import os.path

import torch

import helper

import warnings

from distutils.version import LooseVersion

from PIL import Image

import numpy as np

import torchvision.models as models

from torchsummary import summary

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms

import torch.nn.functional as F

from glob import glob

import math

import re

import sys

import random

from matplotlib import pyplot as plt

import torchvision

from torchvision.utils import make\_grid

import cv2

from skimage import io, transform

import scipy.misc

def normalize(img, mean, std):

    img = img/255.0

    img[0] = (img[0] - mean[0]) / std[0]

    img[1] = (img[1] - mean[1]) / std[1]

    img[2] = (img[2] - mean[2]) / std[2]

    img = np.clip(img, 0.0, 1.0)

    return img

from google.colab import drive

drive.mount('/content/drive')

def gen\_batch\_function(mode, image\_shape):

    if mode == 'train':

        data\_folder = os.path.join("/content/drive/MyDrive/", 'data\_road/training')

    elif mode == 'test':

        data\_folder = os.path.join("/content/drive/MyDrive/data\_road", 'data\_road/testing')

    else:

        warnings.warn('No mode selected, please select either ''train'' or ''test''')

    transform\_img = transforms.Compose([transforms.ToTensor()])

    transform\_label = transforms.Compose([transforms.ToTensor()])

    def get\_batches\_fn(batch\_size):

        image\_paths = glob(os.path.join(data\_folder, 'image\_2', '\*.png'))

        if mode =='train':

            label\_paths = {re.sub(r'\_(lane|road)\_', '\_', os.path.basename(path)): path

                           for path in glob(os.path.join(data\_folder, 'gt\_image\_2', '\*\_road\_\*.png'))}

            background\_color = np.array([255, 0, 0])

        # Shuffling training data

        random.shuffle(image\_paths)

        # Loop through batches

        for batch\_i in range(0, len(image\_paths), batch\_size):

            images = torch.zeros([batch\_size,3, image\_shape[0], image\_shape[1]])

            if mode =='train':

                gt\_images = np.zeros([batch\_size, image\_shape[0], image\_shape[1], 3])  # for display only

                labels = torch.zeros([batch\_size,2, image\_shape[0], image\_shape[1]])  # batch x 2 x H x W

            for index, image\_file in enumerate(image\_paths[batch\_i:batch\_i+batch\_size]):

                image = io.imread(image\_file)

                image = np.asarray(image)

                if mode =='train':

                    gt\_image\_file = label\_paths[os.path.basename(image\_file)]

                    gt\_image = io.imread(gt\_image\_file)

                    gt\_image = np.array(gt\_image, dtype=np.uint8)

                    gt\_image = np.asarray(gt\_image)

                    gt\_bg = np.all(gt\_image == background\_color, axis=2)

                    gt\_bg = gt\_bg.reshape(\*gt\_bg.shape, 1)

                    label = np.concatenate((gt\_bg, np.invert(gt\_bg)), axis=2)

                    label = label.astype("float")

                image = cv2.resize(image, (image\_shape[1], image\_shape[0]))

                if mode =='train':

                    label = cv2.resize(label, (image\_shape[1], image\_shape[0]), interpolation=cv2.INTER\_NEAREST)

                    gt\_image = cv2.resize(gt\_image, (image\_shape[1], image\_shape[0]))

                image = transform\_img(image)

                if mode =='train':

                    label = label.transpose(2,0,1)

                    label = torch.from\_numpy(label)

                images[index,:,:,:] = image

                if mode =='train':

                    gt\_images[index,:,:,:] = gt\_image

                    labels[index,:,:] = label

            if mode =='train':

                yield images, labels, gt\_images

            else:

                yield images, image\_paths[batch\_i:batch\_i+batch\_size]

    return get\_batches\_fn

# Visualize sample data

data\_dir = '/content/drive/MyDrive/'

data\_folder = os.path.join(data\_dir, 'data\_road/training')

image\_shape = (256,256)

batch\_size = 8

# get dataloader

get\_batches\_fn = gen\_batch\_function('train', image\_shape)

images, labels, gt\_images = next(get\_batches\_fn(batch\_size))

img = images[4]

print('image scale: \tMin: ', img.min(), '\tMax: ', img.max())

label = labels[4]

print('label scale: \tMin: ', label.min(), '\tMax: ', label.max())

# Display sample images

plt.figure(figsize=(20,40))

for i in range(4):

    plt.subplot(1,4,i+1)

    img = np.copy(images[i])

    img = np.transpose(img, (1, 2, 0))

    plt.imshow(img)

plt.show()

print('shape of images as tensor\t 3 x H x W: \t', images[0].shape)

print('shape of labels as tensor\t  2 x H x W: \t', labels[0].shape)

print()

for i in range(4):

    plt.figure(figsize=(20,40))

    plt.subplot(i+1,3,1)

    img = np.copy(images[i])

    img = np.transpose(img, (1, 2, 0))

    plt.imshow(img)

    plt.title('original image')

    plt.subplot(i+1,3,2)

    gt = np.array(gt\_images[i], dtype=np.uint8)

    plt.imshow(gt)

    plt.title('labeled image')

    plt.subplot(i+1,3,3)

    label = np.transpose(labels[i], (1, 2, 0))

    plt.imshow(label[:,:,1], cmap='gray')

    plt.title('input mask (in grayscale)')

    plt.show()

image\_shape = (images[0].shape[1],images[0].shape[2])

print('\nUpdated image shape in dataloader W x H :', image\_shape)

vgg = models.vgg16(pretrained=True).features

print(vgg.\_modules)

print()

print('layer 0 or input layer: ', vgg.\_modules['0'])

vgg = models.vgg16(pretrained=True)

print(vgg.\_modules['features'])

print()

print('layer 0 or input layer: ', vgg.\_modules['features'][0])

print('CHILDREN')

print(list(vgg.children()))

print()

print('MODULES')

print(list(vgg.modules()))

print()

print('NAMED\_MODULES')

for x in vgg.named\_modules():

    print(x[0], x[1], "\n-------------------------------")

print()

print('NAMED\_CHILDREN')

for x in vgg.named\_children():

    print(x[0], x[1], "\n-------------------------------")

vgg = nn.DataParallel(vgg):

print(vgg.module)

class Encoder(nn.Module):

    def \_\_init\_\_(self, vgg\_path):

        super(Encoder, self).\_\_init\_\_()

        self.vgg\_path = "content/drive/MyDrive/saved\_module"

        if self.vgg\_path != None and os.path.exists(os.path.join(self.vgg\_path,'saved\_model.pt')):

            vgg.load\_state\_dict(torch.load('saved\_model/saved\_model.pt'))

        else:

            self.vgg\_path = '/content/drive/MyDrive/saved\_module'

            self.vgg = models.vgg16(pretrained=True).features

            for param in self.vgg.parameters():

                param.requires\_grad = False

        if not os.path.exists(os.path.join(self.vgg\_path,'saved\_model.pt')):

            try:

                !mkdir -p data/saved\_model

                torch.save(self.vgg.state\_dict(), 'saved\_model.pt')

            except:

                torch.save(self.vgg.state\_dict(), 'saved\_model.pt')

        if not torch.cuda.is\_available():

            summary(self.vgg, (3, 375, 1242))

        else:

            print(self.vgg)

    def forward(self, images):

        vgg\_layer3\_out\_tensor\_name = 'layer3\_out:0'

        vgg\_layer4\_out\_tensor\_name = 'layer4\_out:0'

        vgg\_layer7\_out\_tensor\_name = 'layer7\_out:0'

        layers = {'16' : 'MaxPool2d\_3\_out',

                  '23' : 'MaxPool2d\_4\_out',

                  '30' : 'MaxPool2d\_7\_out'}

        features = {}

        x = images

        for index in range(len(self.vgg.\_modules)):

            layer = self.vgg.\_modules[str(index)]

            x = layer(x)

            if str(index) in layers.keys():

                features[layers[str(index)]] = x

        return features['MaxPool2d\_3\_out'], features['MaxPool2d\_4\_out'], features['MaxPool2d\_7\_out']

class Decoder(nn.Module):

    def \_\_init\_\_(self, num\_classes=2):

        super(Decoder, self).\_\_init\_\_()

        self.vgg\_layer3\_depth = 256

        self.vgg\_layer4\_depth = 512

        self.vgg\_layer7\_depth = 512

        self.num\_classes = num\_classes

        self.height = image\_shape[0]

        self.width = image\_shape[1]

        self.skip\_vgg\_layer4 = nn.Conv2d(in\_channels = self.vgg\_layer4\_depth, out\_channels = 256,

