ECE695DL: Homework 2

Ashish Gondimalla

February 2021

hw02_ImageNet_Scrapper.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import sys
import torchvision
import torch.utils.data
import glob
import os
import numpy
import PIL
import argparse
import requests
import logging
import json
from tqdm import tqdm


# In[2]:

#Define downloading and downsampling images from URL
from PIL import Image
from requests.exceptions import ConnectionError, ReadTimeout,
TooManyRedirects, MissingSchema, InvalidURL

def get_image_from_url(img_url):
    if len(img_url) <= 1:
        return -1
```

```
    try:
        img_resp = requests.get(img_url, timeout=1 )
    except ConnectionError:
        tqdm.write("Could not fetch image from url due to connection error!")
        return -1
    except ReadTimeout:
        tqdm.write("Could not fetch image from url. Server readtimeout!")
        return -1
    except TooManyRedirects:
        tqdm.write("Could not fetch image from url due to too many redirects!")
        return -1
    except MissingSchema:
        tqdm.write("Missing schema for the url!")
        return -1
    except InvalidURL:
        tqdm.write(str("URL is incorrect! Given URL: " + img_url))
        return -1

    if not 'content-type' in img_resp.headers:
        tqdm.write(str("No content in the image url:" + img_url))
        return -1 #sys.exit(0) # or skip?

    if not 'image' in img_resp.headers['content-type']:
        tqdm.write(str("Content type in not image from the url"))
        return -1 #sys.exit(0) or skip?

    return img_resp

# In[3]:

#Get url identifier from json file
def get_identifier_from_json(subclass ,info_json):
    info_file = open(info_json)
    class_info = json.load(info_file)

    for key in class_info.keys():
        if(class_info[key]["class_name"] == subclass):
            return key

#get_identifier_from_json("Siamese cat","./imagenet_class_info.json")


# In[4]:

def parse_args():
    parser = argparse.ArgumentParser(description='HW02 Task1')
```

```
    parser.add_argument('--subclass_list', nargs='*', type=str, required=True)
    parser.add_argument('--images_per_subclass', type=int, required=True)
    parser.add_argument('--data_root', type=str, required=True)
    parser.add_argument('--main_class', type=str, required=True)
    parser.add_argument('--imagenet_info_json', type=str, required=True)

    args, args_other= parser.parse_known_args()

    return args


# In[7]:

def main(args):
    #Extract variables from args

    subclasses = args.subclass_list
    num_images_per_subclass = args.images_per_subclass
    data_dir = args.data_root
    main_class = args.main_class
    imagenet_info_json = args.imagenet_info_json

    #create class directory
    class_path = os.path.join(data_dir, main_class)

    if not os.path.isdir(class_path):
        os.mkdir(class_path)


# In[8]:

    for subclass in subclasses:
        #Create subclass list urls
        subclass_id = get_identifier_from_json(subclass, imagenet_info_json)

        #Example format:
        #list_url = 'http://www.image-net.org/api/text/
        imagenet.synset.geturls?wnid=n02123597'
        subclass_url = "http://www.image-net.org/api/text/
        imagenet.synset.geturls?wnid="+ subclass_id

        resp = requests.get(subclass_url)
        urls = [url.decode('utf-8') for url in resp.content.splitlines()]
        num_images_downloaded = 0
        pbar = tqdm(total=num_images_per_subclass)
```

```
for url in urls:
    if(num_images_downloaded >= num_images_per_subclass ):
        break

    #print(url)
    img_resp = get_image_from_url(url)
    #print(img_resp.content)

    #Bad response, skipping the url.
    if(img_resp == -1):
        continue

    # Images less than 1kB are skipped
    if(len(img_resp.content) < 1000):
        continue

    img_name = url.split('/')[-1]
    img_name = img_name.split("?")[0]

    if(len(img_name) <= 1):
        tqdm.write("Image name is missing")
        sys.exit(0) #Skip instead of exit??

    if not 'flickr' in url:
        tqdm.write("Non-flickr images are difficult to handle,
        skipping...")
        continue

    img_file_path = os.path.join(class_path, img_name)

    if os.path.exists(img_file_path):
        continue

    with open(img_file_path, 'wb') as img_file:
        img_file.write(img_resp.content)

    im = Image.open(img_file_path)

    if im.mode != "RGB":
        im = im.convert(mode="RGB")

    im_resized = im.resize((64,64), Image.BOX)
    im_resized.save(img_file_path)

    #Count images
```

```
            num_images_downloaded += 1
            pbar.update(1)


# In[ ]:

if __name__ == '__main__':
    args = parse_args()
    main(args)
```

---

hw02_imagenet_task2.py

```
import sys
import torchvision
import torch.utils.data
import glob
import os
import numpy
from PIL import Image
import argparse
import requests
import logging
import json
from tqdm import tqdm

from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
import random

#For reproducibility
seed = 123456
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
```

```
torch.backends.cudnn.benchmarks=False
#os.environ['PYTHONHASHSEED'] = str(seed)


class my_dataset(Dataset):
    '''
        class contains datasets for classes cat and dog
        cat is enumerated as 0
        dog is enumerated as 1
    '''
    def __init__(self, data_dir, transform, class_list):
        '''
            Initiates dataset and labels for all classes indicated
            in class list. Loads file paths for all images.
        '''
        #fetch class directories and image names
        self.class_list = class_list
        self.class_paths = []
        self.image_filenames_per_class = {}
        self.transform = transform
        self.total_samples = 0

        for f in os.listdir(data_dir):
            class_path = os.path.join(data_dir , f)
            if f in class_list and os.path.isdir(class_path):
                self.class_paths.append(class_path)
                self.image_filenames_per_class[f] = os.listdir(class_path)
                self.total_samples = len(self.image_filenames_per_class[f])

        #Create labels
        self.labels = {}
        for label, class_name in enumerate(class_list):
            self.labels[class_name] = label

        #Designed for equally distributed samples in classes
        self.samples_per_class = self.total_samples/len(self.class_list)


    def __len__(self):
        '''
            Returns total number of training samples across all classes
        '''
        return self.total_samples

    def __getitem__(self, index):
        '''
```

```
            Fetches one input data in tensor format and associated one-hot label
        '''
        if(index >= self.total_samples):
            raise Exception
            ("Requested Index is higher than total number of samples!")


        sample_class_index = int(index/self.samples_per_class)
        sample_class = self.class_list[sample_class_index]
        sample_path = os.path.join(self.class_paths[sample_class_index], \
                      self.image_filenames_per_class[sample_class]
                      [int(index - sample_class_index*self.samples_per_class)])

        im = Image.open(sample_path)
        sample_data = self.transform(im)

        label = torch.tensor([1-sample_class_index,sample_class_index])

        return sample_data, label


def main(args):
    data_dir = args.imagenet_root
    class_list = args.class_list
    batch_size = 8
    output_file = open("./output.txt", 'w')

    train = True
    test = train

    transform = tvt.Compose([tvt.ToTensor(),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    train_data_dir = os.path.join(data_dir, 'Train')
    train_dataset = my_dataset(train_data_dir, transform, class_list)
    train_data_loader = torch.utils.data.DataLoader(dataset= train_dataset,
    batch_size=batch_size , shuffle=True, num_workers=4)

    val_data_dir = os.path.join(data_dir, 'Val')
    val_dataset = my_dataset(val_data_dir, transform, class_list)
    val_data_loader = torch.utils.data.DataLoader(dataset= val_dataset,
    batch_size=batch_size, shuffle=False, num_workers=4)

    #Model design
    dtype = torch.float64
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
epochs = 40

#Model sizes
D_in, H1, H2, D_out = 3*64*64, 1000, 256, 2

if(train):
    #Model parameters
    w1 = torch.randn(D_in, H1, device=device, dtype=dtype)
    w2 = torch.randn(H1,H2, device=device, dtype=dtype)
    w3 = torch.randn(H2, D_out, device=device, dtype=dtype)

    learning_rate = 1e-9

    #Training iterations
    for it in range(epochs):
        epoch_loss = 0
        for i, data in enumerate(train_data_loader):
            inputs, labels = data
            #print(inputs.size())

            #Copy to device
            inputs = inputs.to(device)
            labels = labels.to(device)

            #Forward propagation
            x = inputs.view(-1, D_in)

            h1  = x.mm(w1.float())
            h1_relu = h1.clamp(min=0)

            h2 = h1_relu.mm(w2.float())
            h2_relu = h2.clamp(min=0)

            y_pred = h2_relu.mm(w3.float())

            #Compute loss
            loss = (y_pred - labels.float()).pow(2).sum().item()
            y_error = y_pred - labels.float()

            #Backward propagation
            grad_w3 = h2_relu.t().mm(2*y_error)
            h2_error = 2.0 * y_error.mm(w3.t().float())
            h2_error[h2 < 0] = 0

            grad_w2 = h1_relu.t().mm(2*h2_error)
```

```
            h1_error = 2.0 * h2_error.mm(w2.t().float())
            h1_error[h1 < 0] = 0

            grad_w1 = x.t().mm(2*h1_error)

            #Weight update
            w1 -= learning_rate*grad_w1.double()
            w2 -= learning_rate*grad_w2.double()
            w3 -= learning_rate*grad_w3.double()

            epoch_loss += loss
        #Loss per epoch
        print('Epoch %d:\t%0.4f'%(it, epoch_loss),file=output_file)

    torch.save({'w1':w1, 'w2':w2, 'w3': w3},'./wts.pkl')
    print('',file=output_file)

if(test):
    weight_dict = torch.load('./wts.pkl',map_location=device)
    w1 = weight_dict['w1']
    w2 = weight_dict['w2']
    w3 = weight_dict['w3']


    #Validation
    val_loss = 0
    val_acc = 0.0
    cor_pred = 0
    tot_pred = 0

    for i, data in enumerate(val_data_loader):
        inputs, labels = data

        inputs = inputs.to(device)
        labels = labels.to(device)

        #Forward propagation
        x = inputs.view(-1, D_in)

        h1  = x.mm(w1.float())
        h1_relu = h1.clamp(min=0)

        h2 = h1_relu.mm(w2.float())
        h2_relu = h2.clamp(min=0)

        y_pred = h2_relu.mm(w3.float())
```

```
            val_loss += (y_pred - labels.float()).pow(2).sum().item()

            class_pred = y_pred.argmax(1)
            class_truth = labels.argmax(1)

            cor_pred += torch.sum(class_pred == class_truth).item()

            tot_pred += y_pred.size()[0]

        val_acc = float(cor_pred)/float(tot_pred)*100.0

        print('Val Loss: ', val_loss,file=output_file)
        print('Val Accuracy: ', val_acc, '%',file=output_file)

        output_file.close()




def parse_args():
    parser = argparse.ArgumentParser(description='HW02 Task2')

    parser.add_argument('--imagenet_root', type=str, required=True)
    parser.add_argument('--class_list', nargs='*', type=str, required=True)

    args, args_other= parser.parse_known_args()

    return args


if __name__ == '__main__':
    args = parse_args()
    main(args)
```