out_channels = ngf * 2,
                           kernel_size = 4, stride = 2, padding = 1, bias = False),
        nn.BatchNorm2d(ngf * 2),
        nn.ReLU(True),
        # output: (128, 16, 16)
        nn.ConvTranspose2d(in_channels = ngf * 2, out_channels = ngf,
                           kernel_size = 4, stride = 2, padding = 1, bias = False),
        nn.BatchNorm2d(ngf),
        nn.ReLU(True),
        # output: (64, 32, 32)
        nn.ConvTranspose2d(in_channels = ngf, out_channels = nc,
                           kernel_size = 4, stride= 2, padding = 1, bias = False),
        nn.Tanh()
        # output: (3, 64, 64)
    )

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

```

In [28]:

```

net_g = Generator(ngpu).to(device)
net_g.apply(weights_init)

print(net_g)

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 [29]:

```

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # nc = 3, ndf = 64
            nn.Conv2d(in_channels = nc, out_channels = ndf,
                      kernel_size = 4, stride = 2, padding = 1, bias = False),
            nn.LeakyReLU(0.2, inplace = True),
            # output: (64, 32, 32)
            nn.Conv2d(in_channels = ndf, out_channels = ndf * 2,
                      kernel_size = 4, stride = 2, padding = 1, bias = False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace = True),
            # output: (128, 16, 16)
            nn.Conv2d(in_channels = ndf * 2, out_channels = ndf * 4,
                      kernel_size = 4, stride = 2, padding = 1, bias = False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace = True),
            # output: (256, 8, 8)
            nn.Conv2d(in_channels = ndf * 4, out_channels = ndf * 8,
                      kernel_size = 4, stride = 2, padding = 1, bias = False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace = True),
            # output: (512, 4, 4)
            nn.Conv2d(in_channels = ndf * 8, out_channels = 1,
                      kernel_size = 4, stride = 1, padding = 0, bias = True),
            nn.Sigmoid()
        )

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

```

In [30]:

```

net_d = Discriminator(ngpu).to(device)

```

```

net_d.apply(weights_init)
print(net_d)

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))
    (12): Sigmoid()
  )
)

```

```

In [31]: criterion = nn.BCELoss() # Binary Cross Entropy Loss
fixed_noise = torch.randn(64, nz, 1, 1, device=device) # nz = 100
real_label = 1.0
fake_label = 0.0

# lr = 0.0002, beta1 = 0.5
optimizer_d = optim.Adam(net_d.parameters(), lr = lr, betas = (beta1, 0.999))
optimizer_g = optim.Adam(net_g.parameters(), lr = lr, betas = (beta1, 0.999))

```

```

In [32]: img_list = []
g_losses = []
d_losses = []
iters = 0

start_time = time.time()

for epoch in range(num_epochs):
    d_x, d_g_z1, d_g_z2 = 0., 0., 0.
    err_d, err_g = torch.Tensor(), torch.Tensor()

    for i, data in enumerate(dataloader, 0):
        # 1. update discriminator
        # train real data
        net_d.zero_grad()
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0) # batch_size = 128
        label = torch.full((b_size, ), real_label, dtype = torch.float, device = device)
        output = net_d(real_cpu).view(-1)
        err_d_real = criterion(output, label)
        err_d_real.backward()
        d_x = output.mean().item()

        # train fake data
        noise = torch.randn(b_size, nz, 1, 1, device = device) # (128, 3, 1, 1)
        fake = net_g(noise)
        label.fill_(fake_label)

        # classify fake data
        output = net_d(fake.detach()).view(-1)

        # calculate net_d loss on fake data
        err_d_fake = criterion(output, label)
        err_d_fake.backward()
        d_g_z1 = output.mean().item()

        # compute error of net_d as real and fake
        err_d = err_d_real + err_d_fake

        # update net_d
        optimizer_d.step()

        # 2. update generator
        net_g.zero_grad()
        label.fill_(real_label)

        # discriminate fake data after updated net_d
        output = net_d(fake).view(-1)

        # calculate net_g loss on fake data
        err_g = criterion(output, label)
        err_g.backward()
        d_g_z2 = output.mean().item()

    img_list.append(d_g_z1)
    g_losses.append(d_g_z2)
    d_losses.append(d_x)
    iters += 1

```

```

# update net_g
optimizer_g.step()

g_losses.append(err_g.item())
d_losses.append(err_d.item())

if (iters % 100 == 0) or ((epoch == num_epochs - 1) and (i == len(dataloader) - 1)):
    with torch.no_grad():
        fake = net_g(fixed_noise).detach().cpu()
    img_list.append(vutils.make_grid(fake, padding = 2, normalize = True))

    iters += 1
    print("{} / {} epochs, loss_d: {:.4f}, loss_g: {:.4f}, D(x): {:.4f}, D(G(z)): {:.4f} / {:.4f})".format(
        epoch + 1, num_epochs, err_d.item(), err_g.item(), d_x, d_g_z1, d_g_z2))

print("run time: {:.2f} s".format(time.time() - start_time))

