exerciseDCGANs

April 10, 2022

1 Exercise of DCGANs with PyTorch

1.1 1. First of all, you must load the EMNIST dataset, which is already available for download in the PyTorch library, from torchvision.datasets and 2. You must convert the images from one channel (grayscale) to three channel (RGB), see torchvision.transforms.

For the second task, it is necessary change the num of outputs channels to 3 using *transforms.Grayscale()*

[19]:
```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')
```

[20]:
```
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)
```

```
Random Seed:  999
```

[20]:
```
<torch._C.Generator at 0x7f362c227950>
```

[21]:
```
# Root directory for dataset
dataroot = "data/EMNIST"

# 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 = 10

# 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
```

[22]:
```
import torchvision.datasets as datasets
transform=transforms.Compose([
                              transforms.Grayscale(num_output_channels=3),
```

```
                              transforms.Resize(image_size),
                              transforms.CenterCrop(image_size),
                              transforms.ToTensor(),
                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,␣
 ↪0.5)),
                              ])



dataset = datasets.EMNIST(
    split="digits",
    root=dataroot,
    download=True,
    transform=transform
)

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

1.2 3. You must create a subset of EMNIST which contains the samples corresponding to a specific character class. The EMNIST dataset contains samples of 36 character classes. Class 0 is the number zero, class 1 is the number one, up to class 9 which is the number nine. Then class 10 is letter A, class 11 is letter B, up to class 35 which is letter Z. You must choose one of the 36 character classes which is not too simple, for example, letter I or number zero are not eligible. The Subset class of torch.utils.data must be employed.

In this case, number 2 has been chosen.

[23]:
```
classe = torch.tensor([2])
indices = (torch.tensor(dataset.targets)[..., None] == classe).any(-1).
 ↪nonzero(as_tuple=True)[0]
subset = torch.utils.data.Subset(dataset, indices)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: UserWarning: To
copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
```

[24]:
```
dataloader = torch.utils.data.DataLoader(subset, batch_size=batch_size,
                                         shuffle=True, num_workers=1)
```

1.3 4. You must show some examples of the training subset, and you must plot the evolution of the loss of the generator and discriminator networks during training. Also, you must show an animation of the evolution of the generated characters for a set of fixed random noise. Furthermore, you must show some examples of real images side by side with fake images after the DCGAN is trained.

[25]:
```
# Plot some training images
real_batch = next(iter(dataloader))
print(real_batch[0].shape)
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(),(2,1,0)))
```

```
torch.Size([128, 3, 64, 64])
```

[25]:
```
<matplotlib.image.AxesImage at 0x7f361e604650>
```

Training Images

Define the weights initialization function

[26]:
```
# 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)
```

Define the generator of the DCGAN. Please note the traposed convolutional layers

[27]:
```
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 and initialize the generator of the DCGAN

[28]:
```
# 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)
```

```
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()
  )
)
```

Define the discriminator of the DCGAN. Please note the convolutional layers.

[29]:
```python
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 and initialize the discriminator.

[30]:
```
# 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)
```

```
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()
  )
)
```

Setup the training procedure

[31]:
```
# 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))
```

Execute the training loop and keep track of the losses of the generator and discriminator

[ ]:
```
# 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
```

```
Starting Training Loop…
[0/10][0/188]   Loss_D: 1.8096  Loss_G: 1.7687  D(x): 0.2882    D(G(z)): 0.2659
/ 0.2219
[0/10][50/188]  Loss_D: 0.0403  Loss_G: 16.9007 D(x): 0.9742    D(G(z)): 0.0000
/ 0.0000
[0/10][100/188] Loss_D: 0.0957  Loss_G: 6.3213  D(x): 0.9401    D(G(z)): 0.0243
/ 0.0026
[0/10][150/188] Loss_D: 0.1444  Loss_G: 4.9184  D(x): 0.9601    D(G(z)): 0.0884
/ 0.0128
[1/10][0/188]   Loss_D: 0.1372  Loss_G: 4.2046  D(x): 0.9350    D(G(z)): 0.0611
/ 0.0227
[1/10][50/188]  Loss_D: 0.4664  Loss_G: 4.6786  D(x): 0.9246    D(G(z)): 0.2546
/ 0.0182
[1/10][100/188] Loss_D: 0.4721  Loss_G: 6.3589  D(x): 0.9629    D(G(z)): 0.2874
/ 0.0038
[1/10][150/188] Loss_D: 0.3595  Loss_G: 3.9495  D(x): 0.8840    D(G(z)): 0.1920
/ 0.0269
[2/10][0/188]   Loss_D: 0.4096  Loss_G: 3.9133  D(x): 0.9234    D(G(z)): 0.2553
/ 0.0284
[2/10][50/188]  Loss_D: 0.3850  Loss_G: 3.5406  D(x): 0.9360    D(G(z)): 0.2558
/ 0.0400
[2/10][100/188] Loss_D: 0.5121  Loss_G: 3.0635  D(x): 0.9024    D(G(z)): 0.3063
/ 0.0630
[2/10][150/188] Loss_D: 0.4565  Loss_G: 2.3720  D(x): 0.8941    D(G(z)): 0.2591
/ 0.1189
[3/10][0/188]   Loss_D: 0.7445  Loss_G: 1.0178  D(x): 0.5812    D(G(z)): 0.0931
/ 0.4229
[3/10][50/188]  Loss_D: 0.6506  Loss_G: 2.0276  D(x): 0.7748    D(G(z)): 0.2910
/ 0.1694
[3/10][100/188] Loss_D: 0.6376  Loss_G: 1.3054  D(x): 0.6201    D(G(z)): 0.0815
/ 0.3208
[3/10][150/188] Loss_D: 0.8051  Loss_G: 2.3010  D(x): 0.7851    D(G(z)): 0.3914
/ 0.1175
[4/10][0/188]   Loss_D: 0.6328  Loss_G: 3.0015  D(x): 0.9105    D(G(z)): 0.3722
/ 0.0674
[4/10][50/188]  Loss_D: 0.4755  Loss_G: 2.3471  D(x): 0.8315    D(G(z)): 0.2322
/ 0.1159
```

```
[4/10][100/188] Loss_D: 0.8975  Loss_G: 4.0375  D(x): 0.9226    D(G(z)): 0.5235
/ 0.0264
[4/10][150/188] Loss_D: 0.4175  Loss_G: 1.9665  D(x): 0.8136    D(G(z)): 0.1718
/ 0.1718
[5/10][0/188]   Loss_D: 0.6932  Loss_G: 3.1106  D(x): 0.9207    D(G(z)): 0.4280
/ 0.0576
[5/10][50/188]  Loss_D: 0.5295  Loss_G: 2.0228  D(x): 0.7500    D(G(z)): 0.1828
/ 0.1610
[5/10][100/188] Loss_D: 0.5633  Loss_G: 1.4764  D(x): 0.7512    D(G(z)): 0.2176
/ 0.2663
[5/10][150/188] Loss_D: 0.5124  Loss_G: 1.4874  D(x): 0.7258    D(G(z)): 0.1469
/ 0.2574
[6/10][0/188]   Loss_D: 0.5979  Loss_G: 1.5917  D(x): 0.6741    D(G(z)): 0.1366
/ 0.2497
[6/10][50/188]  Loss_D: 0.5681  Loss_G: 1.7993  D(x): 0.7582    D(G(z)): 0.2222
/ 0.1988
[6/10][100/188] Loss_D: 0.6468  Loss_G: 2.3423  D(x): 0.8570    D(G(z)): 0.3585
/ 0.1211
[6/10][150/188] Loss_D: 0.4987  Loss_G: 2.1039  D(x): 0.8002    D(G(z)): 0.2200
/ 0.1480
[7/10][0/188]   Loss_D: 0.5592  Loss_G: 1.9950  D(x): 0.8326    D(G(z)): 0.2924
/ 0.1563
[7/10][50/188]  Loss_D: 1.4745  Loss_G: 1.1108  D(x): 0.4038    D(G(z)): 0.1867
/ 0.3871
[7/10][100/188] Loss_D: 0.6221  Loss_G: 2.3307  D(x): 0.8005    D(G(z)): 0.3050
/ 0.1165
[7/10][150/188] Loss_D: 0.4876  Loss_G: 2.3256  D(x): 0.7644    D(G(z)): 0.1750
/ 0.1171
[8/10][0/188]   Loss_D: 0.8994  Loss_G: 1.1452  D(x): 0.5661    D(G(z)): 0.1917
/ 0.3498
[8/10][50/188]  Loss_D: 1.8691  Loss_G: 0.6565  D(x): 0.2246    D(G(z)): 0.0424
/ 0.5598
[8/10][100/188] Loss_D: 1.0452  Loss_G: 1.5177  D(x): 0.6471    D(G(z)): 0.4067
/ 0.2488
[8/10][150/188] Loss_D: 0.6780  Loss_G: 2.4374  D(x): 0.8231    D(G(z)): 0.3546
/ 0.1017
[9/10][0/188]   Loss_D: 0.8022  Loss_G: 3.5899  D(x): 0.9002    D(G(z)): 0.4471
/ 0.0374
[9/10][50/188]  Loss_D: 0.8161  Loss_G: 1.3643  D(x): 0.5836    D(G(z)): 0.1755
/ 0.2979
[9/10][100/188] Loss_D: 0.8804  Loss_G: 1.9698  D(x): 0.8260    D(G(z)): 0.4509
/ 0.1734
```

Plot the progress of the losses during training

[ ]:
```
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()
```

Show an animation of the generated images as the training progressed

```
[ ]: #%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(2,1,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,␣
 ↪blit=True)

HTML(ani.to_jshtml())
```

Plot some real and fake images side by side

```
[ ]: # 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(),(2,1,0)))

# 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],(2,1,0)))
plt.show()
```

1.4 Optional task4: In order to generate a good quality PDF, you may put the following code as the last cell of your notebook:

```
[ ]: !wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('exerciseDCGANs.ipynb')
```