CycleGAN

July 7, 2019

0.1 1. Import Libs

```
In [ ]: import torch
        from torch.autograd import Variable
        from torchvision import transforms
        from model import Generator, Discriminator
        import argparse
        import os, itertools
        import numpy as np
        from PIL import Image
        import torch.utils.data as data
        import random
        import matplotlib.pyplot as plt
        import imageio
```

```
In [ ]: def to_np(x):
            return x.data.cpu().numpy()



        def plot_train_result(real_image, gen_image, recon_image, epoch,save_dir='results/', f
            fig, axes = plt.subplots(2, 3, figsize=fig_size)

            imgs = [to_np(real_image[0]), to_np(gen_image[0]), to_np(recon_image[0]),
                    to_np(real_image[1]), to_np(gen_image[1]), to_np(recon_image[1])]
            for ax, img in zip(axes.flatten(), imgs):
                ax.axis('off')
                ax.set_adjustable('box-forced')
                # Scale to 0-255
                img = img.squeeze()
                img = (((img - img.min()) * 255) / (img.max() - img.min())).transpose(1, 2, 0)
                ax.imshow(img, cmap=None, aspect='equal')
            plt.subplots_adjust(wspace=0, hspace=0)

            title = 'Epoch {0}'.format(epoch + 1)
            fig.text(0.5, 0.04, title, ha='center')

            save_fn = save_dir + 'Result_epoch_{:d}'.format(epoch+1) + '.png'
```

```
    plt.savefig(save_fn)
    plt.show()




def plot_test_result(real_image, gen_image, recon_image, index, save_dir='results/'):
    fig_size = (real_image.size(2) * 3 / 100, real_image.size(3) / 100)
    fig, axes = plt.subplots(1, 3, figsize=fig_size)

    imgs = [to_np(real_image), to_np(gen_image), to_np(recon_image)]
    for ax, img in zip(axes.flatten(), imgs):
        ax.axis('off')
        ax.set_adjustable('box-forced')
        # Scale to 0-255
        img = img.squeeze()
        img = (((img - img.min()) * 255) / (img.max() - img.min())).transpose(1, 2, 0)
        ax.imshow(img, cmap=None, aspect='equal')
    plt.subplots_adjust(wspace=0, hspace=0)

    if not os.path.exists(save_dir):
        os.mkdir(save_dir)
    save_fn = save_dir + 'Test_result_{:d}'.format(index + 1) + '.png'
    fig.subplots_adjust(bottom=0)
    fig.subplots_adjust(top=1)
    fig.subplots_adjust(right=1)
    fig.subplots_adjust(left=0)
    plt.savefig(save_fn)
    plt.show()



class ImagePool():
    def __init__(self, pool_size):
        self.pool_size = pool_size
        if self.pool_size > 0:
            self.num_imgs = 0
            self.images = []

    def query(self, images):
        if self.pool_size == 0:
            return images
        return_images = []
        for image in images.data:
            image = torch.unsqueeze(image, 0)
```

```
                    if self.num_imgs < self.pool_size:
                        self.num_imgs = self.num_imgs + 1
                        self.images.append(image)
                        return_images.append(image)
                    else:
                        p = random.uniform(0, 1)
                        if p > 0.5:
                            random_id = random.randint(0, self.pool_size-1)
                            tmp = self.images[random_id].clone()
                            self.images[random_id] = image
                            return_images.append(tmp)
                        else:
                            return_images.append(image)
                return_images = Variable(torch.cat(return_images, 0))
                return return_images
```

```
In [ ]: class DatasetFromFolder(data.Dataset):
            def __init__(self, image_dir, subfolder='train', transform=None, resize_scale=None
                super(DatasetFromFolder, self).__init__()
                self.input_path = os.path.join(image_dir, subfolder)
                self.image_filenames = [x for x in sorted(os.listdir(self.input_path))]
                self.transform = transform

                self.resize_scale = resize_scale
                self.crop_size = crop_size
                self.fliplr = fliplr

            def __getitem__(self, index):
                # Load Image
                img_fn = os.path.join(self.input_path, self.image_filenames[index])
                img = Image.open(img_fn).convert('RGB')

                # preprocessing
                if self.resize_scale:
                    img = img.resize((self.resize_scale, self.resize_scale), Image.BILINEAR)

                if self.crop_size:
                    x = random.randint(0, self.resize_scale - self.crop_size + 1)
                    y = random.randint(0, self.resize_scale - self.crop_size + 1)
                    img = img.crop((x, y, x + self.crop_size, y + self.crop_size))
                if self.fliplr:
                    if random.random() < 0.5:
                        img = img.transpose(Image.FLIP_LEFT_RIGHT)

                if self.transform is not None:
                    img = self.transform(img)

                return img
```

```
    def __len__(self):
        return len(self.image_filenames)
```