```

```

1 / 10 epochs, loss_d: 0.5959, loss_g: 5.3781, D(x): 0.8641, D(G(z)): 0.2839 / 0.0135)
2 / 10 epochs, loss_d: 0.5957, loss_g: 4.6023, D(x): 0.8120, D(G(z)): 0.2386 / 0.0173)
3 / 10 epochs, loss_d: 0.3184, loss_g: 4.2069, D(x): 0.8374, D(G(z)): 0.0879 / 0.0209)
4 / 10 epochs, loss_d: 0.5386, loss_g: 4.5292, D(x): 0.8542, D(G(z)): 0.2186 / 0.0229)
5 / 10 epochs, loss_d: 0.8659, loss_g: 3.5099, D(x): 0.5431, D(G(z)): 0.0180 / 0.0715)
6 / 10 epochs, loss_d: 0.6193, loss_g: 7.4736, D(x): 0.8833, D(G(z)): 0.3270 / 0.0022)
7 / 10 epochs, loss_d: 0.5136, loss_g: 4.1903, D(x): 0.6891, D(G(z)): 0.0290 / 0.0447)
8 / 10 epochs, loss_d: 0.4364, loss_g: 5.5258, D(x): 0.9242, D(G(z)): 0.2625 / 0.0066)
9 / 10 epochs, loss_d: 1.6302, loss_g: 0.3278, D(x): 0.2994, D(G(z)): 0.0037 / 0.7812)
10 / 10 epochs, loss_d: 0.3307, loss_g: 5.4971, D(x): 0.8864, D(G(z)): 0.1623 / 0.0084)
run time: 3631.18 s

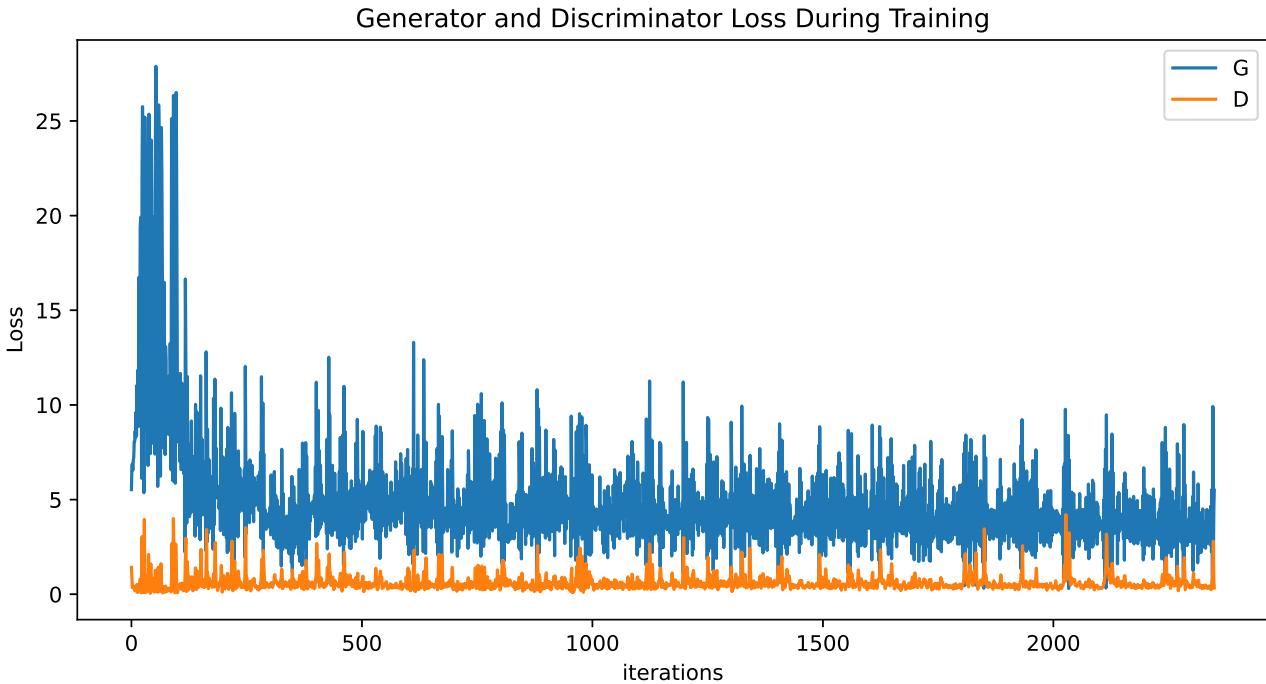
```

