

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.

```
In [ ]: import torch
        from torch.utils.data import DataLoader
        from torchvision import transforms
        from torchvision.datasets import ImageFolder

        import numpy as np

        %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
```

```
In [ ]: from gan.train import train
```

```
In [ ]: 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](https://arxiv.org/pdf/1406.2661.pdf) (<https://arxiv.org/pdf/1406.2661.pdf>). You will also implement the loss from [LS-GAN](https://arxiv.org/abs/1611.04076) (<https://arxiv.org/abs/1611.04076>).

GAN loss

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

The generator loss is given by:

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

and the discriminator loss is:

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

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.

```
In [ ]: 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 \(https://arxiv.org/abs/1611.04076\)](https://arxiv.org/abs/1611.04076), a newer, more stable alternative 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:

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

and the discriminator loss:

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

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`).

```
In [ ]: 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](https://arxiv.org/pdf/1511.06434.pdf) (<https://arxiv.org/pdf/1511.06434.pdf>):

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`

Instead of Relu we LeakyRelu throughout the discriminator (we use a negative slope value of 0.2).

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).

```
In [ ]: from gan.models import Discriminator, Generator
```

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.

```
In [ ]: batch_size = 128
        scale_size = 64 # We resize the images to 64x64 for training

        celeba_root = 'celeba_data'

In [ ]: 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=True)
```

Visualize dataset

```
In [ ]: 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` .

```
In [ ]: NOISE_DIM = 100
        NUM_EPOCHS = 50
        learning_rate = 0.0002
```

Train GAN

```
In [ ]: D = Discriminator().to(device)
        G = Generator(noise_dim=NOISE_DIM).to(device)

In [ ]: D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5,
        0.999))
        G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5,
        0.999))

In [ ]: # original gan
        train(D, G, D_optimizer, G_optimizer, discriminator_loss,
            generator_loss, num_epochs=NUM_EPOCHS, show_every=250,
            train_loader=celeba_loader_train, device=device)
```

Train LS-GAN

```
In [ ]: D = Discriminator().to(device)
        G = Generator(noise_dim=NOISE_DIM).to(device)
```

```
In [ ]: D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate, betas = (0.5,
0.999))
        G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate, betas = (0.5,
0.999))
```

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

```
In [ ]:
```