Lab 12: GANs

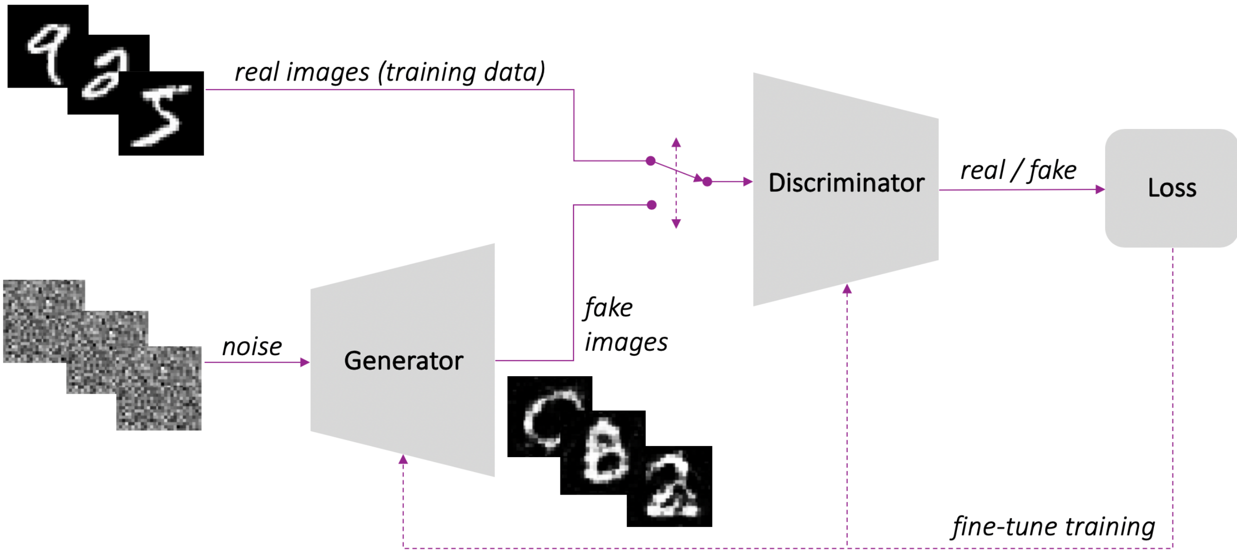
The lab has been adopted some part from RTML2021 (I will add conditional GAN for RTML2022)

Reference

- Build Basic Generative Adversarial Networks: Week 1, Deeplearning.AI, Coursera

In this lab, we will develop several basic GANs and experiment with them.

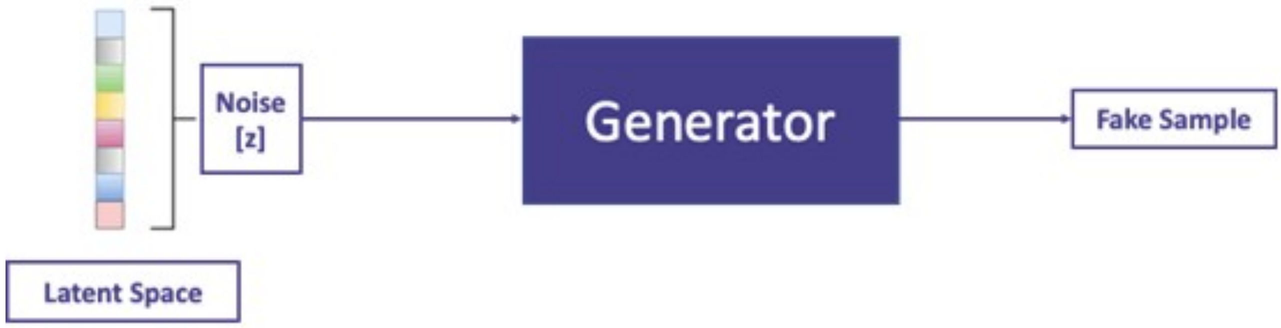
Here is the basic GAN model described by Goodfellow et al. (2014):