0.2 2. Setting Hyperparameters

In [ ]:

```
parser = argparse.ArgumentParser()

#Data Set Parameter
parser.add_argument('--dataset', required=False, default='horse2zebra', help='input da
parser.add_argument('--batch_size', type=int, default=1, help='train batch size')
parser.add_argument('--input_size', type=int, default=256, help='input size')
parser.add_argument('--resize_scale', type=int, default=286, help='resize scale (0 is
parser.add_argument('--crop_size', type=int, default=256, help='crop size (0 is false)
parser.add_argument('--fliplr', type=bool, default=True, help='random fliplr True of Fa

#Model Parameters
parser.add_argument('--ngf', type=int, default=32) # number of generator filters
parser.add_argument('--ndf', type=int, default=64) # number of discriminator filters
parser.add_argument('--num_resnet', type=int, default=6, help='number of resnet blocks

#Learning Parameters
parser.add_argument('--num_epochs', type=int, default=20, help='number of train epochs
parser.add_argument('--decay_epoch', type=int, default=10, help='start decaying learni
parser.add_argument('--lrG', type=float, default=0.0001, help='learning rate for genera
parser.add_argument('--lrD', type=float, default=0.0001, help='learning rate for discri
parser.add_argument('--beta1', type=float, default=0.5, help='beta1 for Adam optimizer
parser.add_argument('--beta2', type=float, default=0.999, help='beta2 for Adam optimize
parser.add_argument('--lambdaA', type=float, default=10, help='lambdaA for cycle loss')
parser.add_argument('--lambdaB', type=float, default=10, help='lambdaB for cycle loss')
params = parser.parse_args([])
print(params)

# Directories for loading data and saving results
data_dir = 'datasets/' + params.dataset + '/'
save_dir = params.dataset + '_results/'
model_dir = params.dataset + '_model/'

if not os.path.exists(save_dir):
    os.mkdir(save_dir)
if not os.path.exists(model_dir):
    os.mkdir(model_dir)
```

0.3 3. Load Dataset

0.3.1 3.2 Preprocessing

```
In [ ]: transform = transforms.Compose([
            transforms.Scale((params.input_size,params.input_size)),
            transforms.ToTensor(),
            transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
        ])
```

0.3.2 3.2 Load Train Data

```
In [ ]: train_data_A = DatasetFromFolder(data_dir, subfolder='trainA', transform=transform,
                                        resize_scale=params.resize_scale, crop_size=params.crop

        train_data_loader_A = torch.utils.data.DataLoader(dataset=train_data_A, batch_size=para

        train_data_B = DatasetFromFolder(data_dir, subfolder='trainB', transform=transform,
                                        resize_scale=params.resize_scale, crop_size=params.crop

        train_data_loader_B = torch.utils.data.DataLoader(dataset=train_data_B, batch_size=para
```

0.3.3 3.3 Load Test Data

```
In [ ]: test_data_A = DatasetFromFolder(data_dir, subfolder='testA', transform=transform)

        test_data_loader_A = torch.utils.data.DataLoader(dataset=test_data_A, batch_size=params

        test_data_B = DatasetFromFolder(data_dir, subfolder='testB', transform=transform)

        test_data_loader_B = torch.utils.data.DataLoader(dataset=test_data_B, batch_size=params


        # Get specific test images
        test_real_A_data = train_data_A.__getitem__(11).unsqueeze(0) # Convert to 4d tensor (B
        test_real_B_data = train_data_B.__getitem__(91).unsqueeze(0)
        print(test_real_A_data)
```

0.4 4. Build Model & Optimizers & Criterions

0.4.1 4.1 Build Model

```
In [ ]: G_A = Generator(3, params.ngf, 3, params.num_resnet) # input_dim, num_filter, output_d
        G_B = Generator(3, params.ngf, 3, params.num_resnet)

        D_A = Discriminator(3, params.ndf, 1) # input_dim, num_filter, output_dim
        D_B = Discriminator(3, params.ndf, 1)

        G_A.normal_weight_init(mean=0.0, std=0.02)
```

```
         G_B.normal_weight_init(mean=0.0, std=0.02)
         D_A.normal_weight_init(mean=0.0, std=0.02)
         D_B.normal_weight_init(mean=0.0, std=0.02)

         print(G_A.cuda())
         print(G_B.cuda())
         print(D_A.cuda())
         print(D_B.cuda())
```

