DCGAN by PyTorch

This is a Deep Convolutional Generative Adversarial Networks(DCGAN) using Street View House Numbers Dataset(SVHN).

The code is originally Udacity's Deep Learning tutorial, dcgan-svhn. It is wriiten by TensorFlow. Recently, another major machine learning library, PyTorch drew people's attention. I'm among them. Out of my curiosity, I tried the same DCGAN by PyTorch. For this attempt, some parts of code were taken from PyTorch's dcgan example.

The model is created following the Original DCGAN paper. The dataset is the Street View House Numbers Dataset (SVHN), whose size is 32 x 32.

The notebook works on Google's colaboratory environment with setting of GPU for its runtime. The code includes PyTorch installation since PyTorch is not preinstalled on colaboratory at this moment.

PyTorch Installation and import

The colaboratory provides the code to install PyTorch. However, it doesn't install latest version, so PyTorch was installed simply calling pip install.

```
# http://pytorch.org/
from os.path import exists
from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())
cuda_output = !ldconfig -p|grep cudart.so|sed -e 's/.*\.\([0-9]*\)\.\([0-9]*\)$/cu\1
accelerator = cuda_output[0] if exists('/dev/nvidia0') else 'cpu'

!pip3 install torch torchvision
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: torchvision in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.6/dist
```

Other libraries

To show SVHN images, the cell below imports three libraries.

```
%matplotlib  inline

import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
```

Parameters

Below is definitions of various parameters.

```
dataroot = './data'     # path to dataset
batchSize = 128         # input batch size
workers = 2             # number of data loading workers
nz = 100                # size of a noise vector z
ngf = 64                # generator factor
ndf = 64                # discriminator factor
nc = 3                  # number of color channels
lr = 0.0002             # learning rate
alpha = 0.2             # leaky ReLU parameter
beta1 = 0.9             # Adam Optimizer parameter
niter = 25              # number of epochs

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

```
cuda:0
```

Getting the dataset

The cell below downloads the SVHN dataset using torchvision. Also, dataloader are created using PyTorch's DataLoader.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)
trainset = torchvision.datasets.SVHN(root=dataroot, split='train',
                                     transform=transforms.Compose([
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5,
                                     ]),
                                     download=True)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batchSize,
                                          shuffle=True, num_workers=workers)
testset = torchvision.datasets.SVHN(root=dataroot, split='test',
                                    transform=transforms.Compose([
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5,
                                     ]),
                                    download=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=batchSize,
                                         shuffle=False, num_workers=workers)
```

```
Using downloaded and verified file: ./data/train_32x32.mat
Using downloaded and verified file: ./data/test_32x32.mat
```

Sample images

The SVHN files are .mat files typically used with Matlab. Those are loaded using scipy.io.loadmat which is imported above. Each image is 32 x 32 with 3 color channels (RGB). These are the real images to be passed to the discriminator. The generator is expected to create the same image eventually.

```
trainset = loadmat(dataroot + '/train_32x32.mat')
testset = loadmat(dataroot + '/test_32x32.mat')


idx = np.random.randint(0, trainset['X'].shape[3], size=36)
fig, axes = plt.subplots(6, 6, sharex=True, sharey=True, figsize=(5,5),)
for ii, ax in zip(idx, axes.flatten()):
    ax.imshow(trainset['X'][:,:,:,ii], aspect='equal')
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
plt.subplots_adjust(wspace=0, hspace=0)
```

Generator

The input will be a noise vector z. The output will be a $tanh$ output with size of 32x32 which is the size of the SVHN images.

The generator will have convolutional layers to create new images. The first layer creates a deep and narrow layer, something like 4x4x512 as in the original DCGAN paper. In the layer, a batch normalization and a leaky ReLU activation will be used. following transposed convolution layers halve the depth and double the width and height of the previous layer. Again, the batch normalization and leaky ReLU come here.

```
class Generator(nn.Module):
  def __init__(self):
    super(Generator, self).__init__()
    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.LeakyReLU(alpha, inplace=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.LeakyReLU(alpha, inplace=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.LeakyReLU(alpha, inplace=True),
      # state size. (ngf*2) x 16 x 16
      nn.ConvTranspose2d(ngf * 2, nc, 4, 2, 1, bias=False),
      nn.Tanh()
      # state size. (nc) x 32 x 32
    )

  def forward(self, input):
    output = self.main(input)
    return output
```

Discriminator

The discriminator is basically a convolutional classifier. The inputs to the discriminator are 32x32x3 tensors/images. The discriminator will have a few convolutional layers, then a fully connected layer for the output. At the last layer, an activation is sigmoid.

> In the original DCGAN paper, they did all the downsampling using only strided convolutional layers with no maxpool layers.

```
class Discriminator(nn.Module):
  def __init__(self):
    super(Discriminator, self).__init__()
    self.main = nn.Sequential(
      # input is (nc) x 32 x 32
      nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
      nn.LeakyReLU(alpha, inplace=True),
      # state size. (ndf) x 16 x 16
      nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ndf * 2),
      nn.LeakyReLU(alpha, inplace=True),
      # state size. (ndf*2) x 8 x 8
      nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ndf * 4),
      nn.LeakyReLU(alpha, inplace=True),
      # state size. (ndf*4) x 4 x 4
      nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
      nn.Sigmoid()
    )

  def forward(self, input):
    output = self.main(input)
    return output.view(-1, 1).squeeze(1)
```

Initializes models, defines a loss function and optimizers

Here, the generator and discriminator models are initialized with weights. The loss function is BCELoss, which is Binary Cross Entropy. The BCELoss is a counterpart of TensorFlow's sigmoid_cross_entropy_with_logits. The optimizer is Adam Optimizer.

```
# custom weights initialization called on models
def weights_init(m):
  classname = m.__class__.__name__
  if classname.find('Conv') != -1:
    m.weight.data.normal_(0.0, 0.02)
  elif classname.find('BatchNorm') != -1:
    m.weight.data.normal_(1.0, 0.02)
    m.bias.data.fill_(0)

g_model = Generator().to(device)
g_model.apply(weights_init)
```

```
print(g_model)

d_model = Discriminator().to(device)
d_model.apply(weights_init)
print(d_model)

# loss function
criterion = nn.BCELoss()

fixed_noise = torch.randn(batchSize, nz, 1, 1, device=device)
sample_noise = torch.randn(72, nz, 1, 1, device=device)
real_label = 1
fake_label = 0

# setup optimizer
d_train_optim = optim.Adam(d_model.parameters(), lr=lr, betas=(beta1, 0.999))
g_train_optim = optim.Adam(g_model.parameters(), lr=lr, betas=(beta1, 0.999))
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=Fa
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
    (5): LeakyReLU(negative_slope=0.2, inplace)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
    (8): LeakyReLU(negative_slope=0.2, inplace)
    (9): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), padding=(
    (10): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), b
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
    (4): LeakyReLU(negative_slope=0.2, inplace)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), 
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
    (7): LeakyReLU(negative_slope=0.2, inplace)
    (8): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
```

Showing samples

The cell below defines a function to show samples. The function is called during the training and after the training.

```
def view_samples(epoch, samples, nrows, ncols, figsize=(5,5)):
  fig, axes = plt.subplots(figsize=figsize, nrows=nrows, ncols=ncols,
                           sharey=True, sharex=True)
  for ax, img in zip(axes.flatten(), samples[epoch]):
    img = np.rollaxis(img, 0, 3)
    ax.axis('off')
    img = ((img - img.min())*255 / (img.max() - img.min())).astype(np.uint8)
    ax.set_adjustable('box-forced')
    im = ax.imshow(img, aspect='equal')

  plt.subplots_adjust(wspace=0, hspace=0)
  return fig, axes
```

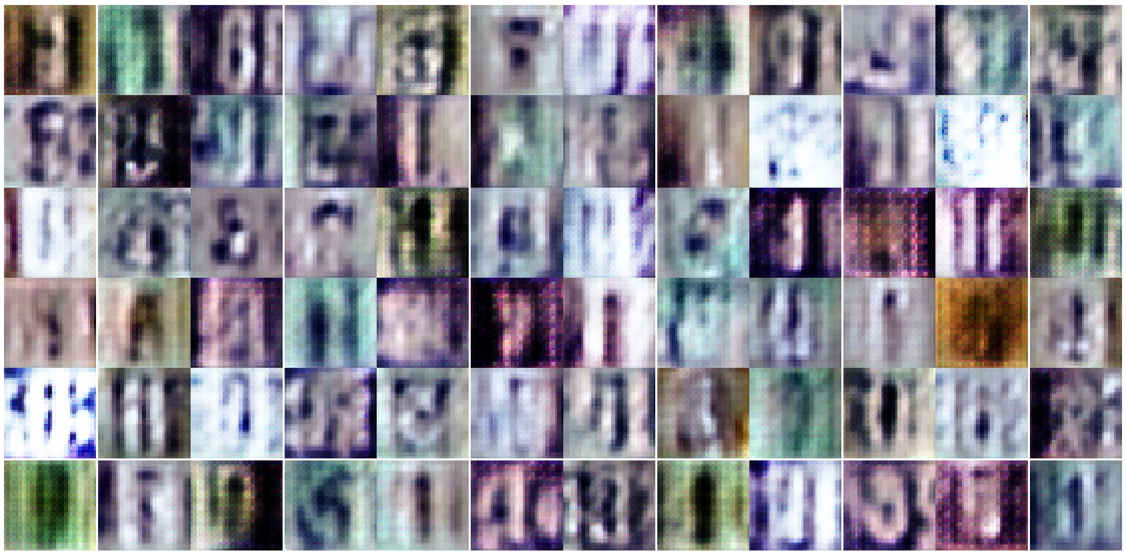
Training

```
print_every=10
show_every=100
samples, losses = [], []

for epoch in range(niter):
  for i, data in enumerate(trainloader, 0):
    if data[0].size()[0] < batchSize:
      continue
    real_data = data[0].to(device)
    ############################
    # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
    ###########################
    # train with real
    d_model.zero_grad()

    output = d_model(real_data)
    label = torch.full((batchSize,), real_label, device=device)

    d_loss_real = criterion(output, label)
    d_loss_real.backward()
    D_x = output.mean().item()

    # train with fake
    noise = torch.randn(batchSize, nz, 1, 1, device=device)
    fake = g_model(noise)
    label.fill_(fake_label)
    output = d_model(fake.detach())
    d_loss_fake = criterion(output, label)
    d_loss_fake.backward()
    D_G_z1 = output.mean().item()
    d_loss = d_loss_real + d_loss_fake
    d_train_optim.step()

    ############################
    # (2) Update G network: maximize log(D(G(z)))
    ###########################
    g_model.zero_grad()
    label.fill_(real_label)  # fake labels are real for generator cost
    output = d_model(fake)
    g_loss = criterion(output, label)
    g_loss.backward()
    D_G_z2 = output.mean().item()
    g_train_optim.step()

    if i % print_every == 0:
      print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f D(x): %.4f D(G(z)): %.4f / %.4
            % (epoch, niter, i, len(trainloader),
               d_loss.item(), g_loss.item(), D_x, D_G_z1, D_G_z2))
      losses.append((d_loss.item(), g_loss.item()))
    if i % show_every == 0:
      gen_samples = g_model(sample_noise).detach()
      sample = gen_samples.to(torch.device('cpu')).numpy()
      samples.append(sample)
      _ = view_samples(-1, samples, 6, 12, figsize=(10,5))
      plt.show()
```

```
[24/25][360/573] Loss_D: 0.8030 Loss_G: 1.7393 D(x): 0.6652 D(G(z)): 0.1978 /
[24/25][370/573] Loss_D: 0.2504 Loss_G: 2.2032 D(x): 0.9270 D(G(z)): 0.1335 /
```

Sample images after the training

```
_ = view_samples(-1, samples, 6, 12, figsize=(10,5))
```



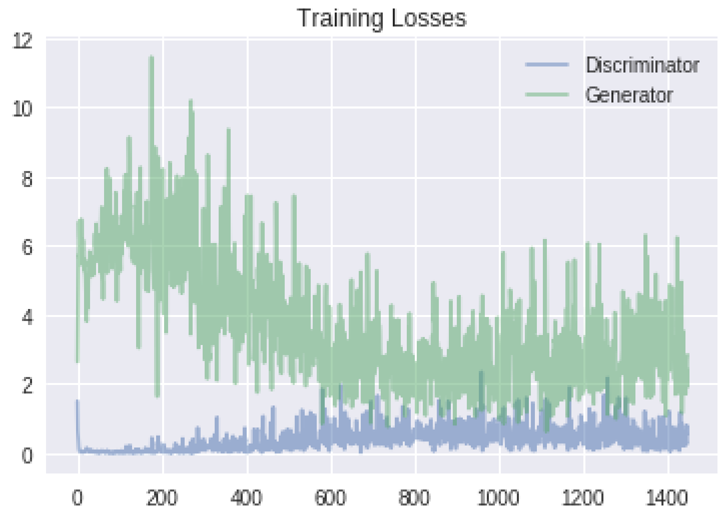
```
[24/25][450/573] Loss_D: 0.2622 Loss_G: 2.6807 D(x): 0.9391 D(G(z)): 0.1537 /
```

Training Losses

```
[24/25][490/573] Loss_D: 0.4831 Loss_G: 2.0445 D(x): 0.7786 D(G(z)): 0.1665 /
```

```
fig, ax = plt.subplots()
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator', alpha=0.5)
plt.plot(losses.T[1], label='Generator', alpha=0.5)
plt.title("Training Losses")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f0459337588>
```



```
[24/25][550/573] Loss_D: 0.8168 Loss_G: 1.8996 D(x): 0.6373 D(G(z)): 0.1757 /
```