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

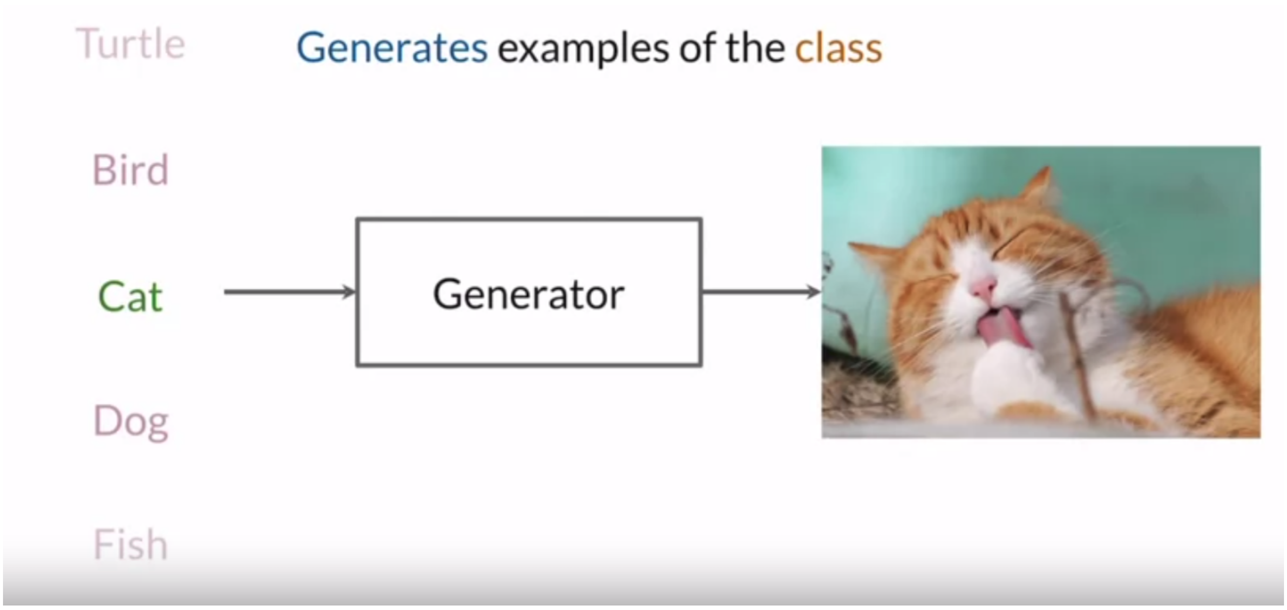
Generator

The generator is driven by a noise vector sampled from a latent space (the domain of p_z and transforms that noise sample into an element of the domain of p_{data}.



The generator in a GAN is like it's heart. It's a model that's used to generate examples and the one that you should be invested in and helping achieve a really high performance at the end of the training process.

The generators final goal is to be able 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 real.



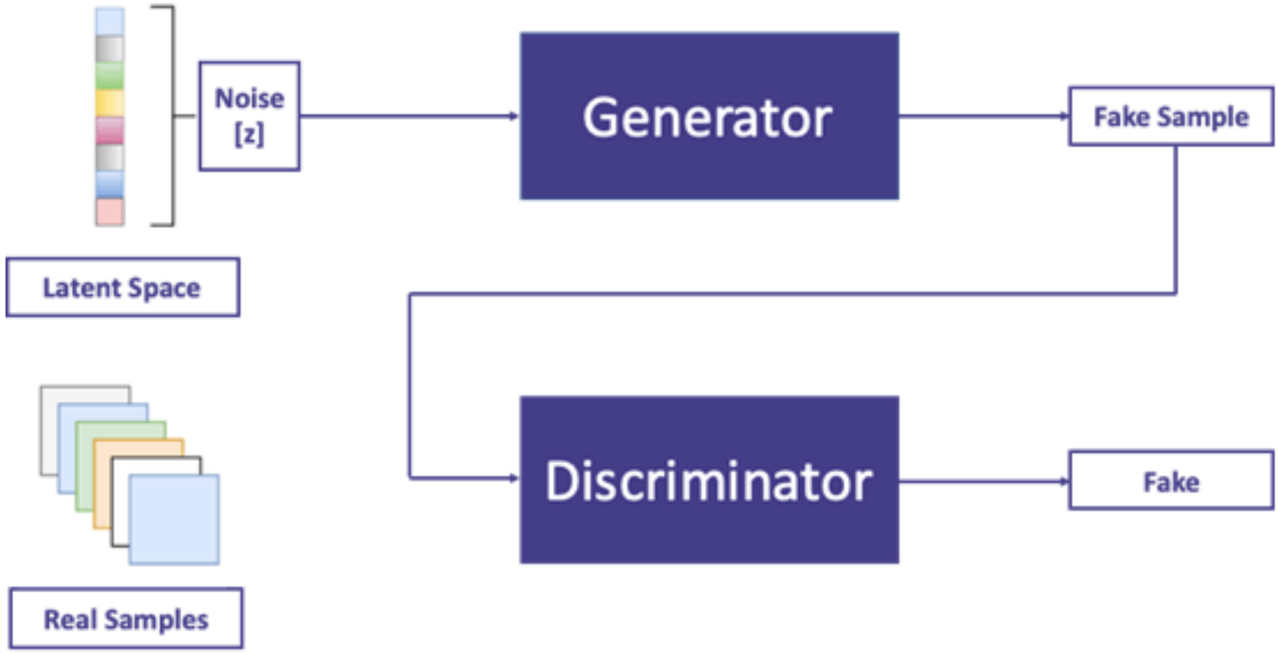
Ideally, the generator won't output the same cat at every run, and so to ensure it's able to produce different examples every single time, you actually will input different sets of random values, also known as a noise vector.

