- Import

In [ ]:
```
%matplotlib inline
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import torch.nn.functional as F
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
from PIL import Image
from torch import autograd
from torch.autograd import Variable
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
import os
import shutil

import torchvision.utils as vutils
import matplotlib.animation as animation
from IPython.display import HTML
```

In [ ]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

print('torch version:',torch.__version__)
print('device:', device)
```

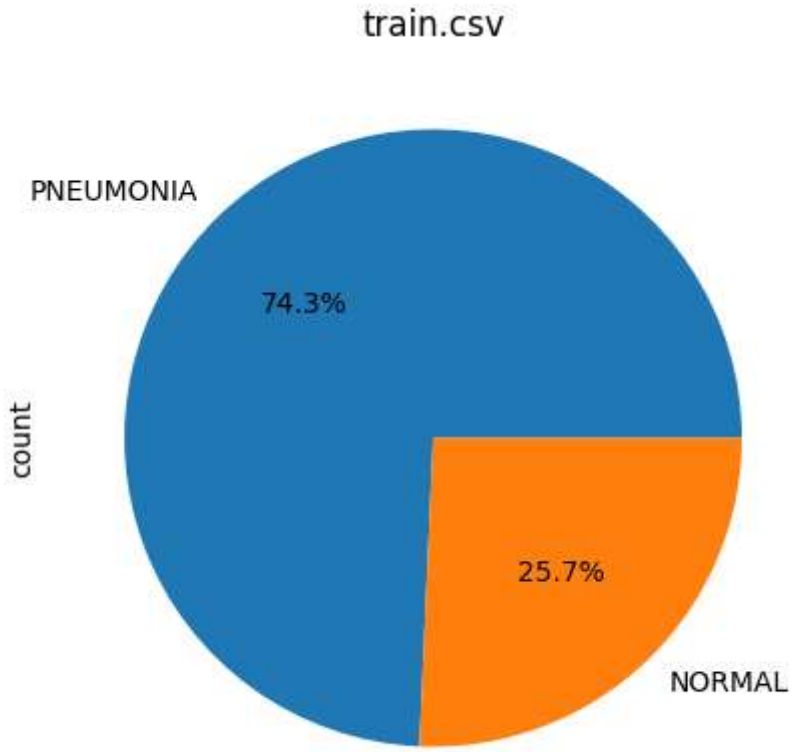
```
torch version: 2.2.1+cu121
device: cuda
```

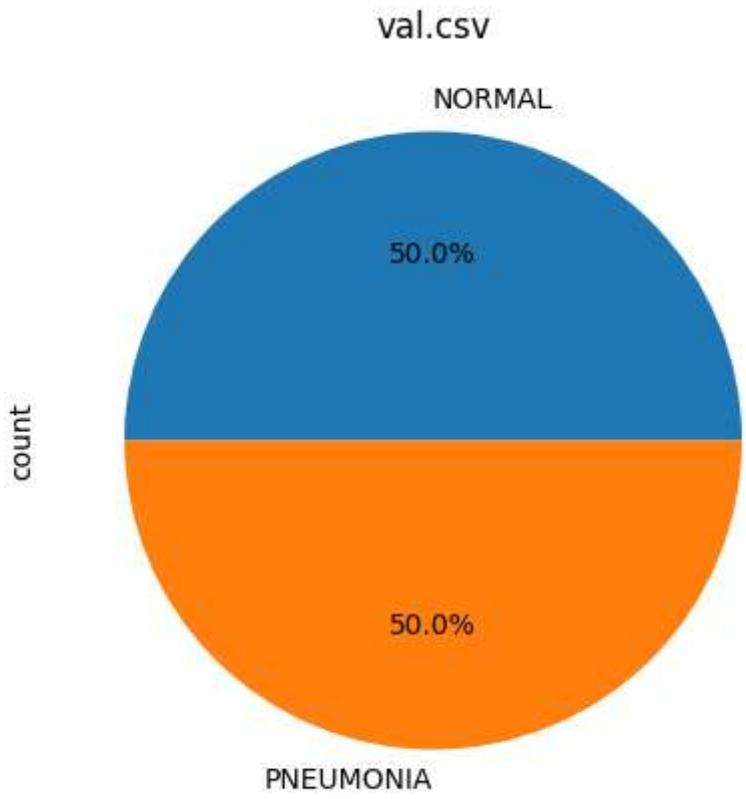
- Parameters

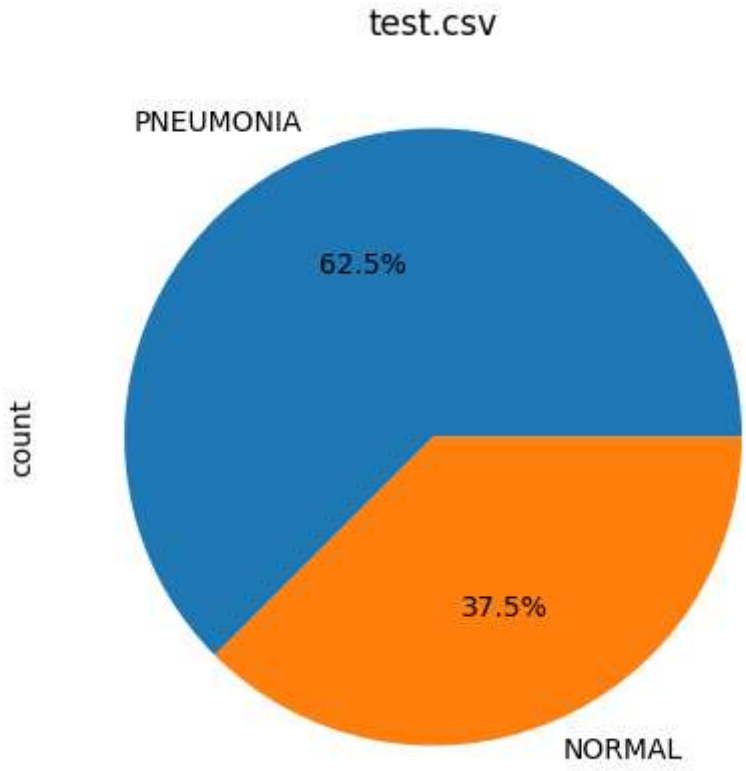
In [ ]:
```
os.listdir('./data/archive/chest_xray')
# Function to create a csv containing image path and labels from folders of images
def create_csv(data_dir, csv_name):
    data = []
    for folder in os.listdir(data_dir):
        if folder in ['NORMAL', 'PNEUMONIA']:
            for file in os.listdir(data_dir + folder):
                data.append([data_dir + folder + '/' + file, folder])
    df = pd.DataFrame(data, columns=['image', 'label'])
    df.to_csv(csv_name, index=False)
    print('CSV created')
    # Print and format number of instances in each class and total instances
    print(df['label'].value_counts())
    # Draw plot for percentage of instances in each class and add labels
    plt.figure(figsize=(5, 5))
    df['label'].value_counts().plot(kind='pie', autopct='%1.1f%%').set_title(csv_na
    # Save csv to suitable location
    df.to_csv(csv_name, index=False)
df
```

In [ ]:
```
train_csv = create_csv('./data/archive/chest_xray/train/', 'train.csv')
val_csv = create_csv('./data/archive/chest_xray/val/', 'val.csv')
test_csv = create_csv('./data/archive/chest_xray/test/', 'test.csv')
```