In [33]:

```

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 [34]:

```

from IPython.display import HTML

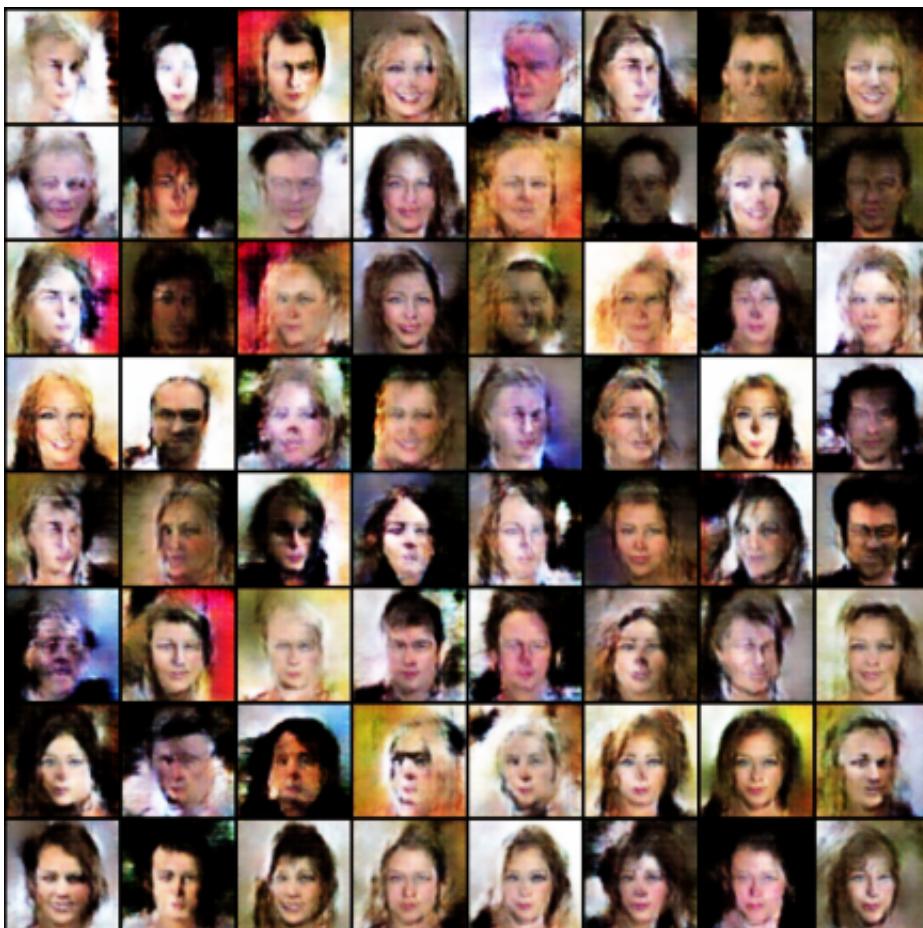
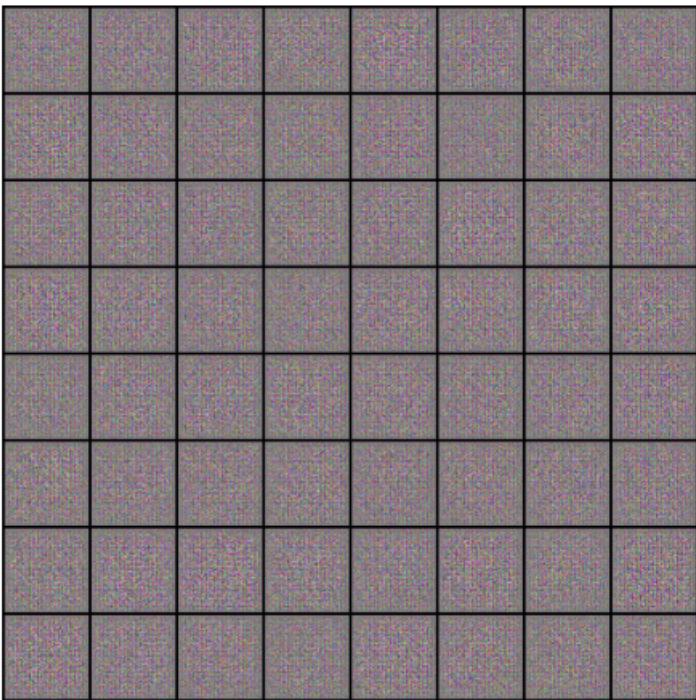
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

# plt.imshow(np.transpose(img_list[-1],(1,2,0)))

HTML(ani.to_jshtml())

```

Out[34]:



In [35]:

```
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(13,13))
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()

```

Real Images



Fake Images

