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

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

```
import matplotlib.pyplot as plt
import numpy as np
import torch

import torch.nn as nn

from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
from torch.nn import Module, Sequential, Conv2d, ConvTranspose2d, LeakyReLU, BatchNorm2d, ReLU, Tanh, Sigmoid, BCELoss

%matplotlib inline
```

```
# Imports PIL module
from PIL import Image
import os
```

```
!unzip lp.zip
```

```
# path to the image directory
dir_data  = '/content/A lichen planus'

# setting image shape to 32x32
img_shape = (32,32, 3)

# listing out all file names
nm_imgs   = np.sort(os.listdir(dir_data))
```

```
X_train = []
for file in nm_imgs:
        img = Image.open(dir_data+'/'+file)
        img = img.convert('RGB')
        img = img.resize((32,32))
        img = np.asarray(img)/255
        X_train.append(img)

X_train = np.array(X_train)
```

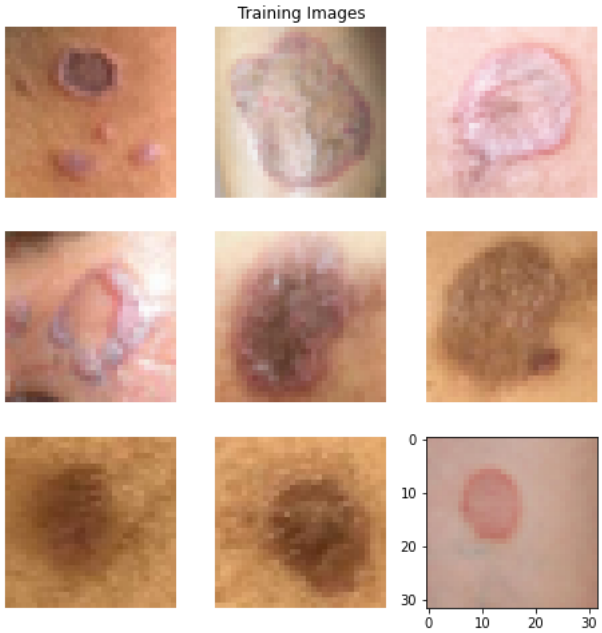
```
from numpy import asarray
from numpy import savez_compressed

# save to npy file
savez_compressed('kaggle_images_32x32.npz', X_train)
```

```
#load dict of arrays
dict_data = np.load('kaggle_images_32x32.npz')

# extract the first array
data = dict_data['arr_0']

# print the array
print(data)
```

```
def tf_dataset(images_path, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices(images_path)
    dataset = dataset.shuffle(buffer_size=10240)
    dataset = dataset.map(load_image, num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

    return dataset
```

```
# Always good to check if gpu support available or not
dev = 'cuda:0' if torch.cuda.is_available() == True else 'cpu'
device = torch.device(dev)
```

```
# plot images in a nxn grid

def plot_images(imgs, grid_size = 5):
    """
    imgs: vector containing all the numpy images
    grid_size: 2x2 or 5x5 grid containing images
    """

    fig = plt.figure(figsize = (8, 8))
    columns = rows = grid_size
    plt.title("Training Images")

    for i in range(1, columns*rows +1):
        plt.axis("off")
        fig.add_subplot(rows, columns, i)
        plt.imshow(imgs[i])
    plt.show()


# load the numpy vector containing image representations
imgs = np.load('kaggle_images_32x32.npz')