```
CSV created
label
PNEUMONIA    3875
NORMAL       1341
Name: count, dtype: int64
CSV created
label
NORMAL       8
PNEUMONIA    8
Name: count, dtype: int64
CSV created
label
PNEUMONIA    390
NORMAL       234
Name: count, dtype: int64
```







In [ ]:
```
# Data
train_data_path = './train.csv' # Path of data
valid_data path = './test.csv' # Path of data
n data path:', train_data_path)
```

```
print('Valid data path:', valid_data_path)

img_size = 64 # Image size
batch_size = 64  # Batch size

# Model
z_size = 100
generator_layer_size = [256, 512, 1024]
discriminator_layer_size = [1024, 512, 256]

# Training
epochs = 30  # Train epochs
learning_rate = 1e-4
```

```
Train data path: ./train.csv
Valid data path: ./test.csv
```

- Pytorch Dataset, DataLoader: Chest Xray

In [ ]:
```
class_list = ['NORMAL', 'PNEUMONIA']
class_num = len(class_list)
```

In [ ]:
```
# Function to find normalizing parameters for data
def find_mean_std(data_path):
    df = pd.read_csv(data_path)
    mean = 0.
    std = 0.
    nb_samples = 0.
    for i in range(len(df)):
        img = Image.open(df['image'][i])
        img = np.array(img)/255
        mean += np.mean(img)
        std += np.std(img)
        nb_samples += 1.
    mean /= nb_samples
    std /= nb_samples
    return mean, std
```

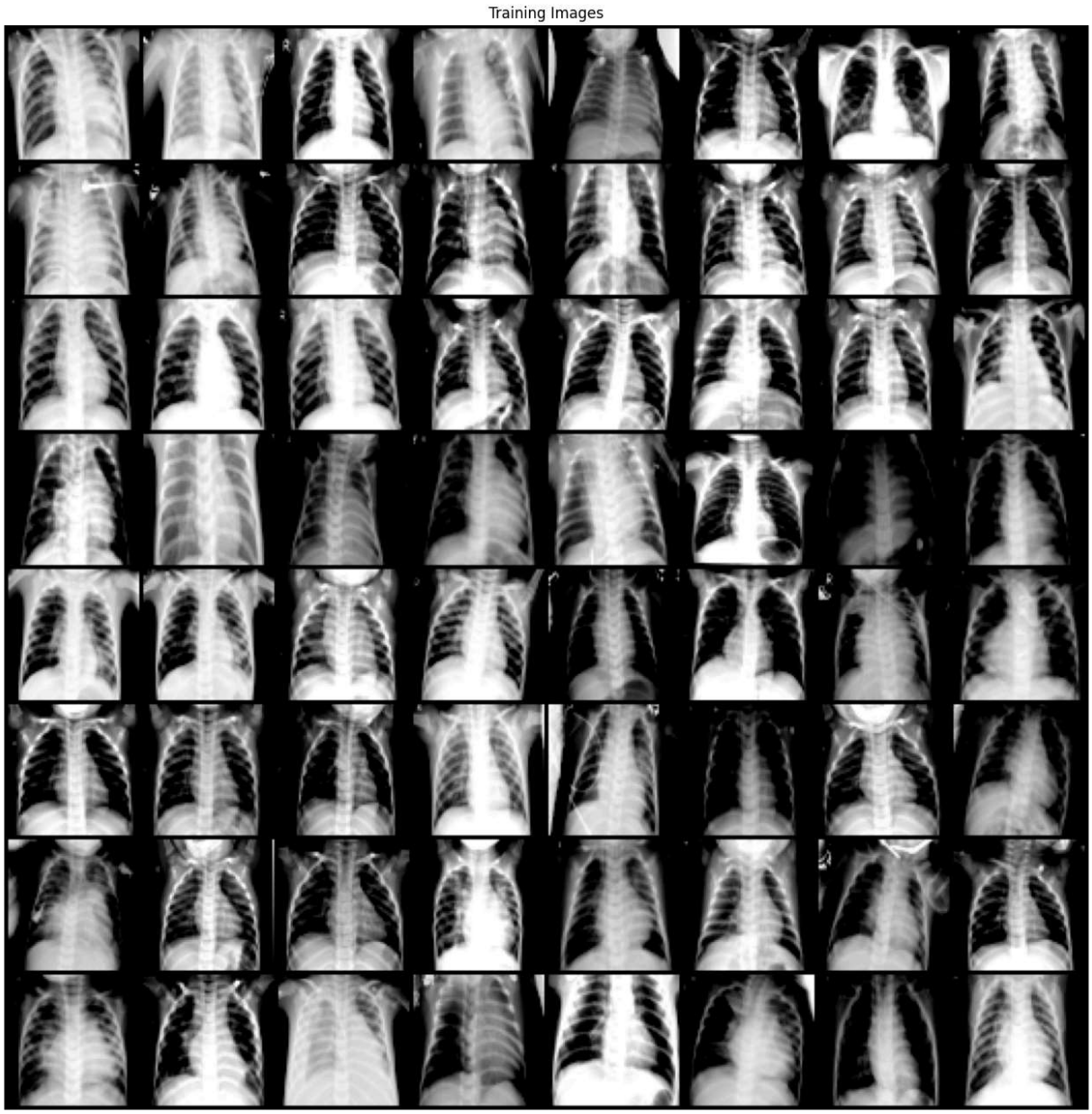
In [ ]:
```
# Pytorch dataset that loads images and labels from csv
class ChestXrayDataset(Dataset):
    def __init__(self, csv_file, transform=None):
        self.data = pd.read_csv(csv_file)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_name = self.data.iloc[idx, 0]
        image = Image.open(img_name).convert('RGB')
        label = 1 if self.data.iloc[idx, 1] == 'PNEUMONIA' else 0
        if self.transform:
            image = self.transform(image)
        return image, label
```

```
# Pytorch dataloader that loads data from dataset and resizes image to 256x256
transform = transforms.Compose([
    transforms.Resize((img_size, img_size)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.354,), std=(0.4435,))
])
train_data = ChestXrayDataset(train_data_path, transform)
valid_data = ChestXrayDataset(valid_data_path, transform)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True,drop_last
```

In [ ]:

