Name: Tahsin Reasat

VUID: Reasatt

Train.py

```
from glob import glob
import os
from albumentations import (
   Compose, Resize, Normalize, RandomBrightnessContrast,
   HorizontalFlip,RandomRotate90,RandomCrop,
CenterCrop
)
import albumentations.pytorch as albu_torch
import sys
sys.path.insert(1,r'..\utility')
sys.path.insert(1,r'..\models')
from dataloader import ISIC_Dataset
from logger import Logger
from loss import bceWithSoftmax
from torch.utils.data import DataLoader
from models import ResNet18, ResNet50, DPN92
import torch.optim as optim
import torch
import time
import argparse
import numpy as np
import pickle
import pandas as pd
from metrics import get_acc,get_recall

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
TIME_STAMP=time.strftime('%Y-%m-%d-%H-%M-%S')

parser=argparse.ArgumentParser()
parser.add_argument('--dir_project', help='project directory', default=r'..')
parser.add_argument('--dir_lf', help='directory large
files',default=r'D:\Data\cs-8395-dl')
parser.add_argument('--folderData', help='data directory', default='assignment2_data')
parser.add_argument('--encoder',help='encoder',default='resnet18')
parser.add_argument('--lr', help='learning rate', type=float, default=0.001)
parser.add_argument('--batchSize', help='batch size', type=int, default=32)
parser.add_argument('--epoch', help='epoch', type=int, default=400)
parser.add_argument('--resume_from', help='filepath to resume training')
parser.add_argument('--bottleneckFeatures', help='bottleneck the encoder Features',
type=int, default=1)
parser.add_argument('--overrideLR', help='override LR from resumed network', type=int,
default=1)
parser.add_argument('--brightness',nargs='+', type=float)
```

```
parser.add_argument('--contrast',nargs='+', type=float)
parser.add_argument('--cropSize', type=int)
parser.add_argument('--resize', type=int)
parser.add_argument('--to_ram',type=int, default=0)
parser.add_argument('--loss_weights',nargs='+', type=float)
args=parser.parse_args()

# setting up directories
DIR_LF = args.dir_lf#r'D:\Data\cs-8395-dl'
dir_data = os.path.join(DIR_LF,args.folderData)
#os.path.join(DIR_LF,'assignment1_data')
dir_model = os.path.join(args.dir_lf, 'model',TIME_STAMP)
dir_history = os.path.join(args.dir_project, 'history')
dir_log = os.path.join(args.dir_project, 'log')
dir_config = os.path.join(args.dir_project, 'config')

if os.path.exists(dir_history) is False:
   os.mkdir(dir_history)
if os.path.exists(dir_log) is False:
   os.mkdir(dir_log)
if os.path.exists(dir_config) is False:
   os.mkdir(dir_config)
if os.path.exists(os.path.join(args.dir_lf, 'model')) is False:
   os.mkdir(os.path.join(args.dir_lf, 'model'))

filepath_hist = os.path.join(dir_history, '{}.bin'.format(TIME_STAMP))
filepath_log = os.path.join(dir_log, '{}.log'.format(TIME_STAMP))
filepath_cfg = os.path.join(dir_config, '{}.cfg'.format(TIME_STAMP))

sys.stdout = Logger(filepath_log)
print(TIME_STAMP)
print(os.path.basename(__file__))
config=vars(args)
config_ls=sorted(list(config.items()))
print('------------------------------------------------------------------------------
-------------------------------------')
for item in config_ls:
   print('{}: {}'.format(item[0],item[1]))
print('------------------------------------------------------------------------------
-------------------------------------')
with open(filepath_cfg, 'w') as file:
   for item in config_ls:
       file.write('{}: {}\n'.format(item[0], item[1]))

if os.path.exists(dir_model)==0:
   print('creating directory to save model at {}'.format(dir_model))
   os.mkdir(dir_model)

filepath_model_best = os.path.join(dir_model, '{}_{}_best.pt'.format(TIME_STAMP,
args.encoder))  ##
```

```
filepath_model_latest = os.path.join(dir_model, '{}_{}_latest.pt'.format(TIME_STAMP,
args.encoder))  ##

dir_data_train = os.path.join(dir_data, 'train')
dir_data_test = os.path.join(dir_data, 'test')

# get train filenames
filepath_train_label = os.path.join(dir_data, 'labels','Train_labels.csv')
df_train = pd.read_csv(filepath_train_label)
df_train.set_index('image',inplace=True)
files_train = df_train.index.values
labels_train_one_hot=[df_train.loc[flname].values for flname in files_train]
labels_train_cat = [np.argmax(label) for label in labels_train_one_hot]
# get test filenames
filepath_test_label = os.path.join(dir_data, 'labels','Test_labels.csv')
df_test = pd.read_csv(filepath_test_label)
df_test.set_index('image',inplace=True)
files_test = df_test.index.values
labels_test_one_hot=[df_test.loc[flname].values for flname in files_test]
labels_test_cat = [np.argmax(label) for label in labels_test_one_hot]

# Dataloader Parameters
aug ={
   'train': Compose([
   HorizontalFlip(),
   RandomRotate90(),
   RandomBrightnessContrast(
       brightness_limit=args.brightness,
       contrast_limit=args.contrast,
   ),
   RandomCrop(args.cropSize, args.cropSize, p=0.5),
   Resize(args.resize,args.resize),
   Normalize(),
   albu_torch.ToTensorV2()
   ]),
   'valid': Compose([
   Resize(args.resize,args.resize),
   Normalize(),
   albu_torch.ToTensorV2()
   ])
}

BATCH_SIZE=args.batchSize
LR = args.lr
EPOCH=args.epoch
Dataset_train = ISIC_Dataset(dir_data=dir_data_train, files=df_train.index.values,
label_cat=labels_train_cat, transform=aug['train'])
loader_train=DataLoader(Dataset_train,batch_size=BATCH_SIZE, shuffle=True)
print('train samples {}'.format(len(Dataset_train)))
```

```
Dataset_valid = ISIC_Dataset(dir_data=dir_data_test, files=df_test.index.values,
label_cat=labels_test_cat, transform=aug['valid'])
loader_valid=DataLoader(Dataset_valid,batch_size=BATCH_SIZE, shuffle=False)
print('validation samples {}'.format(len(Dataset_valid)))
# Model
if args.encoder == 'resnet18':
   model = ResNet18(pretrained=True,
bottleneckFeatures=args.bottleneckFeatures).to(device)
if args.encoder == 'resnet50':
   model = ResNet50(pretrained=True,
bottleneckFeatures=args.bottleneckFeatures).to(device)
if args.encoder == 'dpn92':
   model = DPN92().to(device)

# print(model)

# Optimizer
optimizer = optim.Adam(model.parameters(), lr=LR, betas=(0.9, 0.999), eps=1e-08,
weight_decay=0,
                               amsgrad=False)

# Train
if args.resume_from is not None:
   # Resume?
   print('resuming training from {}'.format(args.resume_from))
   train_states = torch.load(args.resume_from)
   model.load_state_dict(train_states['model_state_dict'])
   if args.overrideLR==0:
       optimizer.load_state_dict(train_states['optimizer_state_dict'])
   epoch_range = np.arange(train_states['epoch']+1, train_states['epoch']+1+EPOCH)
else:
   train_states = {
               'epoch': 0,
               'model_state_dict': model.state_dict(),
               'optimizer_state_dict': optimizer.state_dict(),
               'model_save_criteria': np.inf,
           }
   epoch_range = np.arange(1,EPOCH+1)

loss_train=[]
loss_valid=[]
acc_train = []
acc_valid=[]
recall_macro_valid = []
recall_micro_valid = []
compute_loss = bceWithSoftmax(weights=args.loss_weights)
for epoch in epoch_range:
   running_loss = 0
   running_acc = 0
   model.train()
```

```
    for i, sample in enumerate(loader_train):
        optimizer.zero_grad()
        img = sample[0].to(device)
        target = sample[1].to(device)
        output = model(img)
        # print(target,output)
        loss = compute_loss(output,target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        running_acc += get_acc(target.cpu(),output.cpu())
        mean_loss = running_loss / (i + 1)
        mean_acc = running_acc / (i + 1)
        print('train >>> epoch: {}/{}, batch: {}/{}, mean_loss: {:.4f}, mean_acc:
{:.4f}'.format(
            epoch,
            epoch_range[-1],
            i+1,
            len(loader_train),
            mean_loss,
            mean_acc

        ))
    loss_train.append(mean_loss)
    acc_train.append(mean_acc)
    model.eval()
    running_loss = 0
    output_all=torch.FloatTensor([])
    target_all=torch.FloatTensor([])
    with torch.no_grad():
        for i, sample in enumerate(loader_valid):
            img = sample[0].to(device)
            target = sample[1].to(device)
            output = model(img)
            output_all=torch.cat((output_all,output.float().cpu()),dim=0)
            target_all=torch.cat((target_all,target.float().cpu()),dim=0)
            loss = compute_loss(output,target)
            running_loss += loss.item()
            running_acc += get_acc(target.cpu(),output.cpu())
            mean_loss = running_loss / (i + 1)

    recall_macro = get_recall(target_all, output_all, average='macro')
    recall_micro = get_recall(target_all, output_all, average='micro')
    mean_acc=get_acc(target_all, output_all)
    acc_valid.append(mean_acc)
    print('valid >>> epoch: {}/{}, mean_loss: {:.4f}, mean_acc: {:.4f}'.format(
        epoch,
        epoch_range[-1],
        mean_loss,
        mean_acc
```

```
    ))
    print('recall_micro_valid: {:.4f}, recall_macro_valid: {:4f}'.format(recall_micro,
recall_macro))

    loss_valid.append(mean_loss)
    recall_macro_valid.append(recall_macro)
    recall_micro_valid.append(recall_micro)
    # save train history
    log = {
        'loss_train':loss_train,
        'loss_valid':loss_valid,
        'acc_train': acc_train,
        'acc_valid': acc_valid,
        'recall_micro_valid':recall_micro_valid,
        'recall_macro_valid':recall_macro_valid
        }
    with open(filepath_hist, 'wb') as pfile:
        pickle.dump(log, pfile)

    # save best model
    if mean_loss<train_states['model_save_criteria']:
        print('criteria decreased from {:.4f} to {:.4f}, saving best model at
{}'.format(train_states['model_save_criteria'],

mean_loss,

filepath_model_best))
        train_states = {
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'model_save_criteria': mean_loss,
        }
        torch.save(train_states, filepath_model_best)

# save latest model
train_states = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'model_save_criteria': mean_loss,
}
torch.save(train_states, filepath_model_latest)

print(TIME_STAMP)
```

Test.py

```
from glob import glob
import os
```

```
from albumentations import (
   Compose, Resize, Normalize, RandomBrightnessContrast, HorizontalFlip,
CenterCrop
)
import albumentations.pytorch as albu_torch
import sys
sys.path.insert(1,r'..\utility')
sys.path.insert(1,r'..\models')
from dataloader import ISIC_Dataset
from logger import Logger
from loss import bceWithSoftmax
from torch.utils.data import DataLoader
from models import ResNet18, ResNet50, DPN92
import torch.optim as optim
import torch
import time
import argparse
import numpy as np
import pickle
import pandas as pd
from metrics import get_acc,get_recall,conf_mat
from tqdm import tqdm
import matplotlib.pyplot as plt
from metrics import pretty_plot_confusion_matrix
from pandas import DataFrame
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

parser=argparse.ArgumentParser()
parser.add_argument('--filepath', help='project directory', required=True)
parser.add_argument('--encoder',help='encoder',default='dpn92')
parser.add_argument('--batchSize', help='batch size', type=int, default=32)
parser.add_argument('--load_from', help='filepath to load model',
default=r'D:\Data\cs-8395-dl\model\2020-02-10-22-06-20\2020-02-10-22-06-20_dpn92_best.
pt')
parser.add_argument('--resize', type=int, default=256)

args=parser.parse_args()

# setting up directories

BATCH_SIZE=args.batchSize
dir_data_part = os.path.dirname(args.filepath)
files = os.path.basename(args.filepath).split('.')[0]

# Dataloader Parameters
aug =Compose([
   Resize(args.resize,args.resize),
   Normalize(),
   albu_torch.ToTensorV2()
   ])
```

```
Dataset_valid = ISIC_Dataset(dir_data=dir_data_part, files=[files], label_cat=[ 0 ],
do_cc=True,transform=aug)
loader_valid=DataLoader(Dataset_valid,batch_size=BATCH_SIZE, shuffle=False)
# print('validation samples {}'.format(len(Dataset_valid)))
# Model
if args.encoder == 'resnet18':
   model = ResNet18(pretrained=False, bottleneckFeatures=0).to(device)
if args.encoder == 'resnet50':
   model = ResNet50(pretrained=False, bottleneckFeatures=0).to(device)
if args.encoder == 'dpn92':
   model = DPN92().to(device)
# print(model)

# print('loading model from {}'.format(args.load_from))
train_states = torch.load(args.load_from)
# print('loading model from epoch ', train_states['epoch'])
model.load_state_dict(train_states['model_state_dict'])

model.eval()
with torch.no_grad():
   for sample in loader_valid:
       img = sample[0].to(device)
       target = sample[1].to(device)
       output = model(img)
       output=torch.softmax(output,dim=1).detach().cpu().numpy()
       print(output.argmax())
```

Dataloader.py

```
from torch.utils.data import Dataset, DataLoader
from PIL import Image
from tqdm import tqdm
import os
import random
import numpy as np
from skimage import io
import torch
from glob import glob
from skimage import io
from scipy.ndimage import gaussian_filter
from albumentations import (
Resize,HorizontalFlip,
   Compose,
   Normalize,
RandomBrightnessContrast,
CenterCrop,
)
import albumentations.pytorch as albu_torch
```

```
import pandas as pd
from matplotlib import pyplot as plt
from sampler import BalancedBatchSampler
import sys
sys.path.insert(1,r'..\preprocessing')
from color_constancy import ColorConstancy

def reverse_transform(img_t,mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]):
   img_r = np.array(img_t)
   img_r = img_r.transpose([1,2,0])
   img_r = img_r*std+mean
   img_r *=255
   img_r=img_r.astype(np.uint8)
   img_r = np.squeeze(img_r)
   return img_r

class ISIC_Dataset(Dataset):
   def __init__(self, dir_data, files, label_cat, to_ram = False, transform=None,
do_cc=False):
       self.dir_data = dir_data
       self.transform = transform
       self.files = files
       self.to_ram = to_ram
       self.image_all=[]
       self.label_cat=label_cat
       self.do_cc = do_cc
       self.color_constancy = ColorConstancy(verbose=False, thresh_bg=None)
       if self.to_ram:
           print('loading images to RAM')
           for file in tqdm(self.files):
               # file = self.files[idx]
               path_img = os.path.join(self.dir_data, file+'.jpg')
               image = io.imread(path_img)
               if self.do_cc:
                   image=self.color_constancy.comp(image)
               # print(image.shape)
               self.image_all.append(image)

   def __len__(self):
       size = len(self.files)
       return size

   def __getitem__(self, idx):
       if self.to_ram:
           image=self.image_all[idx]
       else:
           # print(self.files[idx])
           path_img = os.path.join(self.dir_data, self.files[idx] + '.jpg')
           image = io.imread(path_img)
       target=self.label_cat[idx]
```

```
        # print(self.files[idx],image.shape)
        transformed=self.transform(image=image)
        img = transformed['image']
        return img,torch.tensor(target)
```

Loss.py

```
import torch

def bceWithSoftmax(weights=None):
    # i didn't the like the official name of the loss hence the function
    if weights is not None:
        weights = torch.FloatTensor(weights).cuda()
    return torch.nn.CrossEntropyLoss(weights)
if __name__=='__main__':
    loss = bceWithSoftmax()
    input = torch.randn(2, 3, requires_grad=True)
    target = torch.empty(2, dtype=torch.long).random_(3)
    print('input',input)
    print('target',target)
    input_sm = torch.softmax(input,dim=1)
    print('softmax',input_sm)
    print(-torch.log(input_sm))
    print(-torch.log(1-input_sm))

    output = loss(input, target)
    print(output)
    # output.backward()
```

Sampler.py

```
#
https://raw.githubusercontent.com/galatolofederico/pytorch-balanced-batch/master/sampl
er.py
is_torchvision_installed = True
try:
    import torchvision
except:
    is_torchvision_installed = False
import torch.utils.data
import random
import torch


class BalancedBatchSampler(torch.utils.data.sampler.Sampler):
    def __init__(self, dataset, labels=None,shuffle=False):
        self.labels = labels
```

```
        self.dataset = dict() # keys are class labels, values are set of sample indices
associated with each label
        self.balanced_max = 0
        self.shuffle = shuffle
        # Save all the indices for all the classes
        for idx in range(0, len(dataset)):
            label = self._get_label(dataset, idx)
            if label not in self.dataset:
                self.dataset[label] = list()
            self.dataset[label].append(idx)
            self.balanced_max = len(self.dataset[label]) \
                if len(self.dataset[label]) > self.balanced_max else self.balanced_max

        # Oversample the classes with fewer elements than the max
        for label in self.dataset:
            while len(self.dataset[label]) < self.balanced_max:
                self.dataset[label].append(random.choice(self.dataset[label]))
        self.keys = list(self.dataset.keys())
        self.currentkey = 0 # keeps track of which class should be sampled
        self.indices = [-1] * len(self.keys) # keeps track of number of samples per
class
        print('balanced_max: ',self.balanced_max)
        print('number of samples in balanced dataset
{}'.format(self.balanced_max*len(self.keys)))
        # print(self.indices)

    def __iter__(self):
        if self.shuffle:
            print('shuffling dataset')
            for label in self.dataset:
                random.shuffle(self.dataset[label])
        # print(self.dataset)

        while self.indices[self.currentkey] < self.balanced_max - 1:
            self.indices[self.currentkey] += 1
            yield
self.dataset[self.keys[self.currentkey]][self.indices[self.currentkey]]
            self.currentkey = (self.currentkey + 1) % len(self.keys) # I geuss an
assertion that currentkey stays between 0 and num_class-1?
        self.indices = [-1] * len(self.keys)
    def _get_label(self, dataset, idx):
        if self.labels is not None:
            return self.labels[idx].item()
        else:
            # Trying guessing
            dataset_type = type(dataset)
            if is_torchvision_installed and dataset_type is torchvision.datasets.MNIST:
                return dataset.train_labels[idx].item()
            elif is_torchvision_installed and dataset_type is
torchvision.datasets.ImageFolder:
```

```
                return dataset.imgs[idx][1]
            else:
                raise Exception("You should pass the tensor of labels to the
constructor as second argument")

    def __len__(self):
        return self.balanced_max * len(self.keys)
```

Metrics.py

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
import torch

#imports
from pandas import DataFrame
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
from matplotlib.collections import QuadMesh
import seaborn as sn

def get_acc(target,output):
    output_sm = torch.softmax(output, dim=1)
    output_cat = output_sm.argmax(dim=1)
    return accuracy_score(target,output_cat)
def get_recall(target,output,average):
    output_sm = torch.softmax(output, dim=1)
    output_cat = output_sm.argmax(dim=1)
    return recall_score(target,output_cat,average=average)
def conf_mat(target, output, labels=None):
    output_sm = torch.softmax(output, dim=1)
    output_cat = output_sm.argmax(dim=1)
    target=target.cpu().numpy()
    output_cat = output_cat.cpu().numpy()
    # print(output_cat)
    mat=confusion_matrix(target, output_cat, labels=labels, sample_weight=None)
    return mat
```

Color_constancy.py

```
import numpy as np
from skimage import io
import argparse
import math
from matplotlib import pyplot as plt

class ColorConstancy():
```

```
def __init__(self,verbose=False, thresh_bg=None):
    self.verbose = verbose
    self.thresh_bg = thresh_bg
def thresh_img(self,img,thresh):
    red_range = thresh[0]!=img[:,:,0]
    green_range = thresh[1]!=img[:,:,1]
    blue_range = thresh[2]!=img[:,:,2]
    valid_range = np.logical_or(red_range, green_range, blue_range)
    return valid_range

def color_constancy(self,img,preserve_range=True):
    e = np.zeros([3])
    for i in range(3):
        x = img[:,:,i]
        if self.thresh_bg is not None:
            x=x[x!=0]
        e[i]=x.mean()
    if self.verbose: print('channel means',e)
    e=e/math.sqrt(sum(e*e))
    if self.verbose: print('illumination estimate',e)
    d=1/(math.sqrt(3)*e)
    if self.verbose: print('correction coefficient',d)
#     print(d)
    img_t= img*d
    for i in range(3):
        if self.verbose:
            print('transformed image channel {} max\min: {}\{}'.format(
            i+1,img_t[:,:,i].max(),img_t[:,:,i].min()))
    if preserve_range:
        if self.verbose:
            print('setting values above 255 to 255')
        img_t=img_t.flatten()
        img_t[img_t>255]=255
        img_t=img_t.reshape(img.shape)
    return img_t.astype(np.uint8)


def compute_cc(self,img,path_skin=None):


    if img.shape[2]>3:
        img=img[:,:,:3]
    if path_skin is not None:
        mask_skin=io.imread(path_skin)
        mask_skin = mask_skin/mask_skin.max()
        if len(mask_skin.shape)<3:
            mask_skin = np.repeat(mask_skin[:, :, np.newaxis], 3, axis=2)
        img = (img*mask_skin).astype(np.uint8)

    if self.thresh_bg is not None:
```

```
    mask = self.thresh_img(img,self.thresh)
    mask = np.repeat(mask[:, :, np.newaxis], 3, axis=2)
    img = img*mask
img_tx = self.color_constancy(img)
return img_tx
```