# pls ignore the poor quality of the images as we are working with 32x32 sized images.
plot_images(imgs['arr_0'], 3)
```




```
class LP(Dataset):

    def __init__(self, npz_imgs):
        """
        Args:
            npz_imgs (string): npz file with all the images (created in gan.ipynb)
        """
        self.imgs = npz_imgs

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

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        image = self.imgs[idx]

        return image


# Preparing dataloader for training

transpose_imgs = np.transpose( # imp step to convert image size from (7312, 32,32,3) to (7312, 3,32,32)
    np.float32(imgs['arr_0']), # imp step to convert double -&gt; float (by default numpy input uses double as data type)
    (0, 3,1,2) # tuple to describe how to rearrange the dimensions
    )

dset = LP(transpose_imgs) # passing the npz variable to the constructor class
batch_size = 32
shuffle = True
```

```
dataloader = DataLoader(dataset = dset, batch_size = batch_size, shuffle = shuffle)

class Generator(Module):
    def __init__(self):

        # calling constructor of parent class
        super().__init__()

        self.gen = Sequential(
          ConvTranspose2d(in_channels = 100, out_channels = 512 , kernel_size = 4, stride = 1, padding = 0, bias = False),
          # the output from the above will be b_size ,512, 4,4
          BatchNorm2d(num_features = 512), # From an input of size (b_size, C, H, W), pick num_features = C
          ReLU(inplace = True),

          ConvTranspose2d(in_channels = 512, out_channels = 256 , kernel_size = 4, stride = 2, padding = 1, bias = False),
          # the output from the above will be b_size ,256, 8,8
          BatchNorm2d(num_features = 256),
          ReLU(inplace = True),

          ConvTranspose2d(in_channels = 256, out_channels = 128 , kernel_size = 4, stride = 2, padding = 1, bias = False),
          # the output from the above will be b_size ,128, 16,16
          BatchNorm2d(num_features = 128),
          ReLU(inplace = True),

          ConvTranspose2d(in_channels = 128, out_channels = 3 , kernel_size = 4, stride = 2, padding = 1, bias = False),
          # the output from the above will be b_size ,3, 32,32
          Tanh()

        )

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


# Defining the Discriminator class

class Discriminator(Module):
    def __init__(self):

        super().__init__()
        self.dis = Sequential(

            # input is (3, 32, 32)
            Conv2d(in_channels = 3, out_channels = 32, kernel_size = 4, stride = 2, padding = 1, bias=False),
            # ouput from above layer is b_size, 32, 16, 16
            LeakyReLU(0.2, inplace=True),

            Conv2d(in_channels = 32, out_channels = 32*2, kernel_size = 4, stride = 2, padding = 1, bias=False),
            # ouput from above layer is b_size, 32*2, 8, 8
            BatchNorm2d(32 * 2),
            LeakyReLU(0.2, inplace=True),

            Conv2d(in_channels = 32*2, out_channels = 32*4, kernel_size = 4, stride = 2, padding = 1, bias=False),
            # ouput from above layer is b_size, 32*4, 4, 4
            BatchNorm2d(32 * 4),
            LeakyReLU(0.2, inplace=True),

            Conv2d(in_channels = 32*4, out_channels = 32*8, kernel_size = 4, stride = 2, padding = 1, bias=False),
            # ouput from above layer is b_size, 256, 2, 2
            # NOTE: spatial size of this layer is 2x2, hence in the final layer, the kernel size must be 2 instead (or smaller
            BatchNorm2d(32 * 8),
            LeakyReLU(0.2, inplace=True),

            Conv2d(in_channels = 32*8, out_channels = 1, kernel_size = 2, stride = 2, padding = 0, bias=False),
            # ouput from above layer is b_size, 1, 1, 1
            Sigmoid()
        )

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