```
# Check if dataloader is working
for i, (images, labels) in enumerate(train_loader):
    print(images.shape, labels)
    # Show images
    plt.figure(figsize=(16, 24))
    plt.axis("off")
    plt.title("Training Images")
    # Make grid from images and plot labels
    plt.imshow(np.transpose(make_grid(images, nrow=8).cpu(), (1, 2, 0)))
    break
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
torch.Size([64, 3, 64, 64]) tensor([1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0,
1, 1, 1, 0, 0, 1, 0, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1,
        1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1])
```



Implementation using Deep Convolutional GAN (DCGAN)

In [ ]:
```
# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 64

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3
```

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

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 20

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

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

- Generator

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)

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 [ ]:
```
# 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=0.02``.
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 [ ]:
```
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 weights
# like this: ``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=Fals
e)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fal
se)
    (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=Fa
lse)
    (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=Fa
lse)
    (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()
  )
)
```

- Adversarial Learning of Generator & Discriminator

In [ ]:
```
# Initialize the ``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 = torch.optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

In [ ]:
```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")


epoch
```

```
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(train_loader, 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 previou
        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 t
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
```

```
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)):
                  % (epoch, num_epochs, i, len(train_loader),
                     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(train_loade
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

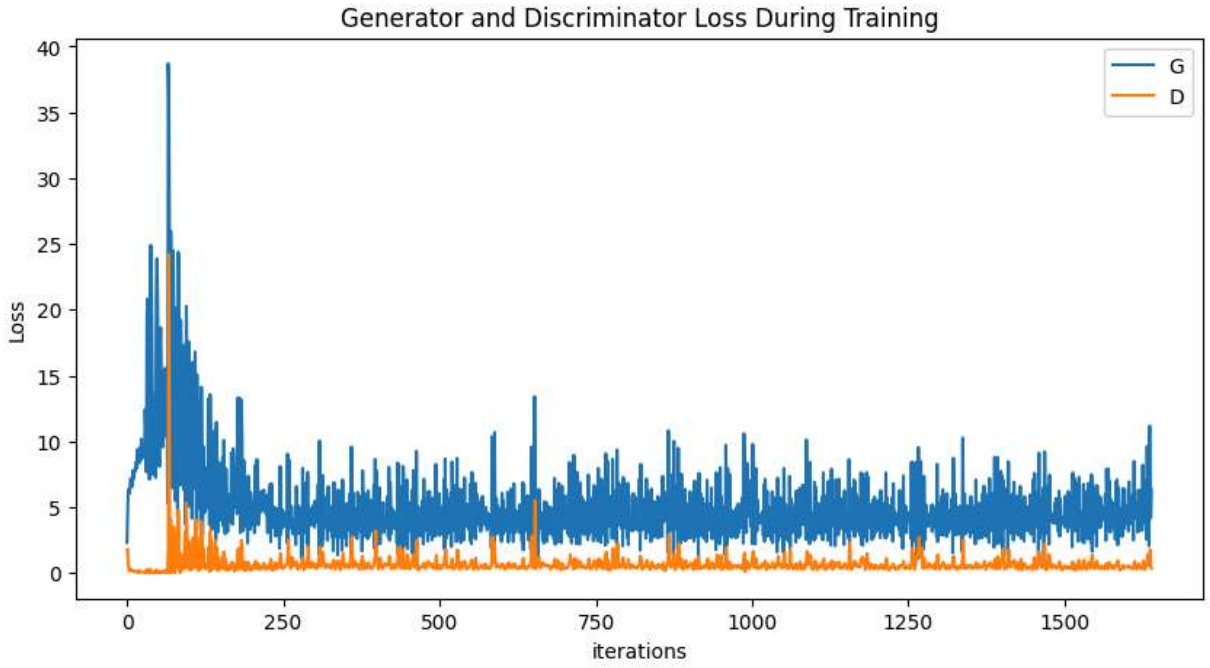
```
Starting Training Loop...
[0/20][0/82]    Loss_D: 1.7407  Loss_G: 2.3106  D(x): 0.2827    D(G(z)): 0.2423 / 0.
1295
[0/20][50/82]   Loss_D: 0.0124  Loss_G: 16.2756 D(x): 0.9894    D(G(z)): 0.0000 / 0.
0000
[1/20][0/82]    Loss_D: 4.7998  Loss_G: 24.3907 D(x): 0.9899    D(G(z)): 0.9777 / 0.
0000
[1/20][50/82]   Loss_D: 0.1183  Loss_G: 3.6803  D(x): 0.9385    D(G(z)): 0.0312 / 0.
0342
[2/20][0/82]    Loss_D: 0.2286  Loss_G: 3.9317  D(x): 0.9118    D(G(z)): 0.0998 / 0.
0230
[2/20][50/82]   Loss_D: 0.4289  Loss_G: 6.1952  D(x): 0.8957    D(G(z)): 0.2427 / 0.
0042
[3/20][0/82]    Loss_D: 1.4852  Loss_G: 2.9272  D(x): 0.3349    D(G(z)): 0.0064 / 0.
0953
[3/20][50/82]   Loss_D: 0.8346  Loss_G: 4.1261  D(x): 0.7215    D(G(z)): 0.2580 / 0.
0251
[4/20][0/82]    Loss_D: 0.6448  Loss_G: 2.7380  D(x): 0.7312    D(G(z)): 0.1719 / 0.
0856
[4/20][50/82]   Loss_D: 0.6988  Loss_G: 5.2372  D(x): 0.6154    D(G(z)): 0.0138 / 0.
0138
[5/20][0/82]    Loss_D: 0.5077  Loss_G: 5.3517  D(x): 0.9037    D(G(z)): 0.2959 / 0.
0103
[5/20][50/82]   Loss_D: 0.3918  Loss_G: 4.2161  D(x): 0.9563    D(G(z)): 0.2669 / 0.
0228
[6/20][0/82]    Loss_D: 0.3618  Loss_G: 4.4418  D(x): 0.8715    D(G(z)): 0.1738 / 0.
0209
[6/20][50/82]   Loss_D: 0.4324  Loss_G: 4.2630  D(x): 0.9370    D(G(z)): 0.2632 / 0.
0308
[7/20][0/82]    Loss_D: 0.5491  Loss_G: 4.9970  D(x): 0.8226    D(G(z)): 0.2418 / 0.
0195
[7/20][50/82]   Loss_D: 0.3357  Loss_G: 4.1042  D(x): 0.9055    D(G(z)): 0.1946 / 0.
0263
[8/20][0/82]    Loss_D: 0.6935  Loss_G: 2.6154  D(x): 0.6329    D(G(z)): 0.0398 / 0.
1278
[8/20][50/82]   Loss_D: 0.4503  Loss_G: 5.9750  D(x): 0.9674    D(G(z)): 0.3067 / 0.
0106
[9/20][0/82]    Loss_D: 0.7605  Loss_G: 7.4600  D(x): 0.9330    D(G(z)): 0.4423 / 0.
0037
[9/20][50/82]   Loss_D: 0.5188  Loss_G: 3.6532  D(x): 0.6751    D(G(z)): 0.0260 / 0.
0569
[10/20][0/82]   Loss_D: 0.5745  Loss_G: 7.1958  D(x): 0.9068    D(G(z)): 0.3416 / 0.
0028
[10/20][50/82]  Loss_D: 0.4096  Loss_G: 3.3341  D(x): 0.8156    D(G(z)): 0.1316 / 0.
0893
[11/20][0/82]   Loss_D: 0.7752  Loss_G: 2.6726  D(x): 0.5755    D(G(z)): 0.0471 / 0.
1064
[11/20][50/82]  Loss_D: 0.2457  Loss_G: 4.8308  D(x): 0.8451    D(G(z)): 0.0525 / 0.
0127
[12/20][0/82]   Loss_D: 0.3273  Loss_G: 3.6274  D(x): 0.8262    D(G(z)): 0.0987 / 0.
0385
[12/20][50/82]  Loss_D: 0.5837  Loss_G: 5.7864  D(x): 0.9705    D(G(z)): 0.3896 / 0.
0086
[13/20][0/82]   Loss_D: 0.4423  Loss_G: 3.0666  D(x): 0.7822    D(G(z)): 0.1315 / 0.
0785
[13/20][50/82]  Loss_D: 0.2739  Loss_G: 3.0641  D(x): 0.9315    D(G(z)): 0.1696 / 0.
```

```
0680
[14/20][0/82]   Loss_D: 0.7148  Loss_G: 6.2585  D(x): 0.9269    D(G(z)): 0.4287 / 0.
0045
[14/20][50/82]  Loss_D: 0.4180  Loss_G: 4.8717  D(x): 0.9475    D(G(z)): 0.2835 / 0.
0170
[15/20][0/82]   Loss_D: 0.2832  Loss_G: 3.9347  D(x): 0.8723    D(G(z)): 0.1018 / 0.
0393
[15/20][50/82]  Loss_D: 0.3909  Loss_G: 3.9928  D(x): 0.8750    D(G(z)): 0.2004 / 0.
0261
[16/20][0/82]   Loss_D: 0.5567  Loss_G: 6.0385  D(x): 0.9308    D(G(z)): 0.3533 / 0.
0049
[16/20][50/82]  Loss_D: 0.4988  Loss_G: 2.5990  D(x): 0.6964    D(G(z)): 0.0415 / 0.
0999
[17/20][0/82]   Loss_D: 0.9614  Loss_G: 4.0359  D(x): 0.5219    D(G(z)): 0.0234 / 0.
0839
[17/20][50/82]  Loss_D: 0.4242  Loss_G: 3.2042  D(x): 0.7702    D(G(z)): 0.0610 / 0.
0543
[18/20][0/82]   Loss_D: 1.1599  Loss_G: 7.4466  D(x): 0.9450    D(G(z)): 0.6023 / 0.
0028
[18/20][50/82]  Loss_D: 0.2849  Loss_G: 4.6865  D(x): 0.9165    D(G(z)): 0.1560 / 0.
0126
[19/20][0/82]   Loss_D: 0.4205  Loss_G: 3.4973  D(x): 0.7462    D(G(z)): 0.0666 / 0.
0660
[19/20][50/82]  Loss_D: 0.3038  Loss_G: 3.8652  D(x): 0.8375    D(G(z)): 0.0802 / 0.
0320
```

In [ ]:
```
# Save model weights
torch.save(netG.state_dict(), 'generator.pth')
torch.save(netD.state_dict(), 'discriminator.pth')
```

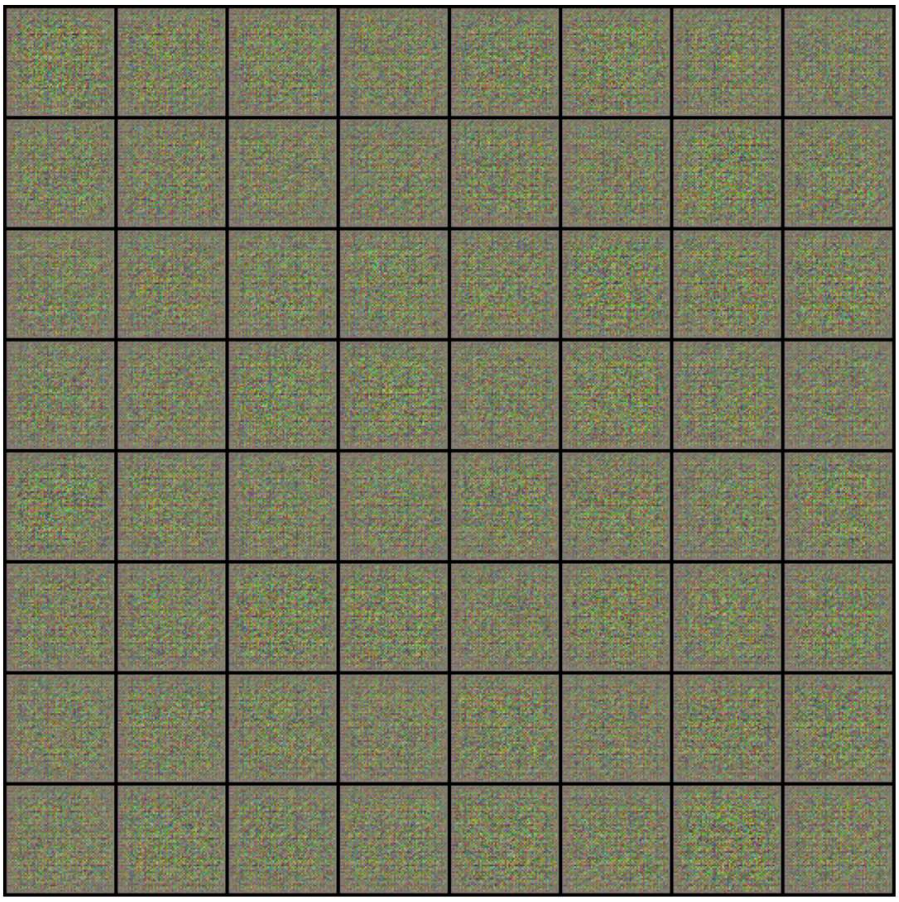
Plot Loss and images

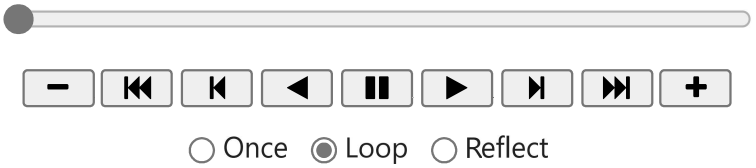
In [ ]:
```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [ ]:
```
fig = plt.figure(figsize=(14,14))
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=Tr

HTML(ani.to_jshtml())
```

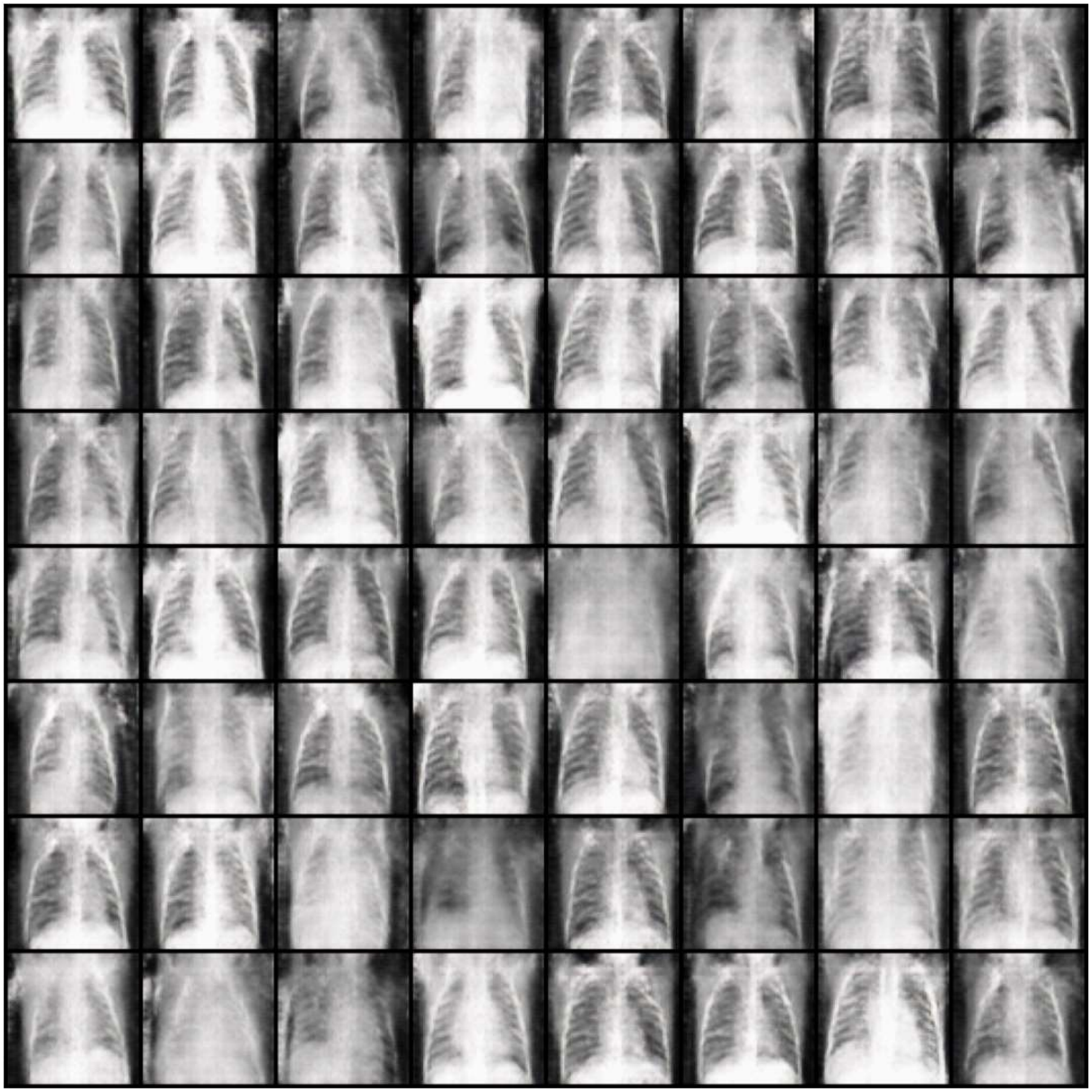
Out[ ]:





Once Loop Reflect

Loading [MathJax]/extensions/Safe.js



Implementation using Conditional GAN (cGANa)

In [ ]:
```
# Arguments
BATCH_SIZE = 64
Z_DIM = 10
LABEL_EMBED_SIZE = 5
NUM_CLASSES = 2
IMGS_TO_DISPLAY_PER_CLASS = 8
LOAD_MODEL = False

DB = 'CHEST_XRAY'
CHANNELS = 3
EPOCHS = 50


# Directories for storing data, model and output samples
s.path.join('./data', DB)
```

```
os.makedirs(db_path, exist_ok=True)
model_path = os.path.join('./model', DB)
os.makedirs(model_path, exist_ok=True)
samples_path = os.path.join('./samples', DB)
os.makedirs(samples_path, exist_ok=True)
```

In [ ]:

```
# Method for storing generated images
def generate_imgs(gen, z, fixed_label, img_list, epoch=0):
    gen.eval()
    fake_imgs = gen(z, fixed_label)
    fake_imgs = (fake_imgs + 1) / 2
    fake_imgs_ = vutils.make_grid(fake_imgs, normalize=False, nrow=IMGS_TO_DISPLAY_
    img_list.append(fake_imgs_)
    vutils.save_image(fake_imgs_, os.path.join(samples_path, 'sample_' + str(epoch)


# Networks
def conv_block(c_in, c_out, k_size=4, stride=2, pad=1, use_bn=True, transpose=False
    module = []
    if transpose:
        module.append(nn.ConvTranspose2d(c_in, c_out, k_size, stride, pad, bias=not
    else:
        module.append(nn.Conv2d(c_in, c_out, k_size, stride, pad, bias=not use_bn))
    if use_bn:
        module.append(nn.BatchNorm2d(c_out))
    return nn.Sequential(*module)
```

In [ ]:

```
class Generator(nn.Module):
    def __init__(self, z_dim=10, num_classes=2, label_embed_size=5, channels=3, con
        super(Generator, self).__init__()
        self.label_embedding = nn.Embedding(num_classes, label_embed_size)
        self.tconv1 = conv_block(z_dim + label_embed_size, conv_dim * 4, pad=0, tra
        self.tconv2 = conv_block(conv_dim * 4, conv_dim * 2, transpose=True)
        self.tconv3 = conv_block(conv_dim * 2, conv_dim * 2, transpose=True)
        self.tconv4 = conv_block(conv_dim * 2, conv_dim, transpose=True)
        self.tconv5 = conv_block(conv_dim, channels, transpose=True, use_bn=False)

        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
                nn.init.normal_(m.weight, 0.0, 0.02)

            if isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x, label):
        x = x.reshape([x.shape[0], -1, 1, 1])
        label_embed = self.label_embedding(label)
        label_embed = label_embed.reshape([label_embed.shape[0], -1, 1, 1])
        x = torch.cat((x, label_embed), dim=1)
        x = F.relu(self.tconv1(x))
        x = F.relu(self.tconv2(x))
        x = F.relu(self.tconv3(x))
        x = F.relu(self.tconv4(x))
```

```
        x = torch.tanh(self.tconv5(x))
        return x
```

In [ ]:
```
class Discriminator(nn.Module):
    def __init__(self, num_classes=2, channels=3, conv_dim=64):
        super(Discriminator, self).__init__()
        self.image_size = img_size
        self.label_embedding = nn.Embedding(num_classes, self.image_size*self.image
        self.conv1 = conv_block(channels + 1, conv_dim, use_bn=False)
        self.conv2 = conv_block(conv_dim, conv_dim * 2)
        self.conv3 = conv_block(conv_dim * 2, conv_dim * 4)
        self.conv4 = conv_block(conv_dim * 4, conv_dim * 6)
        self.conv5 = conv_block(conv_dim * 6, 1, k_size=4, stride=1, pad=0, use_bn=

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.normal_(m.weight, 0.0, 0.02)

            if isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x, label):
        alpha = 0.2
        label_embed = self.label_embedding(label)
        label_embed = label_embed.reshape([label_embed.shape[0], 1, self.image_size
        x = torch.cat((x, label_embed), dim=1)
        x = F.leaky_relu(self.conv1(x), alpha)
        x = F.leaky_relu(self.conv2(x), alpha)
        x = F.leaky_relu(self.conv3(x), alpha)
        x = F.leaky_relu(self.conv4(x), alpha)
        x = torch.sigmoid(self.conv5(x))
        return x.squeeze()
```

In [ ]:
```
gen = Generator(z_dim=Z_DIM, num_classes=NUM_CLASSES, label_embed_size=LABEL_EMBED_
dis = Discriminator(num_classes=NUM_CLASSES, channels=CHANNELS)

# Load previous model
if LOAD_MODEL:
    gen.load_state_dict(torch.load(os.path.join(model_path, 'gen.pkl')))
    dis.load_state_dict(torch.load(os.path.join(model_path, 'dis.pkl')))

# Model Summary
print("------------------Generator------------------")
print(gen)
print("------------------Discriminator------------------")
print(dis)
```

```
------------------Generator------------------
Generator(
  (label_embedding): Embedding(2, 5)
  (tconv1): Sequential(
    (0): ConvTranspose2d(15, 256, kernel_size=(4, 4), stride=(2, 2), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (tconv2): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (tconv3): Sequential(
    (0): ConvTranspose2d(128, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (tconv4): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
rue)
  )
  (tconv5): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
)
------------------Discriminator------------------
Discriminator(
  (label_embedding): Embedding(2, 4096)
  (conv1): Sequential(
    (0): Conv2d(4, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (conv2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (conv3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fa
lse)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (conv4): Sequential(
    (0): Conv2d(256, 384, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fa
lse)
    (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
  )
  (conv5): Sequential(
    (0): Conv2d(384, 1, kernel_size=(4, 4), stride=(1, 1))
```

```
        )
    )
```

In [ ]:
```
# Define Optimizers
g_opt = torch.optim.Adam(gen.parameters(), lr=0.00005, betas=(0.5, 0.999), weight_d
d_opt = torch.optim.Adam(dis.parameters(), lr=0.00005, betas=(0.5, 0.999), weight_d

# Loss functions
loss_fn = nn.BCELoss()

# Fix images for viz
fixed_z = torch.randn(IMGS_TO_DISPLAY_PER_CLASS*NUM_CLASSES, Z_DIM)
fixed_label = torch.arange(0, NUM_CLASSES)
fixed_label = torch.repeat_interleave(fixed_label, IMGS_TO_DISPLAY_PER_CLASS)
```

In [ ]:
```
len(train_loader)
```

Out[ ]:
```
81
```

In [ ]:
```
# Labels
real_label = torch.ones(BATCH_SIZE)
fake_label = torch.zeros(BATCH_SIZE)
img_list = []
G_losses = []
D_losses = []

# GPU Compatibility
is_cuda = torch.cuda.is_available()
if is_cuda:
    gen, dis = gen.cuda(), dis.cuda()
    real_label, fake_label = real_label.cuda(), fake_label.cuda()
    fixed_z, fixed_label = fixed_z.cuda(), fixed_label.cuda()

total_iters = 0
max_iter = len(train_loader)

# Training
for epoch in range(EPOCHS):
    gen.train()
    dis.train()

    for i, data in enumerate(train_loader):

        total_iters += 1

        # Loading data
        x_real, x_label = data
        z_fake = torch.randn(BATCH_SIZE, Z_DIM)

        if is_cuda:
            x_real = x_real.cuda()
            x_label = x_label.cuda()
            z_fake = z_fake.cuda()

        # Generate fake data
```

```
        x_fake = gen(z_fake, x_label)
        # Train Discriminator
        fake_out = dis(x_fake.detach(), x_label)
        real_out = dis(x_real.detach(), x_label)

        d_loss = (loss_fn(fake_out, fake_label) + loss_fn(real_out, real_label)) /

        d_opt.zero_grad()
        d_loss.backward()
        d_opt.step()

        # Train Generator
        fake_out = dis(x_fake, x_label)
        g_loss = loss_fn(fake_out, real_label)

        g_opt.zero_grad()
        g_loss.backward()
        g_opt.step()

        # Save Losses for plotting later
        G_losses.append(g_loss.item())
        D_losses.append(d_loss.item())

        if i % 50 == 0:
            print("Epoch: " + str(epoch + 1) + "/" + str(EPOCHS)
                  + "\titer: " + str(i) + "/" + str(max_iter)
                  + "\ttotal_iters: " + str(total_iters)
                  + "\td_loss:" + str(round(d_loss.item(), 4))
                  + "\tg_loss:" + str(round(g_loss.item(), 4))
                  )

    if (epoch + 1) % 5 == 0:
        torch.save(gen.state_dict(), os.path.join(model_path, 'gen.pkl'))
        torch.save(dis.state_dict(), os.path.join(model_path, 'dis.pkl'))



        generate_imgs(gen, fixed_z, fixed_label,img_list, epoch=epoch + 1)
generate_imgs(gen, fixed_z, fixed_label, img_list)
```

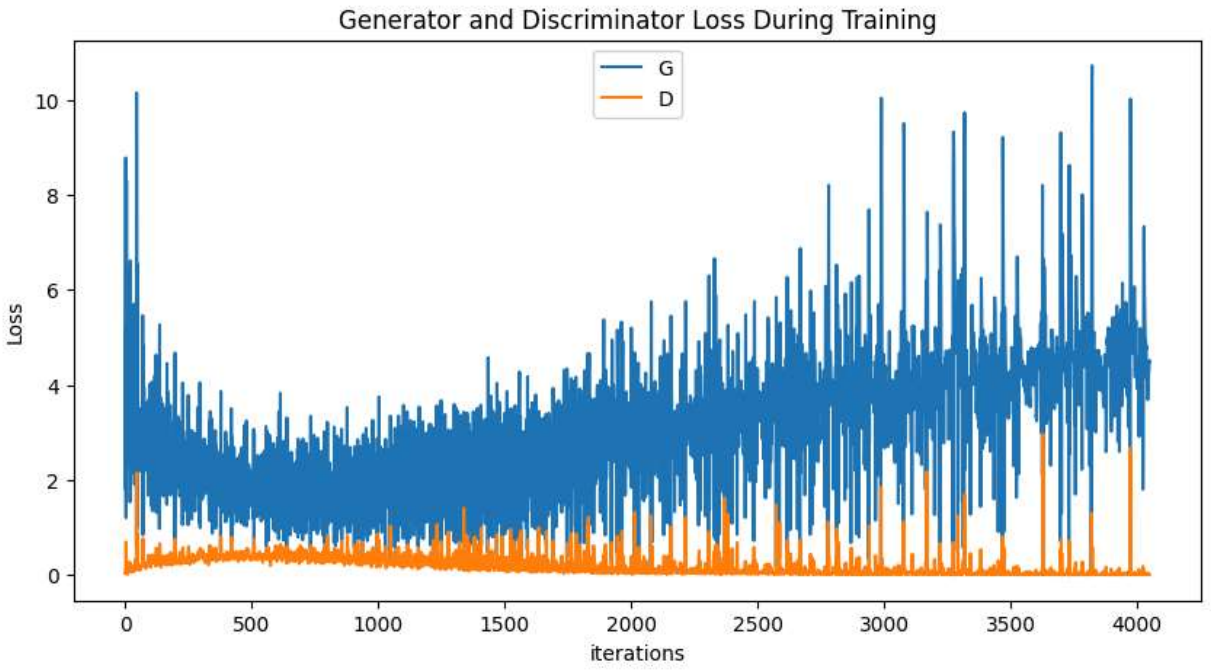
```
Epoch: 1/50    iter: 0/81     total_iters: 1  d_loss:0.0382   g_loss:3.4649
Epoch: 1/50    iter: 50/81    total_iters: 51 d_loss:0.4847   g_loss:4.1603
Epoch: 2/50    iter: 0/81     total_iters: 82 d_loss:0.1927   g_loss:2.5483
Epoch: 2/50    iter: 50/81    total_iters: 132        d_loss:0.2168   g_loss:2.648
8
Epoch: 3/50    iter: 0/81     total_iters: 163        d_loss:0.2546   g_loss:2.135
9
Epoch: 3/50    iter: 50/81    total_iters: 213        d_loss:0.2733   g_loss:1.931
6
Epoch: 4/50    iter: 0/81     total_iters: 244        d_loss:0.3348   g_loss:1.950
7
Epoch: 4/50    iter: 50/81    total_iters: 294        d_loss:0.3471   g_loss:1.896
Epoch: 5/50    iter: 0/81     total_iters: 325        d_loss:0.4182   g_loss:1.932
9
Epoch: 5/50    iter: 50/81    total_iters: 375        d_loss:0.3246   g_loss:1.874
8
Epoch: 6/50    iter: 0/81     total_iters: 406        d_loss:0.3496   g_loss:1.945
6
Epoch: 6/50    iter: 50/81    total_iters: 456        d_loss:0.4045   g_loss:1.782
1
Epoch: 7/50    iter: 0/81     total_iters: 487        d_loss:0.4214   g_loss:1.636
1
Epoch: 7/50    iter: 50/81    total_iters: 537        d_loss:0.4439   g_loss:1.855
8
Epoch: 8/50    iter: 0/81     total_iters: 568        d_loss:0.5826   g_loss:0.731
Epoch: 8/50    iter: 50/81    total_iters: 618        d_loss:0.3672   g_loss:2.656
6
Epoch: 9/50    iter: 0/81     total_iters: 649        d_loss:0.2722   g_loss:2.253
5
Epoch: 9/50    iter: 50/81    total_iters: 699        d_loss:0.2957   g_loss:2.380
5
Epoch: 10/50   iter: 0/81     total_iters: 730        d_loss:0.4207   g_loss:1.603
4
Epoch: 10/50   iter: 50/81    total_iters: 780        d_loss:0.314    g_loss:1.967
3
Epoch: 11/50   iter: 0/81     total_iters: 811        d_loss:0.2719   g_loss:2.359
3
Epoch: 11/50   iter: 50/81    total_iters: 861        d_loss:0.2826   g_loss:2.009
4
Epoch: 12/50   iter: 0/81     total_iters: 892        d_loss:0.2811   g_loss:2.114
7
Epoch: 12/50   iter: 50/81    total_iters: 942        d_loss:0.6948   g_loss:2.960
3
Epoch: 13/50   iter: 0/81     total_iters: 973        d_loss:0.2864   g_loss:1.838
5
Epoch: 13/50   iter: 50/81    total_iters: 1023       d_loss:0.2733   g_loss:2.645
7
Epoch: 14/50   iter: 0/81     total_iters: 1054       d_loss:0.3407   g_loss:1.874
9
Epoch: 14/50   iter: 50/81    total_iters: 1104       d_loss:0.2893   g_loss:1.100
5
Epoch: 15/50   iter: 0/81     total_iters: 1135       d_loss:0.1459   g_loss:2.299
1
Epoch: 15/50   iter: 50/81    total_iters: 1185       d_loss:0.2552   g_loss:1.634
2
Epoch: 16/50   iter: 0/81     total_iters: 1216       d_loss:0.1904   g_loss:2.947
```

```
3
Epoch: 16/50    iter: 50/81     total_iters: 1266       d_loss:0.1485   g_loss:3.379
7
Epoch: 17/50    iter: 0/81      total_iters: 1297       d_loss:0.3354   g_loss:1.448
1
Epoch: 17/50    iter: 50/81     total_iters: 1347       d_loss:0.1108   g_loss:3.102
7
Epoch: 18/50    iter: 0/81      total_iters: 1378       d_loss:0.1093   g_loss:3.438
9
Epoch: 18/50    iter: 50/81     total_iters: 1428       d_loss:0.2845   g_loss:2.420
4
Epoch: 19/50    iter: 0/81      total_iters: 1459       d_loss:0.0908   g_loss:2.287
2
Epoch: 19/50    iter: 50/81     total_iters: 1509       d_loss:0.288    g_loss:0.969
2
Epoch: 20/50    iter: 0/81      total_iters: 1540       d_loss:0.4001   g_loss:0.648
8
Epoch: 20/50    iter: 50/81     total_iters: 1590       d_loss:0.1182   g_loss:2.994
7
Epoch: 21/50    iter: 0/81      total_iters: 1621       d_loss:0.1097   g_loss:2.486
5
Epoch: 21/50    iter: 50/81     total_iters: 1671       d_loss:0.0857   g_loss:2.540
9
Epoch: 22/50    iter: 0/81      total_iters: 1702       d_loss:0.0856   g_loss:3.243
6
Epoch: 22/50    iter: 50/81     total_iters: 1752       d_loss:0.1865   g_loss:2.908
2
Epoch: 23/50    iter: 0/81      total_iters: 1783       d_loss:0.0576   g_loss:4.165
2
Epoch: 23/50    iter: 50/81     total_iters: 1833       d_loss:0.8476   g_loss:1.587
9
Epoch: 24/50    iter: 0/81      total_iters: 1864       d_loss:0.1312   g_loss:2.017
9
Epoch: 24/50    iter: 50/81     total_iters: 1914       d_loss:0.1001   g_loss:2.882
5
Epoch: 25/50    iter: 0/81      total_iters: 1945       d_loss:0.0706   g_loss:3.141
8
Epoch: 25/50    iter: 50/81     total_iters: 1995       d_loss:0.0753   g_loss:2.915
6
Epoch: 26/50    iter: 0/81      total_iters: 2026       d_loss:0.209    g_loss:1.555
1
Epoch: 26/50    iter: 50/81     total_iters: 2076       d_loss:0.173    g_loss:1.730
4
Epoch: 27/50    iter: 0/81      total_iters: 2107       d_loss:0.0829   g_loss:3.257
3
Epoch: 27/50    iter: 50/81     total_iters: 2157       d_loss:0.5511   g_loss:0.239
3
Epoch: 28/50    iter: 0/81      total_iters: 2188       d_loss:0.1357   g_loss:2.110
3
Epoch: 28/50    iter: 50/81     total_iters: 2238       d_loss:0.1447   g_loss:2.634
4
Epoch: 29/50    iter: 0/81      total_iters: 2269       d_loss:0.3398   g_loss:4.913
3
Epoch: 29/50    iter: 50/81     total_iters: 2319       d_loss:0.0913   g_loss:2.670
1
Epoch: 30/50    iter: 0/81      total_iters: 2350       d_loss:0.1221   g_loss:3.671
```

```
5
Epoch: 30/50    iter: 50/81     total_iters: 2400       d_loss:0.5588   g_loss:1.085
8
Epoch: 31/50    iter: 0/81      total_iters: 2431       d_loss:0.0743   g_loss:3.545
7
Epoch: 31/50    iter: 50/81     total_iters: 2481       d_loss:0.2085   g_loss:2.150
5
Epoch: 32/50    iter: 0/81      total_iters: 2512       d_loss:0.0449   g_loss:3.313
3
Epoch: 32/50    iter: 50/81     total_iters: 2562       d_loss:0.0137   g_loss:4.163
3
Epoch: 33/50    iter: 0/81      total_iters: 2593       d_loss:0.0202   g_loss:3.816
6
Epoch: 33/50    iter: 50/81     total_iters: 2643       d_loss:0.0911   g_loss:2.518
5
Epoch: 34/50    iter: 0/81      total_iters: 2674       d_loss:0.1028   g_loss:4.067
1
Epoch: 34/50    iter: 50/81     total_iters: 2724       d_loss:0.0849   g_loss:1.900
4
Epoch: 35/50    iter: 0/81      total_iters: 2755       d_loss:0.017    g_loss:4.105
7
Epoch: 35/50    iter: 50/81     total_iters: 2805       d_loss:0.0883   g_loss:3.652
5
Epoch: 36/50    iter: 0/81      total_iters: 2836       d_loss:0.0299   g_loss:3.752
9
Epoch: 36/50    iter: 50/81     total_iters: 2886       d_loss:0.0313   g_loss:3.850
8
Epoch: 37/50    iter: 0/81      total_iters: 2917       d_loss:0.0158   g_loss:4.780
8
Epoch: 37/50    iter: 50/81     total_iters: 2967       d_loss:0.4114   g_loss:1.152
2
Epoch: 38/50    iter: 0/81      total_iters: 2998       d_loss:0.0216   g_loss:3.934
7
Epoch: 38/50    iter: 50/81     total_iters: 3048       d_loss:0.0227   g_loss:4.060
5
Epoch: 39/50    iter: 0/81      total_iters: 3079       d_loss:0.0398   g_loss:9.502
9
Epoch: 39/50    iter: 50/81     total_iters: 3129       d_loss:0.0176   g_loss:4.041
2
Epoch: 40/50    iter: 0/81      total_iters: 3160       d_loss:0.0242   g_loss:4.210
9
Epoch: 40/50    iter: 50/81     total_iters: 3210       d_loss:0.0615   g_loss:3.557
Epoch: 41/50    iter: 0/81      total_iters: 3241       d_loss:0.0145   g_loss:4.164
1
Epoch: 41/50    iter: 50/81     total_iters: 3291       d_loss:0.0143   g_loss:4.714
Epoch: 42/50    iter: 0/81      total_iters: 3322       d_loss:0.0968   g_loss:2.226
5
Epoch: 42/50    iter: 50/81     total_iters: 3372       d_loss:0.0666   g_loss:3.282
Epoch: 43/50    iter: 0/81      total_iters: 3403       d_loss:0.0374   g_loss:3.856
Epoch: 43/50    iter: 50/81     total_iters: 3453       d_loss:0.2506   g_loss:3.556
4
Epoch: 44/50    iter: 0/81      total_iters: 3484       d_loss:0.0341   g_loss:3.741
6
Epoch: 44/50    iter: 50/81     total_iters: 3534       d_loss:0.0176   g_loss:4.700
5
Epoch: 45/50    iter: 0/81      total_iters: 3565       d_loss:0.0235   g_loss:4.055
```

```
5
Epoch: 45/50    iter: 50/81     total_iters: 3615       d_loss:0.0205   g_loss:4.450
8
Epoch: 46/50    iter: 0/81      total_iters: 3646       d_loss:0.0213   g_loss:3.988
1
Epoch: 46/50    iter: 50/81     total_iters: 3696       d_loss:0.0073   g_loss:4.779
4
Epoch: 47/50    iter: 0/81      total_iters: 3727       d_loss:0.0581   g_loss:3.359
8
Epoch: 47/50    iter: 50/81     total_iters: 3777       d_loss:0.009    g_loss:4.761
2
Epoch: 48/50    iter: 0/81      total_iters: 3808       d_loss:0.0043   g_loss:5.232
1
Epoch: 48/50    iter: 50/81     total_iters: 3858       d_loss:0.0206   g_loss:4.307
3
Epoch: 49/50    iter: 0/81      total_iters: 3889       d_loss:0.0081   g_loss:4.706
6
Epoch: 49/50    iter: 50/81     total_iters: 3939       d_loss:0.0434   g_loss:3.995
6
Epoch: 50/50    iter: 0/81      total_iters: 3970       d_loss:0.0788   g_loss:3.991
8
Epoch: 50/50    iter: 50/81     total_iters: 4020       d_loss:0.0093   g_loss:4.547
1
```

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()
```



In [ ]:
```
fig = plt.figure(figsize=(14,14))
plt.axis("off")
```

```
ims = [[plt.imshow(np.transpose(i.cpu(),(1,2,0)), animated=True)] for i in img_list
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=Tr

HTML(ani.to_jshtml())
```

Out[ ]: