Generative Adversarial Networks

For this part of the assignment you implement two different types of generative adversarial networks. We will train the networks on the Celeb A dataset which is a large set of celebrity face images.

```
from google.colab import drive
drive.mount('/content/gdrive')
import os
os.chdir("gdrive/My Drive/assignment4")
```

```
    Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
```

```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder

import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2
```

```
# from gan.train import train
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

GAN loss functions

In this assignment you will implement two different types of GAN cost functions. You will first implement the loss from the original GAN paper. You will also implement the loss from LS-GAN.

GAN loss

TODO: Implement the `discriminator_loss` and `generator_loss` functions in `gan/losses.py`.

The generator loss is given by:

$$\ell_G = -\mathbb{E}_{z \sim p(z)} \left[\log D(G(z))\right]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x \sim p_\text{data}} \left[\log D(x)\right] - \mathbb{E}_{z \sim p(z)} \left[\log(1 - D(G(z)))\right]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

HINTS: You should use the `torch.nn.functional.binary_cross_entropy_with_logits` function to compute the binary cross entropy loss since it is more numerically stable than using a softmax followed by BCE loss. The BCE loss is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log(1 - D(G(z)))$, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

```
from gan.losses import discriminator_loss, generator_loss
```

Least Squares GAN loss

TODO: Implement the `ls_discriminator_loss` and `ls_generator_loss` functions in `gan/losses.py`.

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z \sim p(z)} \left[(D(G(z)) - 1)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x \sim p_\text{data}} \left[(D(x) - 1)^2\right] + \frac{1}{2}\mathbb{E}_{z \sim p(z)} \left[(D(G(z)))^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`).

```
# from gan.losses import ls_discriminator_loss, ls_generator_loss
```

GAN model architecture

TODO: Implement the `Discriminator` and `Generator` networks in `gan/models.py`.

We recommend the following architectures which are inspired by DCGAN:

Discriminator:

- convolutional layer with in_channels=3, out_channels=128, kernel=4, stride=2
- convolutional layer with in_channels=128, out_channels=256, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=256, out_channels=512, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=512, out_channels=1024, kernel=4, stride=2
- batch norm
- convolutional layer with in_channels=1024, out_channels=1, kernel=4, stride=1

Use padding = 1 (not 0) for all the convolutional layers.

Instead of Relu we LeakyReLu throughout the discriminator (we use a negative slope value of 0.2). You can use simply use relu as well.

The output of your discriminator should be a single value score corresponding to each input sample. See `torch.nn.LeakyReLU`.

Generator:

Note: In the generator, you will need to use transposed convolution (sometimes known as fractionally-strided convolution or deconvolution). This function is implemented in pytorch as `torch.nn.ConvTranspose2d`.

- transpose convolution with in_channels=NOISE_DIM, out_channels=1024, kernel=4, stride=1
- batch norm
- transpose convolution with in_channels=1024, out_channels=512, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=512, out_channels=256, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=256, out_channels=128, kernel=4, stride=2
- batch norm
- transpose convolution with in_channels=128, out_channels=3, kernel=4, stride=2

The output of the final layer of the generator network should have a `tanh` nonlinearity to output values between -1 and 1. The output should be a 3x64x64 tensor for each sample (equal dimensions to the images from the dataset).

```
# from gan.models import Discriminator, Generator

import torch
import torch.nn as nn
from gan.spectral_normalization import SpectralNorm
```

```python
class Discriminator(nn.Module):
    def __init__(self, input_channels=3):
        super(Discriminator, self).__init__()


        self.conv1 = SpectralNorm(nn.Conv2d(3, 128, 4, stride=2, padding=1))
        self.conv2 = SpectralNorm(nn.Conv2d(128, 256, 4, stride=2, padding=1))
        self.bn1 = nn.BatchNorm2d(256)
        self.conv3 = SpectralNorm(nn.Conv2d(256, 512, 4, stride=2, padding=1))
        self.bn2 = nn.BatchNorm2d(512)
        self.conv4 = SpectralNorm(nn.Conv2d(512, 1024, 4, stride=2, padding=1))
        self.bn3 = nn.BatchNorm2d(1024)
        self.conv5 = SpectralNorm(nn.Conv2d(1024, 1, 4, stride=1, padding=0))
        self.leakyrelu = nn.LeakyReLU(0.2)

        # self.conv1 = nn.Conv2d(3, 128, 4, stride=2, padding=1)
        # self.conv2 = nn.Conv2d(128, 256, 4, stride=2, padding=1)
        # self.bn1 = nn.BatchNorm2d(256)
        # self.conv3 = nn.Conv2d(256, 512, 4, stride=2, padding=1)
        # self.bn2 = nn.BatchNorm2d(512)
        # self.conv4 = nn.Conv2d(512, 1024, 4, stride=2, padding=1)
        # self.bn3 = nn.BatchNorm2d(1024)
        # self.conv5 = nn.Conv2d(1024, 1, 4, stride=1, padding=0)
        # self.leakyrelu = nn.LeakyReLU(0.2)


    def forward(self, x):
        x = self.leakyrelu(self.conv1(x))
        x = self.leakyrelu(self.conv2(x))
        x = self.bn1(x)
        x = self.leakyrelu(self.conv3(x))
        x = self.bn2(x)
        x = self.leakyrelu(self.conv4(x))
        x = self.bn3(x)
        x = self.conv5(x)
        # x = self.leakyrelu(x)
        batch_size = x.shape[0]
        x = x.view(batch_size,1)
        return x


class Generator(nn.Module):
    def __init__(self, noise_dim, output_channels=3):
        super(Generator, self).__init__()
        self.noise_dim = noise_dim
        self.model = nn.Sequential(
            nn.ConvTranspose2d(self.noise_dim, 1024, 4, stride = 1),
            nn.BatchNorm2d(1024),
            nn.ReLU(),
            nn.ConvTranspose2d(1024, 512, 4, stride = 2, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.ConvTranspose2d(512, 256, 4, stride = 2, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
```

```
            nn.ConvTranspose2d(256, 128, 4, stride = 2, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 3, 4, stride = 2, padding=1),
            nn.Tanh()
        )


    def forward(self, x):
        x = self.model(x)
        return x
```

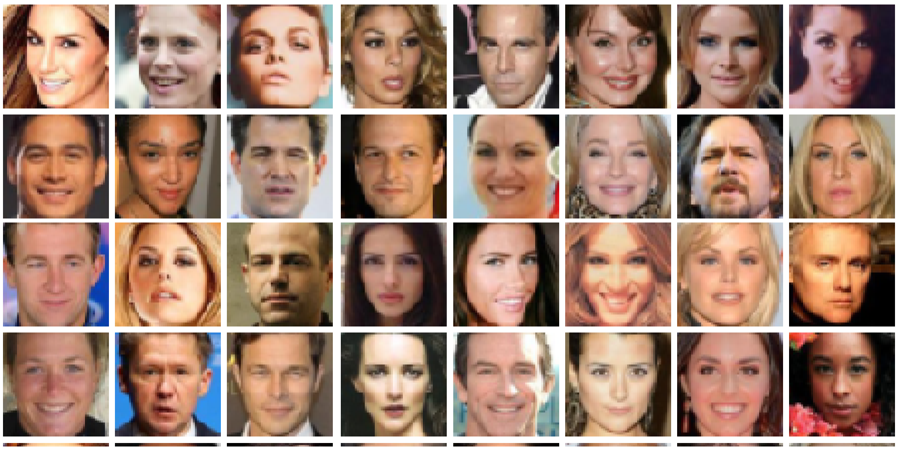
Data loading: Celeb A Dataset

The CelebA images we provide have been filtered to obtain only images with clear faces and have been cropped and downsampled to 128x128 resolution.

```
batch_size = 64
scale_size = 64  # We resize the images to 64x64 for training

celeba_root = 'celeba_data'
# celeba_root = 'celeba_train_128res'


celeba_train = ImageFolder(root=celeba_root, transform=transforms.Compose([
  transforms.Resize(scale_size),
  transforms.ToTensor()
]))

celeba_loader_train = DataLoader(celeba_train, batch_size=batch_size, drop_last=Tru
```

Visualize dataset

```
from gan.utils import sample_noise, show_images, deprocess_img, preprocess_img
imgs = celeba_loader_train.__iter__().next()[0].numpy().squeeze()
show_images(imgs, color=True)
```



Training

TODO: Fill in the training loop in `gan/train.py`.

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from gan.utils import sample_noise, show_images, deprocess_img, preprocess_img
import time
import math
from tqdm import tqdm

def time_since(since):
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def train(D, G, D_solver, G_solver, discriminator_loss, generator_loss, show_every=
              batch_size=128, noise_size=100, num_epochs=10, train_loader=None, dev

    iter_count = 0
    start = time.time()
    for epoch in tqdm(range(num_epochs)):
        print('EPOCH: ', (epoch+1))
        for x, _ in train_loader:
            _, input_channels, img_size, _ = x.shape
            real_images = preprocess_img(x).to(device)  # normalize

            d_error = None
            g_error = None
            fake_images = None

            # Discriminator step
            D_solver.zero_grad()
            fake_input = sample_noise(batch_size, noise_size).view(batch_size, nois
            fake_images = G(fake_input).detach()
            fake_images = fake_images.view(batch_size, input_channels, img_size, im
            logits_fake = D(fake_images)
            logits_real = D(real_images)
```

```
            d_error = discriminator_loss(logits_real, logits_fake)
            d_error.backward()
            D_solver.step()

            # Generator Step
            G_solver.zero_grad()
            fake_input = sample_noise(batch_size, noise_size).view(batch_size, nois
            fake_images = G(fake_input)
            fake_images = fake_images.view(batch_size, input_channels, img_size, im
            logits_fake = D(fake_images)
            g_error = generator_loss(logits_fake)
            g_error.backward()
            G_solver.step()

            # Logging and output visualization
            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}, Time: {}'.format(iter_count,d_e
                disp_fake_images = deprocess_img(fake_images.data)  # denormalize
                imgs_numpy = (disp_fake_images).cpu().numpy()
                show_images(imgs_numpy[0:16], color=input_channels!=1)
                plt.show()
                print()
            iter_count += 1
```

```
NOISE_DIM = 100
NUM_EPOCHS = 10
learning_rate = 0.0002
```

Train GAN

```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.99
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.99
```

```
# original gan
train(D, G, D_optimizer, G_optimizer, discriminator_loss,
          generator_loss, num_epochs=NUM_EPOCHS, show_every=150,
          train_loader=celeba_loader_train, device=device)
```

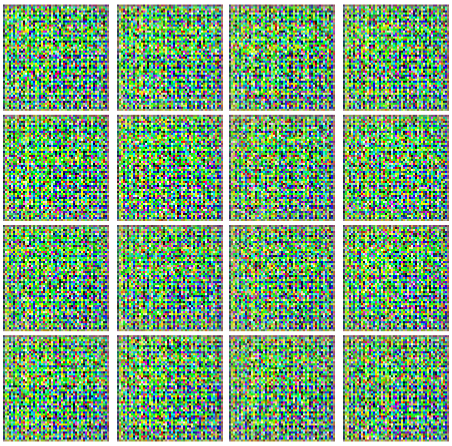
+ 代码 + 文本

Train LS-GAN

```
D = Discriminator().to(device)
G = Generator(noise_dim=NOISE_DIM).to(device)
```

```
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5, 0.99
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5, 0.99


# ls-gan
train(D, G, D_optimizer, G_optimizer, ls_discriminator_loss,
          ls_generator_loss, num_epochs=NUM_EPOCHS, show_every=200,
          train_loader=celeba_loader_train, device=device)
```

```
  0%|          | 0/20 [00:00<?, ?it/s]EPOCH:  1
Iter: 0, D: 1.338, G:11.32, Time: 0m 0s
```



```
Iter: 200, D: 1.128, G:1.908, Time: 1m 7s
```

```
Iter: 400, D: 0.8737, G:3.609, Time: 2m 14s
```

```
Iter: 600, D: 1.277, G:1.045, Time: 10m 45s
```