0.4.2 4.2 Optimizers

```
In [ ]: G_optimizer = torch.optim.Adam(itertools.chain(G_A.parameters(), G_B.parameters()), lr
        D_A_optimizer = torch.optim.Adam(D_A.parameters(), lr=params.lrD, betas=(params.beta1,
        D_B_optimizer = torch.optim.Adam(D_B.parameters(), lr=params.lrD, betas=(params.beta1,
```

0.4.3 4.3 Loss Functions

```
In [ ]: MSE_Loss = torch.nn.MSELoss().cuda()
        L1_Loss = torch.nn.L1Loss().cuda()

        # # Training GAN
        D_A_avg_losses = []
        D_B_avg_losses = []
        G_A_avg_losses = []
        G_B_avg_losses = []

        # Generated image pool
        num_pool = 50
        fake_A_pool = ImagePool(num_pool)
        fake_B_pool = ImagePool(num_pool)
```

0.5 5. Training Models

```
In [ ]: step = 0
        for epoch in range(params.num_epochs):
            D_A_losses = []
            D_B_losses = []
            G_A_losses = []
            G_B_losses = []
            cycle_A_losses = []
            cycle_B_losses = []
            if(epoch + 1) > params.decay_epoch:
                D_A_optimizer.param_groups[0]['lr'] -= params.lrD / (params.num_epochs - params
                D_B_optimizer.param_groups[0]['lr'] -= params.lrD / (params.num_epochs - params
                G_optimizer.param_groups[0]['lr'] -= params.lrG / (params.num_epochs - params.


            for i, (real_A, real_B) in enumerate(zip(train_data_loader_A, train_data_loader_B))
```

```
# input image data
real_A = Variable(real_A.cuda())
real_B = Variable(real_B.cuda())

# -------------------------- train generator G --------------------------
# A --> B
fake_B = G_A(real_A)
D_B_fake_decision = D_B(fake_B)
G_A_loss = MSE_Loss(D_B_fake_decision, Variable(torch.ones(D_B_fake_decision.si

# forward cycle loss
recon_A = G_B(fake_B)
cycle_A_loss = L1_Loss(recon_A, real_A) * params.lambdaA

# B --> A
fake_A = G_B(real_B)
D_A_fake_decision = D_A(fake_A)
G_B_loss = MSE_Loss(D_A_fake_decision, Variable(torch.ones(D_A_fake_decision.si

# backward cycle loss
recon_B = G_A(fake_A)
cycle_B_loss = L1_Loss(recon_B, real_B) * params.lambdaB

# Back propagation
G_loss = G_A_loss + G_B_loss + cycle_A_loss + cycle_B_loss
G_optimizer.zero_grad()
G_loss.backward()
G_optimizer.step()


# -------------------------- train discriminator D_A -------------------------
D_A_real_decision = D_A(real_A)
D_A_real_loss = MSE_Loss(D_A_real_decision, Variable(torch.ones(D_A_real_decisi

fake_A = fake_A_pool.query(fake_A)

D_A_fake_decision = D_A(fake_A)
D_A_fake_loss = MSE_Loss(D_A_fake_decision, Variable(torch.zeros(D_A_fake_decisi

# Back propagation
D_A_loss = (D_A_real_loss + D_A_fake_loss) * 0.5
D_A_optimizer.zero_grad()
D_A_loss.backward()
D_A_optimizer.step()

# -------------------------- train discriminator D_B -------------------------
D_B_real_decision = D_B(real_B)
```

```
    D_B_real_loss = MSE_Loss(D_B_real_decision, Variable(torch.ones(D_B_fake_decis

    fake_B = fake_B_pool.query(fake_B)

    D_B_fake_decision = D_B(fake_B)
    D_B_fake_loss = MSE_Loss(D_B_fake_decision, Variable(torch.zeros(D_B_fake_decis

    # Back propagation
    D_B_loss = (D_B_real_loss + D_B_fake_loss) * 0.5
    D_B_optimizer.zero_grad()
    D_B_loss.backward()
    D_B_optimizer.step()

    # ------------------------ Print -----------------------------
    # loss values
    D_A_losses.append(D_A_loss.item())
    D_B_losses.append(D_B_loss.item())
    G_A_losses.append(G_A_loss.item())
    G_B_losses.append(G_B_loss.item())


    if i%10 == 0:
        print('Epoch [%d/%d], Step [%d/%d], D_A_loss: %.4f, D_B_loss: %.4f, G_A_los
              % (epoch+1, params.num_epochs, i+1, len(train_data_loader_A), D_A_los


    step += 1

D_A_avg_loss = torch.mean(torch.FloatTensor(D_A_losses))
D_B_avg_loss = torch.mean(torch.FloatTensor(D_B_losses))
G_A_avg_loss = torch.mean(torch.FloatTensor(G_A_losses))
G_B_avg_loss = torch.mean(torch.FloatTensor(G_B_losses))

# avg loss values for plot
D_A_avg_losses.append(D_A_avg_loss)
D_B_avg_losses.append(D_B_avg_loss)
G_A_avg_losses.append(G_A_avg_loss)
G_B_avg_losses.append(G_B_avg_loss)


# Show result for test image
test_real_A = Variable(test_real_A_data.cuda())
test_fake_B = G_A(test_real_A)
test_recon_A = G_B(test_fake_B)

test_real_B = Variable(test_real_B_data.cuda())
test_fake_A = G_B(test_real_B)
test_recon_B = G_A(test_fake_A)
```

```
            plot_train_result([test_real_A, test_real_B], [test_fake_B, test_fake_A], [test_rec
                                    epoch, save_dir=save_dir)
```

```
In [ ]: # Plot average losses
        avg_losses = []
        avg_losses.append(D_A_avg_losses)
        avg_losses.append(D_B_avg_losses)
        avg_losses.append(G_A_avg_losses)
        avg_losses.append(G_B_avg_losses)

        fig,ax = plt.subplots()
        ax.set_xlim(0,params.num_epochs)
        for i in range(len(avg_losses)):
            temp = max(np.max(avg_losses[i]),temp)
        ax.set_ylim(0,temp*1.2)
        plt.xlabel("Epochs")
        plt.ylabel("Values")
        plt.label("Loss Curve")
        plt.plt(avg_losses[0],label='D_A')
        plt.plt(avg_losses[1],label='D_B')
        plt.plt(avg_losses[2],label='G_A')
        plt.plt(avg_losses[3],label='G_B')
        plt.legend()
        save_fn = save_dir+'Loss_Values_epoch_{:d}'.format(params.num_epochs)+'.png'
        plt.savefig(save_fn)
        plt.show()



        # Save trained parameters of model
        torch.save(G_A.state_dict(), model_dir + 'generator_A_param.pkl')
        torch.save(G_B.state_dict(), model_dir + 'generator_B_param.pkl')
        torch.save(D_A.state_dict(), model_dir + 'discriminator_A_param.pkl')
        torch.save(D_B.state_dict(), model_dir + 'discriminator_B_param.pkl')
```

```
In [ ]: transform = transforms.Compose([
            transforms.Scale((params.input_size,params.input_size)),
            transforms.ToTensor(),
            transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
        ])
```

```
In [ ]: test_data_A = DatasetFromFolder(data_dir, subfolder='testA', transform=transform)
        test_data_loader_A = torch.utils.data.DataLoader(
            dataset=test_data_A, batch_size=params.batch_size, shuffle=False)

        test_data_B = DatasetFromFolder(data_dir, subfolder='testB', transform=transform)
        test_data_loader_B = torch.utils.data.DataLoader(
            dataset=test_data_B, batch_size=params.batch_size, shuffle=False)
```

```
In [ ]: G_A = Generator(3, params.ngf, 3, params.num_resnet)
        G_B = Generator(3, params.ngf, 3, params.num_resnet)
        G_A.cuda()
        G_B.cuda()
        G_A.load_state_dict(torch.load(model_dir + 'generator_A_param.pkl'))
        G_B.load_state_dict(torch.load(model_dir + 'generator_B_param.pkl'))
```

```
In [ ]: for i, real_A in enumerate(test_data_loader_A):
            # input image data
            real_A = Variable(real_A.cuda())

            # A --> B --> A
            fake_B = G_A(real_A)
            recon_A = G_B(fake_B)

            # Show result for test data
            plot_test_result(real_A, fake_B, recon_A, i, save_dir=save_dir + 'AtoB/')

            print('%d images are generated.' % (i + 1))

        for i, real_B in enumerate(test_data_loader_B):

            # input image data
            real_B = Variable(real_B.cuda())

            # B -> A -> B
            fake_A = G_B(real_B)
            recon_B = G_A(fake_A)

            # Show result for test data
            plot_test_result(real_B, fake_A, recon_B, i,  save_dir=save_dir + 'BtoA/')

            print('%d images are generated.' % (i + 1))
```