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 our class y for cat into the generators 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.

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:

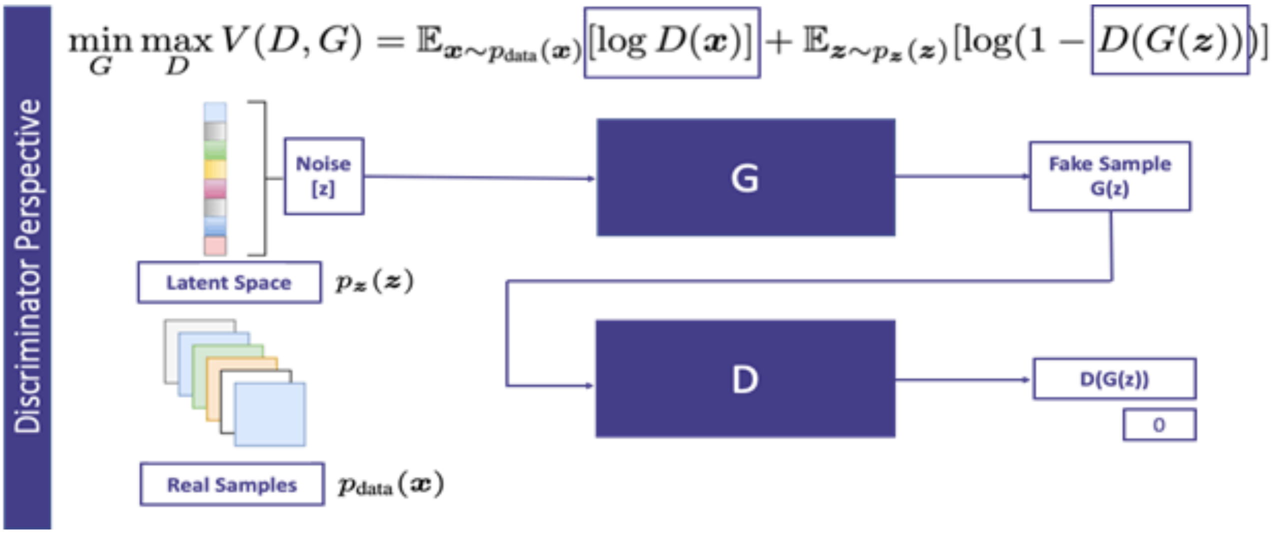
Discriminator models is the probability of an example being fake given a set of input X. It will look at the image of fake cats and determined that they are about 80% probability it isn't the real one, so it will classify as **FAKE**.

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

How about the optimizer?

The optimization is a min-max game. The generator wants to minimize the objective function, whereas the discriminator wants to maximize the same objective function.



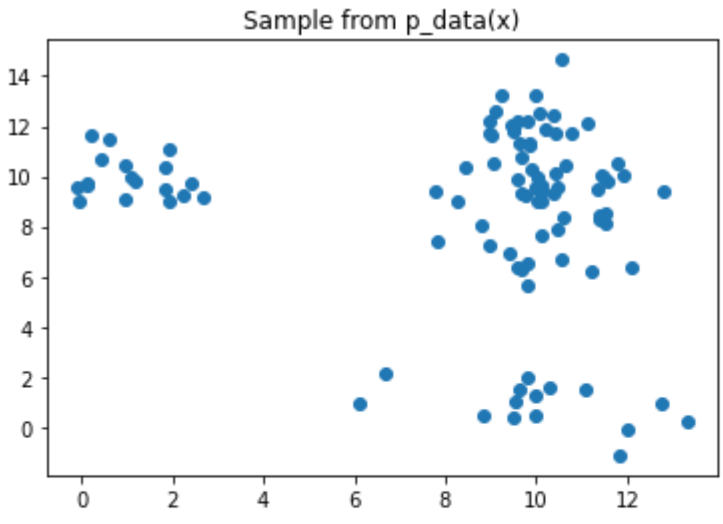
Example 1: Generate a mixture of Gaussians