                                         kernel\_size = (1,1), stride = 1, padding = 0)

        self.skip\_vgg\_layer3 = nn.Conv2d(in\_channels = self.vgg\_layer3\_depth, out\_channels = 128,

                                         kernel\_size = (1,1), stride = 1, padding = 0)

        self.bn1 = nn.BatchNorm2d(256)

        self.bn2 = nn.BatchNorm2d(128)

        self.bn3 = nn.BatchNorm2d(64)

        self.bn4 = nn.BatchNorm2d(32)

        self.bn5 = nn.BatchNorm2d(16)

        self.deconv1 = nn.ConvTranspose2d(in\_channels= 512, out\_channels= 256,

                                                   kernel\_size=2, stride=2, padding=0, dilation=1, output\_padding=0)

        self.deconv2 = nn.ConvTranspose2d(in\_channels= 256, out\_channels= 128,

                                                   kernel\_size=2, stride=2, padding=0, dilation=1, output\_padding=0)

        self.deconv3 = nn.ConvTranspose2d(in\_channels= 128, out\_channels= 64,

                                                   kernel\_size=2, stride=2, padding=0, dilation=1, output\_padding=0)

        self.deconv4 = nn.ConvTranspose2d(in\_channels= 64, out\_channels= 32,

                                                   kernel\_size=2, stride=2, padding=0, dilation=1, output\_padding=0)

        self.deconv5 = nn.ConvTranspose2d(in\_channels= 32, out\_channels= 16,

                                                   kernel\_size=2, stride=2, padding=0, dilation=1, output\_padding=0)

        self.AMP = nn.AdaptiveMaxPool3d(output\_size = (2, self.height, self.width))

        self.model\_init()

        self.print\_tensor\_dimensions = True

    def forward(self, vgg\_layer3\_out, vgg\_layer4\_out, vgg\_layer7\_out):

        self.batch\_size = vgg\_layer3\_out.shape[0]

        self.height = image\_shape[0]

        self.width = image\_shape[1]

        self.vgg\_layer3\_out = vgg\_layer3\_out

        self.vgg\_layer4\_out = vgg\_layer4\_out

        self.vgg\_layer7\_out = vgg\_layer7\_out

        if self.print\_tensor\_dimensions: print('F-32 VGG output dimensions in: \t\t\t\t\t', self.vgg\_layer7\_out.shape)

        self.vgg\_layer4\_logits = self.skip\_vgg\_layer4(self.vgg\_layer4\_out)

        if self.print\_tensor\_dimensions: print('VGG F-16 skip connection post Conv1x1: \t\t\t\t', self.vgg\_layer4\_logits.shape)

        self.vgg\_layer3\_logits = self.skip\_vgg\_layer3(self.vgg\_layer3\_out)

        if self.print\_tensor\_dimensions: print('VGG F-8 skip connection post Conv1x1: \t\t\t\t', self.vgg\_layer4\_logits.shape)

        x = F.relu\_(self.deconv1( self.vgg\_layer7\_out))

        if self.print\_tensor\_dimensions: print('Dimensions of Decoder upsampling output deconv1 : \t\t', x.shape)

        # Skip connection

        x = self.bn1(x.add(self.vgg\_layer4\_logits))

        if self.print\_tensor\_dimensions: print('Dimensions of Decoder upsampling deconv1 + skip FCN-16 : \t', x.shape)

        # Upsampling H,W by 2

        x = F.relu\_(self.deconv2(x))

        if self.print\_tensor\_dimensions: print('Dimensions of Decoder upsampling output deconv2 : \t\t', x.shape)

        # Skip connection

        x = self.bn2(x.add(self.vgg\_layer3\_logits))

        if self.print\_tensor\_dimensions: print('Dimensions of Decoder upsampling deconv2 + skip FCN-8 : \t', x.shape)

        # Upsampling H,W by 8

        x = self.bn3(F.relu\_(self.deconv3(x)))

        if self.print\_tensor\_dimensions: print('Dimensions of upsampling output deconv3 : \t\t\t', x.shape)

        # Upsampling H,W by 16

        x = self.bn4(F.relu\_(self.deconv4(x)))

        if self.print\_tensor\_dimensions: print('Dimensions of upsampling output deconv4 : \t\t\t', x.shape)

        # Upsampling H,W by 32

        x = self.bn5(F.relu\_(self.deconv5(x)))

        if self.print\_tensor\_dimensions: print('Dimensions of upsampling output deconv5 : \t\t\t', x.shape)

        # Bring feature depth to num\_classes

        output = self.AMP(x)

        # We ensure appropriate Tensor shape:  batchsize x num\_classes x H x W

        output = output.view(self.batch\_size,self.num\_classes, self.height, self.width)

        if self.print\_tensor\_dimensions: print('Prediction AMP output dimensions: \t\t\t\t', output.shape)

        # Turn off printing dimensions after instantiation

        self.print\_tensor\_dimensions = False

        return output

    def model\_init(self):

        # We initialize the decoder parameters using Xavier's approach

        torch.nn.init.xavier\_uniform\_(self.deconv1.weight)

        torch.nn.init.xavier\_uniform\_(self.deconv2.weight)

        torch.nn.init.xavier\_uniform\_(self.deconv3.weight)

        torch.nn.init.xavier\_uniform\_(self.deconv4.weight)

        torch.nn.init.xavier\_uniform\_(self.deconv5.weight)

        pass

num\_classes = 2

encoder = Encoder(None)

decoder = Decoder(num\_classes)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

encoder.to(device)

decoder.to(device)

def count\_parameters(model):

    return np.sum(p.numel() for p in model.parameters() if p.requires\_grad)

print('total number of trainable parameters for encoder [%d] and decoder [%d]'

      %(count\_parameters(encoder), count\_parameters(decoder)))

criterion = nn.BCELoss().cuda() if torch.cuda.is\_available() else nn.BCELoss()

params = list(decoder.parameters())

optimizer = optim.Adam(params, lr=0.001, betas=(0.9,0.999), eps=1e-08)

if torch.cuda.is\_available():

    print(torch.cuda.get\_device\_name(0))

num\_classes = 2

data\_dir = '/content/drive/MyDrive'

num\_epochs = 40

batch\_size = 32

print\_every = 10

save\_every = 10

data\_folder = os.path.join(data\_dir, 'data\_road/training')

image\_paths = glob(os.path.join(data\_folder, 'image\_2', '\*.png'))

num\_train = len(image\_paths)

print('Number of training images: ', len(image\_paths))

total\_step = math.ceil(num\_train / batch\_size)

print('number of training steps per batch: ', total\_step)

decoder.train()

get\_batches\_fn\_test = gen\_batch\_function('test', image\_shape)

def get\_test\_paths(test\_path):

    test\_paths = [os.path.basename(path) for path in glob(os.path.join(test\_path, '\*.png'))]

    return test\_paths

def resize\_label(image\_path, label):

    image = io.imread(image\_path)

    label = transform.resize(label, image.shape)

    output = cv2.addWeighted(image, 0.6, label, 0.4, 0, dtype = 0)

    return output

# Visualize sample data

decoder.eval()

# obtain one batch of training images

images = images.to(device)

from IPython.display import display

from PIL import Image

image\_path = '/content/s001.png'

image = Image.open(image\_path)

display(image)

with torch.no\_grad():

    # Pass the inputs through the FCN model.

    vgg\_layer3\_out, vgg\_layer4\_out, vgg\_layer7\_out = encoder(images)

    outputs = decoder(vgg\_layer3\_out, vgg\_layer4\_out, vgg\_layer7\_out) # batch\_size x num\_classes x H x W

    outputs = torch.sigmoid(outputs)

    for index, output in enumerate(outputs):

          break

# Get the predictions

pred = output.detach().cpu().numpy()

pred = pred.transpose((1, 2, 0))

pred = pred.argmax(axis=2)

pred = (pred > 0.5)

pred = pred.reshape(\*pred.shape, 1)

pred = np.concatenate((pred, np.invert(pred)), axis=2).astype('float')

pred = np.concatenate((pred, np.zeros((\*pred[:,:,0].shape, 1))), axis=2).astype('float')

image\_path = '/content/s002.png'

image = Image.open(image\_path)

display(image)

pred[pred == 1.0] = 127.0

# Combine mask with image

img = io.imread(image\_paths[index])

pred = cv2.resize(pred, (img.shape[1],img.shape[0]))

img = np.array(img, dtype='uint8')

# Merge original image with mask

street = cv2.addWeighted(img, 0.6, pred, 0.15, 0, dtype=0)

street = street/127.0

street = np.clip(street, 0.0, 1.0)

image\_path = '/content/s003.png'

image = Image.open(image\_path)

display(image)