netG = Generator().to(device)
netD = Discriminator().to(device)
```

```
def init_weights(m):
    if type(m) == ConvTranspose2d:
        nn.init.normal_(m.weight, 0.0, 0.02)
    elif type(m) == BatchNorm2d:
        nn.init.normal_(m.weight, 1.0, 0.02)
        nn.init.constant_(m.bias, 0)


# initializing the weights
netD.apply(init_weights)
netG.apply(init_weights)
```

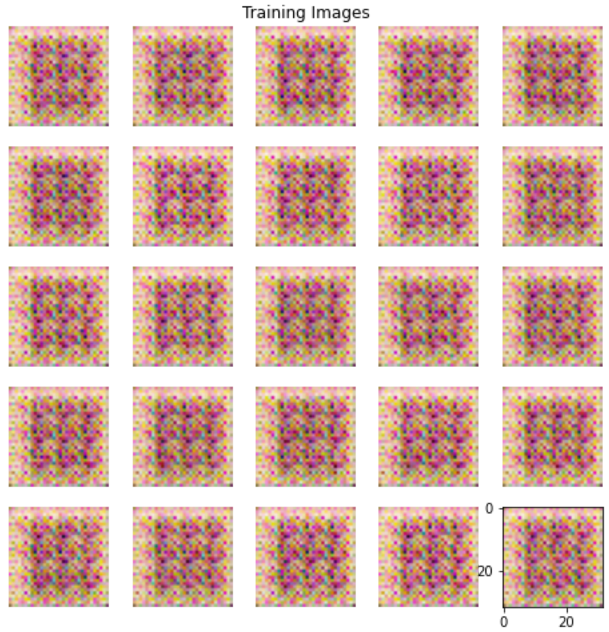
```
Generator(
  (gen): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 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, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): Tanh()
  )
)
```

```
# Setting up otimizers for both Generator and Discriminator

opt_D = optim.Adam(netD.parameters(), lr = 0.0002, betas= (0.5, 0.999))
opt_G = optim.Adam(netG.parameters(), lr = 0.0002, betas= (0.5, 0.999))


# Setting up the loss function - BCELoss (to check how far the predicted value is from real value)

loss = BCELoss()
```

```
# TRAINING GANS
epochs = 100

# going over the entire dataset 10 times
for e in range(epochs):

    # pick each batch b of input images: shape of each batch is (32, 3, 32, 32)
    for i, b in enumerate(dataloader):

        ############################
        ## Update Discriminator ##
        ############################

        # Loss on real images

        # clear the gradient
        opt_D.zero_grad() # set the gradients to 0 at start of each loop because gradients are accumulated on subsequent backw
        # compute the D model output
        yhat = netD(b.to(device)).view(-1) # view(-1) reshapes a 4-d tensor of shape (2,1,1,1) to 1-d tensor with 2 values onl
        # specify target labels or true labels
        target = torch.ones(len(b), dtype=torch.float, device=device)
        # calculate loss
        loss_real = loss(yhat, target)
        # calculate gradients -  or rather accumulation of gradients on loss tensor
        loss_real.backward()

        # Loss on fake images

        # generate batch of fake images using G
        # Step1: creating noise to be fed as input to G
        noise = torch.randn(len(b), 100, 1, 1, device = device)
        # Step 2: feed noise to G to create a fake img (this will be reused when updating G)
        fake_img = netG(noise)

        # compute D model output on fake images
        yhat = netD(fake_img.detach()).view(-1) # .cuda() is essential because our input i.e. fake_img is on gpu but model isn
        # specify target labels
        target = torch.zeros(len(b), dtype=torch.float, device=device)
        # calculate loss
        loss_fake = loss(yhat, target)
        # calculate gradients
        loss_fake.backward()

        # total error on D
        loss_disc = loss_real + loss_fake
```

```
# Update weights of D
opt_D.step()

#########################
#### Update Generator ####
#########################

# clear gradient
opt_G.zero_grad()
# pass fake image through D
yhat = netD(fake_img).view(-1)
# specify target variables - remember G wants D *to think* these are real images so label is 1
target = torch.ones(len(b), dtype=torch.float, device=device)
# calculate loss
loss_gen = loss(yhat, target)
# calculate gradients
loss_gen.backward()
# update weights on G
opt_G.step()


###################################
#### Plot some Generator images ####
###################################

# during every epoch, print images at every 10th iteration.
if i% 10 == 0:
    # convert the fake images from (b_size, 3, 32, 32) to (b_size, 32, 32, 3) for plotting
    img_plot = np.transpose(fake_img.detach().cpu(), (0,2,3,1)) # .detach().cpu() is imp for copying fake_img tensor t
    plot_images(img_plot)
    print("********************")
    print(" Epoch %d and iteration %d " % (e, i))
```

```
********************
 Epoch 60 and iteration 0
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```