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

In [1]:
```
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)  # Resul
t: binary matrix of size (m x k)
    for i in range(m):
        # Convert one-hot representation z_vectors[i,:] to an ind
ex
        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()
```

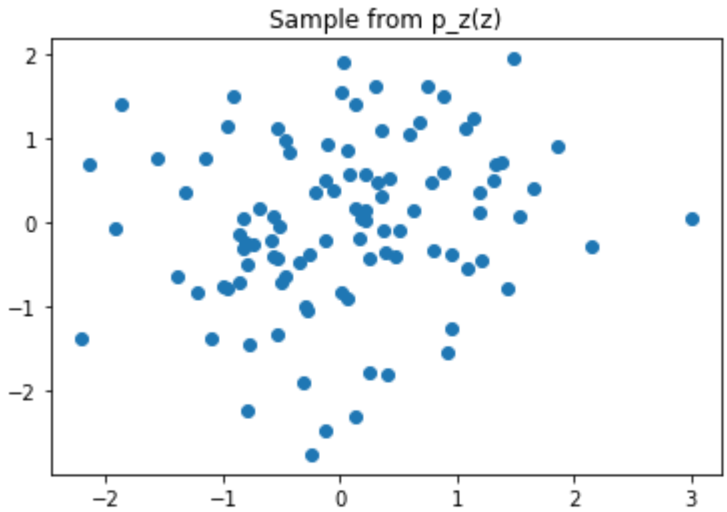


Next we need a function to sample from the noise distribution:

In [5]:
```
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 [6]:
```
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 [26]:
```
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 = self.output2(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:

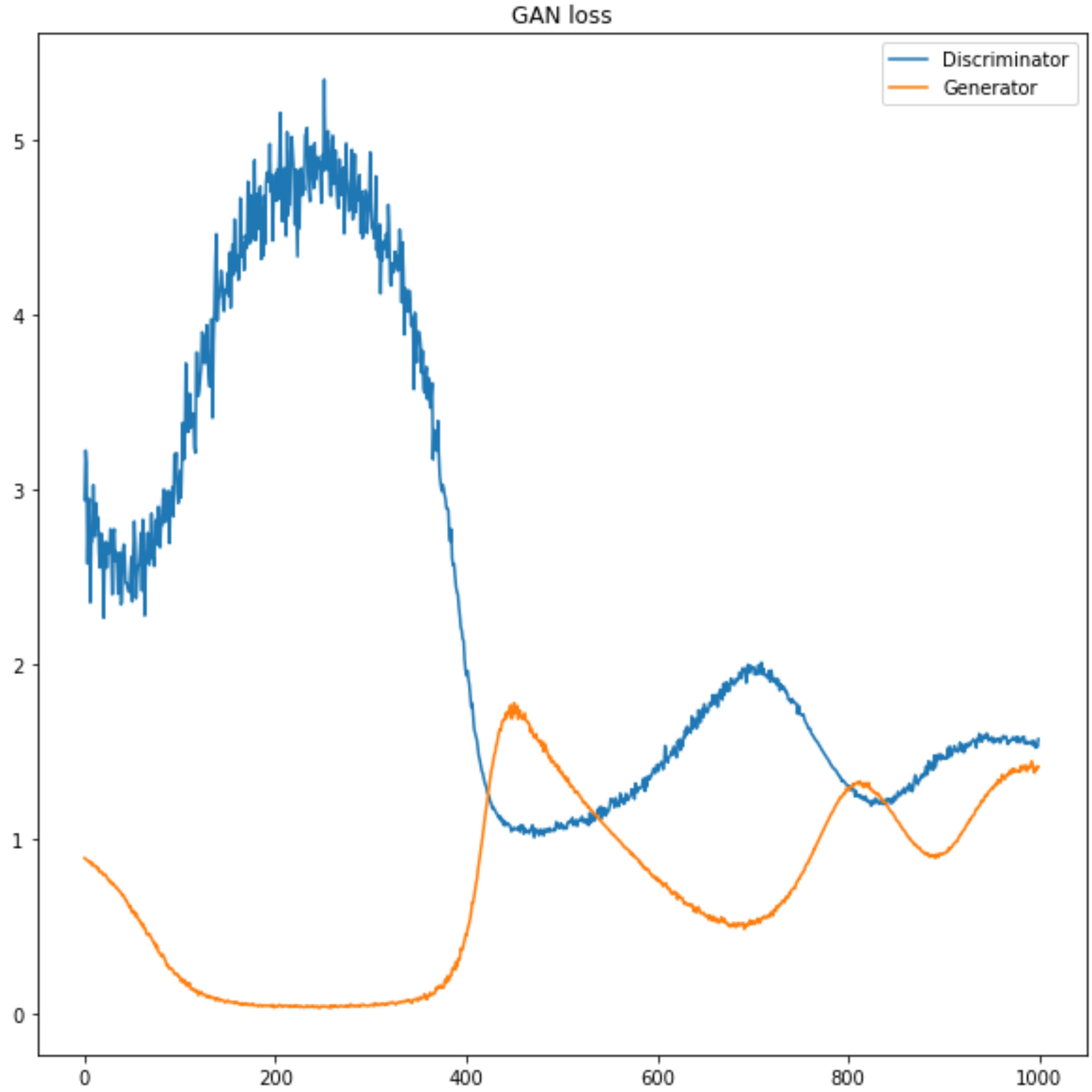
In [27]:
```
# Instantiate the generator and discriminator

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([[-2.2878, -0.3623],
        [-1.1662,  0.6565],
        [ 0.0986, -1.0655],
        [-0.8500,  1.5014],
        [-0.0608, -1.2202],
        [ 0.2323,  0.9980],
        [-1.8262, -1.5295],
        [-0.4151, -0.4695],
        [ 0.5339, -1.0977],
        [ 0.4249,  1.9014]])
Generator output: tensor([[ 0.3076, -0.0243],
        [ 0.3095, -0.0866],
        [ 0.1654, -0.0271],
        [ 0.3305, -0.0973],
        [ 0.1596, -0.0207],
        [ 0.2510, -0.0747],
        [ 0.2895,  0.0144],
        [ 0.2321, -0.0530],
        [ 0.1393, -0.0322],
        [ 0.3250, -0.0518]], grad_fn=<AddmmBackward>)
Discriminator output for generated data: tensor([[0.5195],
        [0.5177],
        [0.4988],
        [0.5204],
        [0.4982],
        [0.5096],
        [0.5182],
        [0.5076],
        [0.4949],
        [0.5211]], grad_fn=<SigmoidBackward>)
Discriminator output for real data: tensor([[0.9982],
        [0.9935],
        [0.9194],
        [0.9995],
        [0.9983],
        [0.9997],
        [0.8389],
        [0.9996],
        [0.9979],
        [0.9988]], grad_fn=<SigmoidBackward>)
```

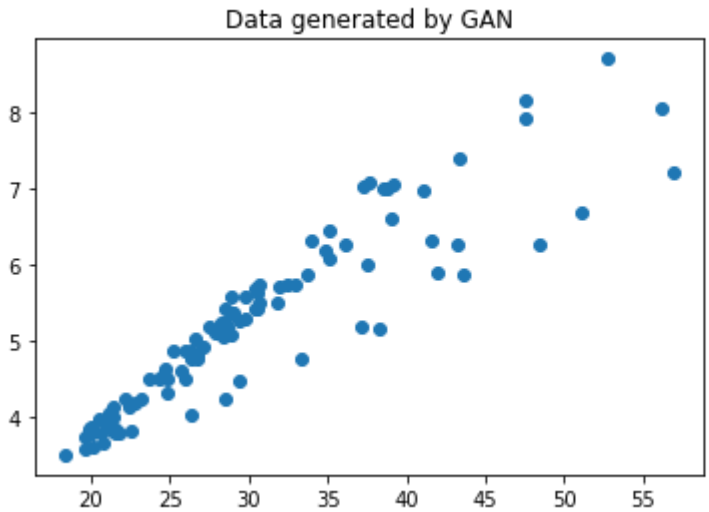
Let's write some code to train these models using the algorithm from Goodfellow et al. (2014):

In [28]:
```
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 minib
atches

    d_loss = 0
    for discriminator_iter in range(num_minibatches_discriminato
r):
        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 [29]:
```
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 [1]:
```
# Write your solution
```

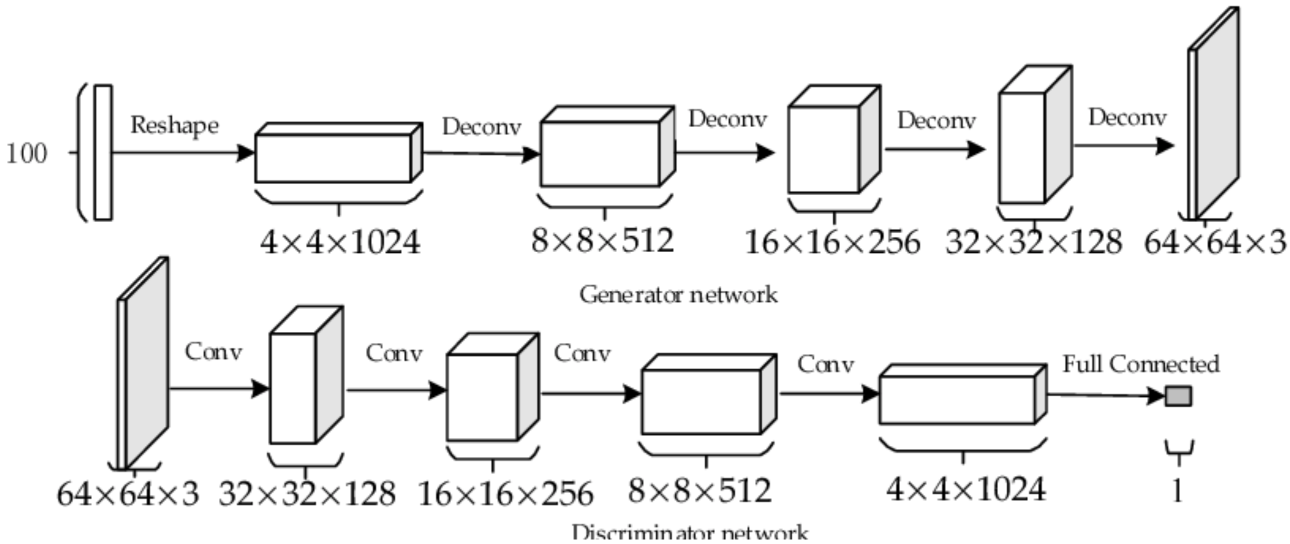
Example 2: Deep Convolutional GAN

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

In [2]:
```
%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[2]:
```
<torch._C.Generator at 0x7f7eb887ccb0>
```

Define some important variables

In [ ]:
```
# Root directory for dataset
dataroot = "data"

# 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 thi
s is 3
nc = 3

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

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 100 # 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
```

Create and preview the dataset

In [ ]:
```
# 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=batc
h_size,
                                         shuffle=True, num_worker
s=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(devic
e)[:64], padding=2, normalize=True).cpu(),(1,2,0)))
```

DCGAN Implementation

Initialize Weights

In [ ]:
```
# 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 model

In [ ]:
```
# 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=Fals
e),
            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=Fa
lse),
            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=F
alse),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=Fals
e),
            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 [ ]:
```
# 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 weig
hts
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

Discriminator model

In [ ]:
```
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 [ ]:
```
# 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 weig
hts
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

Loss functions and Optimizers

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

Train it

This is instruction concept.

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 [ ]:
```
# Training Loop

from IPython.display import clear_output

# 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.floa
t32, 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 batche
s
        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 gener
ator cost
        # Since we just updated D, perform another forward pass o
f 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()

         # 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 o
n fixed_noise
        if (i % 50 == 0) or ((epoch == num_epochs-1) and (i == le
n(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, nor
malize=True))

        # Output training stats
        if i % 50 == 0:
            clear_output(wait=True)
            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_z
2))

            fig = plt.figure(figsize=(10,10))
            plt.imshow(np.transpose(img_list[-1],(1,2,0)))
            plt.show()
```

Loss plot

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

Training process

In [ ]:
```
#%%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_d
elay=1000, blit=True)

HTML(ani.to_jshtml())
```

Final Results

In [ ]:
```
# 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(devic
e)[: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()
```

Take-home exercise (50 points)

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

In [ ]:

In [ ]:

In [ ]: