Ta le of Contents

Cola   Note ook   GitHu

DCGAN T TORIAL

Author: Nathan Inkawhich

Intro uction

This tutorial will give an intro uction to DCGANs through an example. We will train a generative a versarial network (GAN) to generate new cele rities after showing it pictures of many real cele rities. Most of the co e here is from the cgan implementation in pytorch/examples, an this ocument will give a thorough explanation of the implementation an she light on how an why this mo el works. But on't worry, no prior knowle ge of GANs is require , ut it may require a first-timer to spen some time reasoning a out what is actually happening un er the hoo . Also, for the sake of time it will help to have a GP , or two. Lets start from the eginning.

Generative A versarial Networks

What is a GAN?

GANs are a framework for teaching a DL mo el to capture the training ata's istri ution so we can generate new ata from that same istri ution. GANs were invente y Ian Goo fellow in 2014 an first escri e in the paper Generative A versarial Nets. They are ma e of two istinct mo els, a *generator* an a *discriminator*. The o of the generator is to spawn 'fake' images that look like the training images. The o of the iscriminator is to look at an image an output whether or not it is a real training image or a fake image from the generator. During training, the generator is constantly trying to outsmart the iscriminator y generating etter an etter fakes, while the iscriminator is working to ecome a etter etective an correctly classify the real an fake images. The equili rium of this game is when the generator is generating perfect fakes that look as if they came irectly from the training ata, an the iscriminator is left to always guess at 50% confi ence that the generator output is real or fake.

Now, lets efine some notation to e use throughout tutorial starting with the iscriminator. Let x e ata representing an image. $D(x)$ is the iscriminator network which outputs the (scalar) pro a ility that x came from training ata rather than the generator. Here, since we are ealing with images the input to $D(x)$ is an image of CHW size 3x64x64. Intuitively, $D(x)$ shoul e HIGH when x comes from training ata an LOW when x comes from the generator. $D(x)$ can also e thought of as a tra itional inary classifier.

For the generator's notation, let z e a latent space vector sample from a stan ar normal istri ution. $G(z)$ represents the generator function which maps the latent vector z to ata-space. The goal of G is to estimate the istri ution that the training ata comes from (p_{data}) so it can generate fake samples from that estimate istri ution (p_g).

So, $D(G(z))$ is the pro a ility (scalar) that the output of the generator G is a real image. As escri e in Goo fellow's paper, D an G play a minimax game in which D tries to maximize the pro a ility it correctly classifies reals an fakes ($logD(x)$), an G tries to minimize the pro a ility that D will pre ict its outputs are fake ($log(1-D(G(x)))$). From the paper, the GAN loss function is

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

In theory, the solution to this minimax game is where $p_g = p_{data}$, an the iscriminator guesses ran omly if the inputs are real or fake. However, the convergence theory of GANs is still eing actively researche an in reality mo els o not always train to this point.

What is a DCGAN?

A DCGAN is a irect extension of the GAN escri e a ove, except that it explicitly uses convolutional an convolutional-transpose layers in the iscriminator an generator, respectively. It was first escri e y Ra for et. al. in the paper nsupervise Representation Learning With Deep Convolutional Generative A versarial Networks. The iscriminator is ma e up of stri e convolution layers, atch norm layers, an LeakyReL activations. The input is a 3x64x64 input image an the output is a scalar pro a ility that the input is from the real ata istri ution. The generator is comprise of convolutional-transpose layers, atch norm layers, an ReL activations. The input is a latent vector, z, that is rawn from a stan ar normal istri ution an the output is a 3x64x64 RGB image. The stri e conv-transpose layers allow the latent vector to e transforme into a volume with the same shape as an image. In the paper, the authors also give some tips a out how to setup the optimizers, how to calculate the loss functions, an how to initialize the mo el weights, all of which will e explaine in the coming sections.

```
from __future__ import print_function
#%matplotlib inline
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
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Out:

```
Random Seed:  999
```

In uts

Let's efine some inputs for the run:

- ataroot - the path to the root of the ataset fol er. We will talk more a out the ataset in the next section
- **workers** - the num er of worker threa s for loa ing the ata with the DataLoa er
- atch_size - the atch size use in training. The DCGAN paper uses a atch size of 128
- **image_size** - the spatial size of the images use for training. This implementation efaults to 64x64. If another size is esire , the structures of D an G must e change . See here for more etails
- **nc** - num er of color channels in the input images. For color images this is 3
- **nz** - length of latent vector
- **ngf** - relates to the epth of feature maps carrie through the generator
- **n f** - sets the epth of feature maps propagate through the iscriminator
- **num_e ochs** - num er of training epochs to run. Training for longer will pro a ly lea to etter results ut will also take much longer
- **lr** - learning rate for training. As escri e in the DCGAN paper, this num er shoul e 0.0002
- eta1 - eta1 hyperparameter for A am optimizers. As escri e in paper, this num er shoul e 0.5
- **ng u** - num er of GP s availa le. If this is 0, co e will run in CP mo e. If this num er is greater than 0 it will run on that num er of GP s

```python
# 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 = 5

# 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
```

Data

In this tutorial we will use the Cele -A Faces ataset which can e ownloa e at the linke site, or in Google Drive. The ataset will ownloa as a file name *img_align_celeba.zip*. Once ownloa e , create a irectory name *celeba* an extract the zip file into that irectory. Then, set the *dataroot* input for this note ook to the *celeba* irectory you ust create . The resulting irectory structure shoul e:

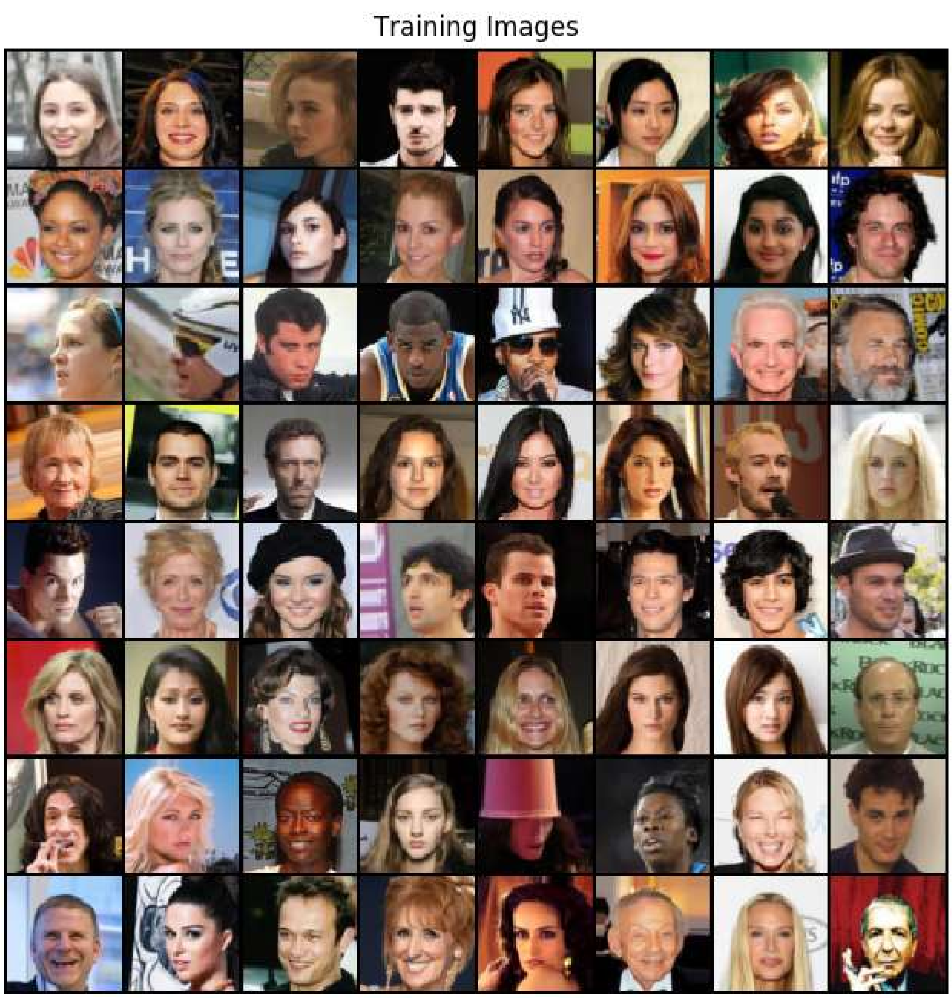
```
/path/to/celeba
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
           ...
```

This is an important step ecause we will e using the ImageFol er ataset class, which requires there to e su irectories in the ataset's root fol er. Now, we can create the ataset, create the ataloa er, set the evice to run on, an finally visualize some of the training ata.

```python
# We can use an image folder dataset the way we have it setup.
# Create the 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 dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(),(1,2,0)))
```



Im lementation

With our input parameters set an the ataset prepare , we can now get into the implementation. We will start with the weigth initialization strategy, then talk a out the generator, iscriminator, loss functions, an training loop in etail.

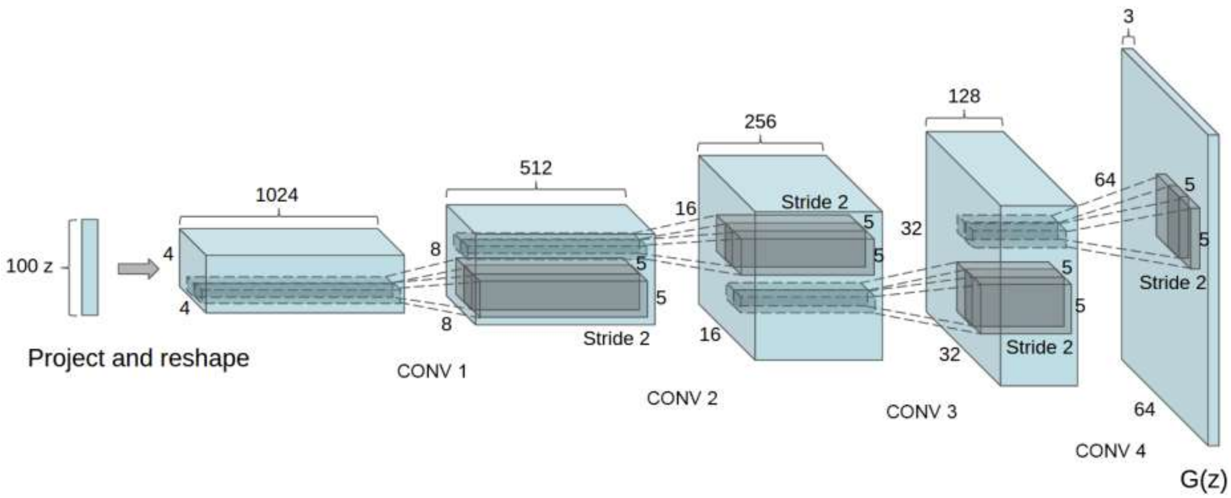
Weight Initialization

From the DCGAN paper, the authors specify that all mo el weights shall e ran omly initialize from a Normal istri ution with mean=0, st ev=0.02. The `weights_init` function takes an initialize mo el as input an reinitializes all convolutional, convolutional-transpose, an atch normalization layers to meet this criteria. This function is applie to the mo els imme iately after initialization.

```python
# 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)
```

Generator

The generator, G, is esigne to map the latent space vector (z) to ata-space. Since our ata are images, converting z to ata-space means ultimately creating a RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplishe through a series of stri e two imensional convolutional transpose layers, each paire with a 2 atch norm layer an a relu activation. The output of the generator is fe through a tanh function to return it to the input ata range of $[-1,1]$. It is worth noting the existence of the atch norm functions after the conv-transpose layers, as this is a critical contri ution of the DCGAN paper. These layers help with the flow of gra ients uring training. An image of the generator from the DCGAN paper is shown elow.



Notice, the how the inputs we set in the input section (*nz*, *ngf*, an *nc*) influence the generator architecture in co e. *nz* is the length of the z input vector, *ngf* relates to the size of the feature maps that are propagate through the generator, an *nc* is the num er of channels in the output image (set to 3 for RGB images). Below is the co e for the generator.

```python
# Generator Code

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)
```

Now, we can instantiate the generator an apply the `weights_init` function. Check out the printe mo el to see how the generator o ect is structure .

```
# 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=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

Out:

```
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

As mentione , the iscriminator, D, is a inary classification network that takes an image as input an outputs a scalar pro a ility that the input image is real (as oppose to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2 , BatchNorm2 , an LeakyReL layers, an outputs the final pro a ility through a Sigmoi activation function. This architecture can e exten e with more layers if necessary for the pro lem, ut there is significance to the use of the stri e convolution, BatchNorm, an LeakyReL s. The DCGAN paper mentions it is a goo practice to use stri e convolution rather than pooling to ownsample ecause it lets the network learn its own pooling function. Also atch norm an leaky relu functions promote healthy gra ient flow which is critical for the learning process of oth G an D.

Discriminator Co e

```
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)
```

Now, as with the generator, we can create the iscriminator, apply the `weights_init` function, an print the mo el's structure.

```
# 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)
```

Out:

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (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=False)
    (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=False)
    (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()
  )
)
```

Loss Functions an O timizers

With D an G setup, we can specify how they learn through the loss functions an optimizers. We will use the Binary Cross Entropy loss (BCELoss) function which is efine in PyTorch as:

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

Notice how this function provi es the calculation of oth log components in the o ective function (i.e. $log(D(x))$ an $log(1-D(G(z)))$). We can specify what part of the BCE equation to use with the y input. This is accomplishe in the training loop which is coming up soon, ut it is important to un erstan how we can choose which component we wish to calculate ust y changing y (i.e. GT la els).

Next, we efine our real la el as 1 an the fake la el as 0. These la els will e use when calculating the losses of D an G, an this is also the convention use in the original GAN paper. Finally, we set up two separate optimizers, one for D an one for G. As specifie in the DCGAN paper, oth are A am optimizers with learning rate 0.0002 an Beta1 = 0.5. For keeping track of the generator's learning progression, we will generate a fixe atch of latent vectors that are rawn from a Gaussian istri ution (i.e. fixe _noise) . In the training loop, we will perio ically input this fixe _noise into G, an over the iterations we will see images form out of the noise.

```
# 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

Finally, now that we have all of the parts of the GAN framework efine , we can train it. Be min ful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lea to mo e collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goo fellow's paper, while a i ing y some of the est practices shown in ganhacks. Namely, we will "construct ifferent mini- atches for real an fake" images, an also a ust G's o ective function to maximize $logD(G(z))$. Training is split up into two main parts. Part 1 up ates the Discriminator an Part 2 up ates the Generator.

Part 1 - Train the Discriminator

Recall, the goal of training the iscriminator is to maximize the pro a ility of correctly classifying a given input as real or fake. In terms of Goo fellow, we wish to "up ate the iscriminator y ascen ing its stochastic gra ient". Practically, we want to maximize $log(D(x)) + log(1-D(G(z)))$. Due to the separate mini- atch suggestion from ganhacks, we will calculate this in two steps. First, we will construct a atch of real samples from the training set, forwar pass through D, calculate the loss ($log(D(x))$), then calculate the gra ients in a ackwar pass. Secon ly, we will construct a atch of fake samples with the current generator, forwar pass this atch through D, calculate the loss ($log(1-D(G(z)))$), an *accumulate* the gra ients with a ackwar pass. Now, with the gra ients accumulate from oth the all-real an all-fake atches, we call a step of the Discriminator's optimizer.

Part - Train the Generator

As state in the original paper, we want to train the Generator y minimizing $log(1-D(G(z)))$ in an effort to generate etter fakes. As mentione , this was shown y Goo fellow to not provi e sufficient gra ients, especially early in the learning process. As a fix, we instea wish to maximize $log(D(G(z)))$. In the co e we accomplish this y: classifying the Generator output from Part 1 with the Discriminator, computing G's loss *using real labels as GT*, computing G's gra ients in a ackwar pass, an finally up ating G's parameters with an optimizer step. It may seem counter-intuitive to use the real la els as GT la els for the loss function, ut this allows us to use the $log(x)$ part of the BCELoss (rather than the $log(1-x)$ part) which is exactly what we want.

Finally, we will o some statistic reporting an at the en of each epoch we will push our fixe _noise atch through the generator to visually track the progress of G's training. The training statistics reporte are:

- **Loss_D** - iscriminator loss calculate as the sum of losses for the all real an all fake atches ($log(D(x)) + log(D(G(z)))$).
- **Loss_G** - generator loss calculate as $log(D(G(z)))$
- **D(x)** - the average output (across the atch) of the iscriminator for the all real atch. This shoul start close to 1 then theoretically converge to 0.5 when G gets etter. Think a out why this is.
- **D(G(z))** - average iscriminator outputs for the all fake atch. The first num er is efore D is up ate an the secon num er is after D is up ate . These num ers shoul start near 0 an converge to 0.5 as G gets etter. Think a out why this is.

Note: This step might take a while, epen ing on how many epochs you run an if you remove some ata from the ataset.

```
# 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
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake 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
```

Out:

```
[0/5][1100/1583]        Loss_D: 0.4006  Loss_G: 4.0563  D(x): 0.8060    D(G(z)): 0.1029 / 0.0350
[0/5][1150/1583]        Loss_D: 0.7146  Loss_G: 2.6193  D(x): 0.6174    D(G(z)): 0.0270 / 0.1379
[0/5][1200/1583]        Loss_D: 0.5904  Loss_G: 3.8754  D(x): 0.7973    D(G(z)): 0.2312 / 0.0336
[0/5][1250/1583]        Loss_D: 0.5726  Loss_G: 3.5084  D(x): 0.7041    D(G(z)): 0.0820 / 0.0518
[0/5][1300/1583]        Loss_D: 0.2397  Loss_G: 3.9215  D(x): 0.9188    D(G(z)): 0.1226 / 0.0389
[0/5][1350/1583]        Loss_D: 0.8970  Loss_G: 2.2317  D(x): 0.5205    D(G(z)): 0.0306 / 0.1601
[0/5][1400/1583]        Loss_D: 0.4736  Loss_G: 3.4108  D(x): 0.7690    D(G(z)): 0.1115 / 0.0600
[0/5][1450/1583]        Loss_D: 0.4735  Loss_G: 2.5680  D(x): 0.7829    D(G(z)): 0.1259 / 0.1264
[0/5][1500/1583]        Loss_D: 0.4586  Loss_G: 4.1247  D(x): 0.8614    D(G(z)): 0.2135 / 0.0288
[0/5][1550/1583]        Loss_D: 0.4869  Loss_G: 2.6384  D(x): 0.7691    D(G(z)): 0.1455 / 0.1007
[1/5][0/1583]   Loss_D: 1.2928  Loss_G: 6.8135  D(x): 0.9547    D(G(z)): 0.6274 / 0.0034
[1/5][50/1583]  Loss_D: 0.5992  Loss_G: 2.4385  D(x): 0.6827    D(G(z)): 0.0836 / 0.1259
[1/5][100/1583] Loss_D: 0.2856  Loss_G: 3.8549  D(x): 0.8473    D(G(z)): 0.0859 / 0.0314
[1/5][150/1583] Loss_D: 0.4447  Loss_G: 3.4027  D(x): 0.8653    D(G(z)): 0.2035 / 0.0618
[1/5][200/1583] Loss_D: 0.7865  Loss_G: 2.9328  D(x): 0.6328    D(G(z)): 0.1500 / 0.1019
[1/5][250/1583] Loss_D: 1.9063  Loss_G: 2.2006  D(x): 0.2396    D(G(z)): 0.0026 / 0.1657
[1/5][300/1583] Loss_D: 0.3532  Loss_G: 3.3867  D(x): 0.8000    D(G(z)): 0.0833 / 0.0549
[1/5][350/1583] Loss_D: 0.7881  Loss_G: 6.2703  D(x): 0.9287    D(G(z)): 0.4413 / 0.0038
[1/5][400/1583] Loss_D: 0.5074  Loss_G: 3.1325  D(x): 0.7897    D(G(z)): 0.1819 / 0.0643
[1/5][450/1583] Loss_D: 0.4046  Loss_G: 3.2195  D(x): 0.7891    D(G(z)): 0.1097 / 0.0570
[1/5][500/1583] Loss_D: 0.9081  Loss_G: 6.0022  D(x): 0.9280    D(G(z)): 0.5038 / 0.0051
```

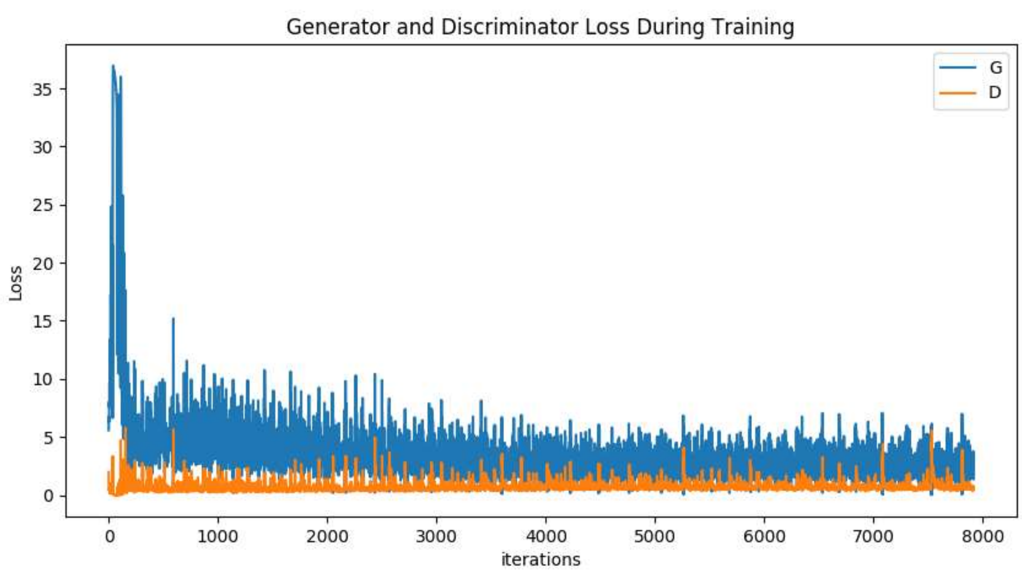
Results

Finally, lets check out how we i . Here, we will look at three ifferent results. First, we will see how D an G’s losses change uring training. Secon , we will visualize G’s output on the fixe _noise atch for every epoch. An thir , we will look at a atch of real ata next to a atch of fake ata from G.

Loss versus training iteration

Below is a plot of D & G’s losses versus training iterations.

```
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()
```

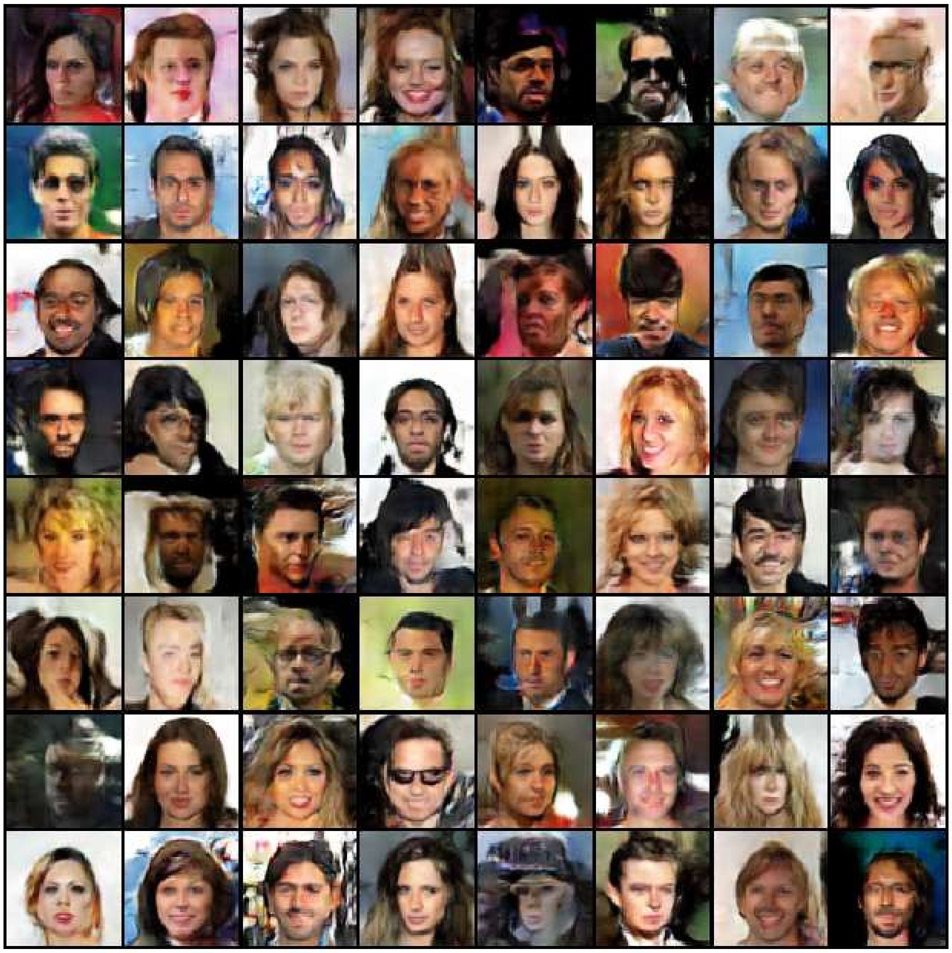


Visualization of G’s rogression

Remem er how we save the generator’s output on the fixe _noise atch after every epoch of training. Now, we can visualize the training progression of G with an animation. Press the play utton to start the animation.

```
#%%capture
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)

HTML(ani.to_jshtml())
```



Real Images vs. Fake Images

Finally, lets take a look at some real images an fake images si e y si e.

```
# 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.show()
```

Where to Go Next

We have reache the en of our ourney, ut there are several places you coul go from here. You coul :

- Train for longer to see how goo the results get
- Mo ify this mo el to take a ifferent ataset an possi ly change the size of the images an the mo el architecture
- Check out some other cool GAN pro ects here
- Create GANs that generate music

Total running time of the scri t: (28 minutes 31.275 secon s)

< Previous   Next >

Was this hel ful?   Yes   No

Docs
Access com rehensive evelo er ocumentation for PyTorch
View Docs >

Tutorials
Get in- e th tutorials for eginners an a vance evelo ers
View Tutorials >

Resources
Fin evelo ment resources an get your uestions answere
View Resources >