Prepare your working environment

If you have installed following libraries in your python environment, you can skip this step.

```
pytorch
torchvision
numpy
matplotlib
```

Otherwise, you can create a new Anaconda environment:

```
conda create -n dcgan python=3.6.5
conda activate dcgan
```

And follow the instruction of installing pytorch according to your device.

Then install other dependencies:

```
conda install numpy
conda install matplotlib
```

Prepare your data

In this tutorial we will use the Celeb-A Faces dataset which can be downloaded here. The dataset will download as a file named `img_align_celeba.zip`. Once downloaded, create a directory named `data/celeba` and extract the zip file into that directory. The resulting directory structure should be:

```
fake_imgs/
data/celeba
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
            ...
```

Note that you also need to create an empty folder named `fake_imgs` under the same directory as the `data` folder.

Introduction

This tutorial will guide you through the implementation of DCGAN, a popular generative adversarial network (GAN) for image generation. We will train the model to generate new celebrities after showing it pictures of many real celebrities.

What is a GAN?

GANs are a framework for teaching a DL model to capture the training data's distribution so we can generate new data from that same distribution. GANs were invented by Ian Goodfellow in 2014 and first described in the paper Generative Adversarial Nets.

Let x be data representing an image. $D(x)$ is the discriminator network which outputs the (scalar) probability that x came from training data rather than the generator. Here, since we are dealing with images, the input to $D(x)$ is an image of (C, H, W) size 3x64x64. Intuitively, $D(x)$ should be high when x comes from training data and low when x comes from the generator. $D(x)$ can also be thought of as a traditional binary classifier.

For the generator's notation, let z be a latent space vector sampled from a standard normal distribution. $G(z)$ represents the generator function which maps the latent vector z to data-space. The goal of G is to estimate the distribution that the training data comes from (p_{data}) so it can generate fake samples from that estimated distribution (p_g). So, $D(G(z))$ is the probability (scalar) that the output of the generator G is a real image.

D and G play a minimax game in which D tries to maximize the probability it correctly classifies reals and fakes ($logD(x)$), and G tries to minimize the probability that D will predict its outputs are fake ($log(1 - D(G(z)))$). The GAN loss function is

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

In theory, the solution to this minimax game is where $p_g = p_{data}$, and the discriminator guesses randomly if the inputs are real or fake. However, the convergence theory of GANs is still being actively researched and in reality models do not always train to this point.

What is a DCGAN?

A DCGAN is a direct extension of the GAN described above, except that it uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et. al. in the paper Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks.

The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution.

The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, z, that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image.

Coding Time

You can paste the following code blocks into your own python scirpt to run the experiments.

Import dependencies

```
from __future__ import print_function
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
```

Set random seed for reproducibility

```
manualSeed = 999
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Parameters

- **dataroot** - the path to the root of the dataset folder.
- **workers** - the number of worker threads for loading the data with the DataLoader.
- **batch_size** - the batch size used in training.
- **image_size** - the spatial size of the images used for training. This implementation defaults to 64x64. If another size is desired, the structures of D and G must be changed. See here for more details.
- **nc** - number of color channels in the input images. For color images this is 3.
- **nz** - length of latent vector.
- **ngf** - relates to the depth of feature maps carried through the generator.
- **ndf** - sets the depth of feature maps propagated through the discriminator.
- **num_epochs** - number of training epochs to run. Training for longer will probably lead to better results but will also take more time.
- **lr** - learning rate for training.
- **beta1** - beta1 hyperparameter for Adam optimizers.
- **ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs.

```
# Root directory for dataset
dataroot = "data/celeba"

# Number of workers for dataloader
workers = 2

# 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 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 = 1

# 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 the PyTorch 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 PyTorch dataloader

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)
```

Set devices

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

Weight initialization

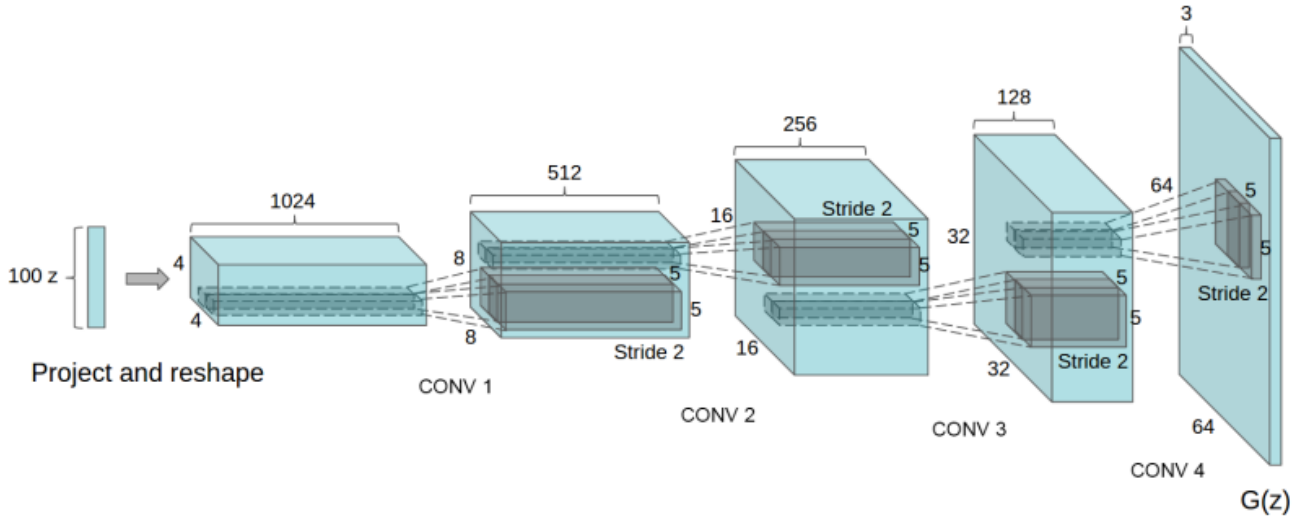
From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with mean=0, stdev=0.02. The `weights_init` function takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers to meet this criteria. This function is applied to the models immediately after initialization.

```
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)
```

Define Generator

The generator, G is designed to map the latent space vector (z) to data-space. Since our data are images, converting z to data-space means ultimately creating a RGB image with the same size as the training images (i.e., 3x64x64). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1,1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as these layers help with the flow of gradients

during training.




```
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)
```

```
# 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)
```

Define Discriminator

The discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D.

```
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)
```

```
# 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
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

Loss Functions and Optimizers

We will use the Binary Cross Entropy loss (BCELoss) function which is defined in PyTorch as:

$$\ell(x, y) = L = l_1, \dots, l_N{}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

```
# 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))
```

Training

Train the Discriminator

First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss $log(D(x))$, then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss $log(1 - D(G(z)))$, and **accumulate** the gradients with a backward pass.

Train the Generator

As stated in the original paper, we want to train the Generator by minimizing $log(1 - D(G(z)))$. However, this was shown to not provide sufficient gradients, especially early in the learning process. As a fix, we

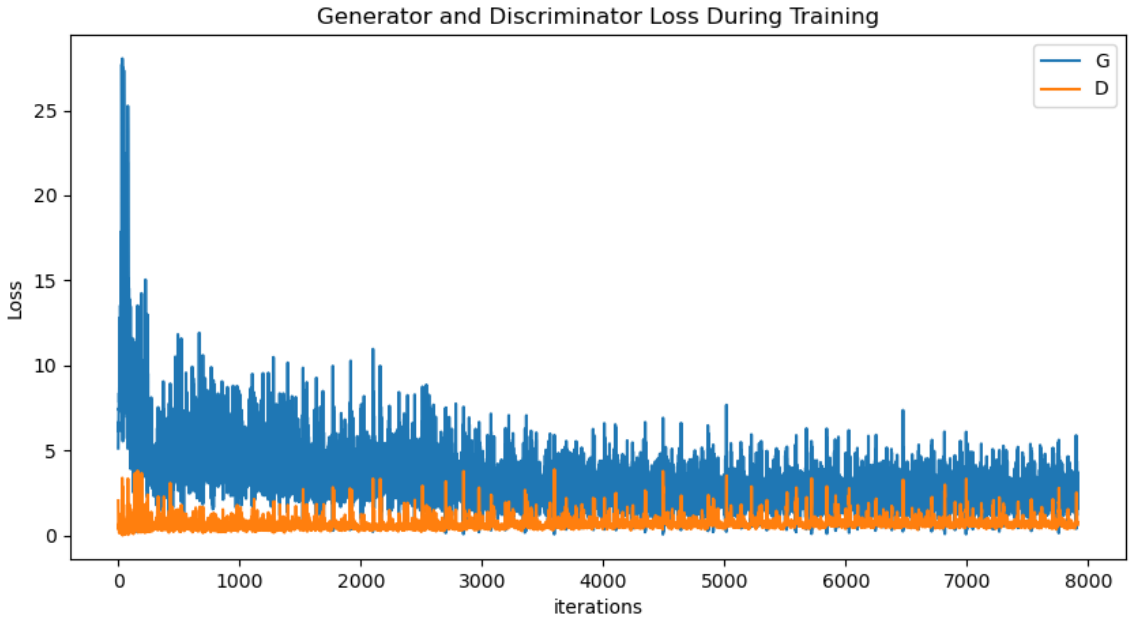
instead wish to maximize $log(D(G(z)))$.

```
# Training Loop

# 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.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
previous gradients
        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()
```

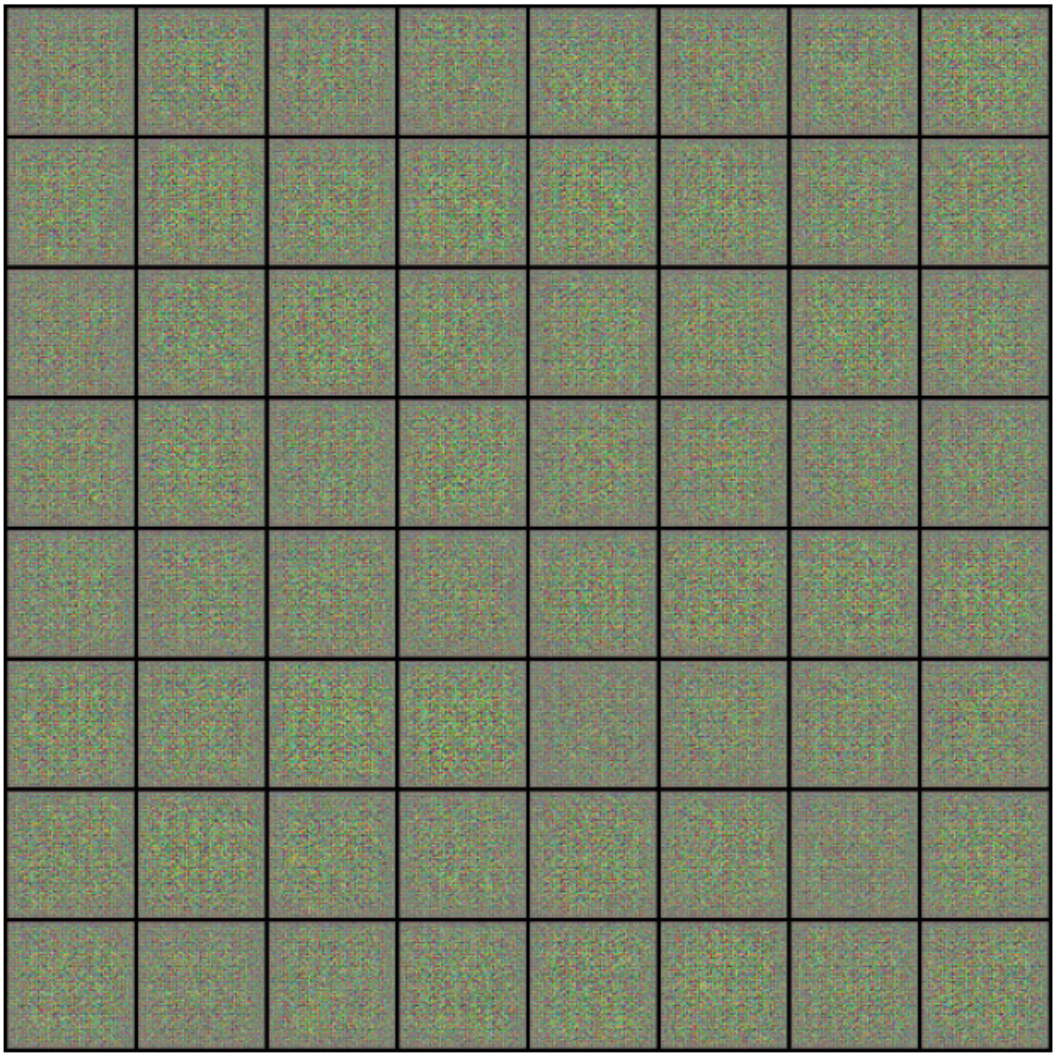
```
        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake
batch 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()

        # Output training stats
        if i % 50 == 0:
            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_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(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1
```

Visualize results

```
# Loss versus training iteration

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.savefig('loss.png')
```




```
# Visualization of G’s progression (generate a GIF image, best view in
Markdown format)

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_delay=1000,
blit=True)

ani.save('gan_progression.gif', writer='pillow')
```



```
# Real Images vs. Fake Images

# 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(device)[: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.savefig('real_vs_fake.png')
```


```
# Save images for evaluation
fixed_noise_for_IS = torch.randn(1000, nz, 1, 1, device=device)

with torch.no_grad():
    fake = netG(fixed_noise_for_IS).detach().cpu()

for x in range(fake.shape[0]):
    name = str(x).zfill(4) + '.png'
    img = np.transpose(fake[x, :, :, :], (1, 2, 0)).numpy()
    img = (img + 1.0) / 2.0 * 255.0
    plt.imsave(os.path.join('fake_imgs', name), img.astype(np.uint8))
```

Evaluation

We will use Inception Score (IS) to evaluate the performance of our GAN. Higher IS indicates better image generation quality.

Under the dcgan environment, type the following commands in your command line:

```
pip install torch-fidelity
```

```
fidelity --gpu 0 --isc --input1 fake_imgs
```

Note that this command requires gpu(s).

Exercises

1. Change the random seed to generate different visualizations.
2. Modify the number of epochs and see how results will change.
3. Change learning rate and batch size to see how they affect the training process.
4. Use the CIFAR-10 dataset to train the model.

Reference

1. Generative Adversarial Nets.
2. Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks.
3. PyTorch Tutorial on DCGAN.
4. PyTorch Example on DCGAN.