import os

import copy

import numpy as np

import pandas as pd

from sklearn import metrics

import torch

import torchvision

from torch.utils.tensorboard import SummaryWriter

import cross\_validation as cv

import dataset

import config

def train\_one\_epoch(model, criterion, loader, optimizer):

device = next(model.parameters()).device

running\_loss = 0.0

for i, data in enumerate(loader, 0):

inputs, labels = data.values()

inputs, labels = inputs.to(dtype=config.dtype, device=device), labels.to(dtype=config.dtype, device=device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

# break # for testing purposes

return running\_loss / (i + 1)

def train\_one\_epoch\_accumulated(model, criterion, loader, optimizer, batch\_size\_to\_update):

device = next(model.parameters()).device

running\_loss = 0.0

current\_batch\_size = 0

optimizer.zero\_grad()

for i, data in enumerate(loader, 0):

inputs, labels = data.values()

inputs, labels = inputs.to(dtype=config.dtype, device=device), labels.to(dtype=config.dtype, device=device)

outputs = model(inputs)

loss = criterion(outputs, labels)

# accumulate gradient and then do Adam step

loss.backward()

current\_batch\_size += loader.batch\_size

if current\_batch\_size >= batch\_size\_to\_update:

optimizer.step()

current\_batch\_size = 0

optimizer.zero\_grad()

running\_loss += loss.item()

# break # for testing purposes

return running\_loss / (i + 1)

def evaluate\_performance(model, criterion, loader):

device = next(model.parameters()).device

loss = 0.

labels\_all = []

probs\_all = []

for i, data in enumerate(loader, 0):

inputs, labels = data.values()

inputs, labels = inputs.to(dtype=config.dtype, device=device), labels.to(dtype=config.dtype, device=device)

with torch.no\_grad():

outputs = model(inputs)

probs = torch.sigmoid(outputs)

loss += criterion(outputs, labels).item()

labels\_all.append(labels)

probs\_all.append(probs)

labels\_all = torch.cat(labels\_all, dim=0).cpu().numpy()

probs\_all = torch.cat(probs\_all, dim=0).cpu().numpy()

# binarize labels

labels\_all = labels\_all.argmax(axis=1)

labels\_all = np.stack([1 - labels\_all, labels\_all], axis=1)

auc\_multiclass = metrics.roc\_auc\_score(labels\_all, probs\_all, average=None)

# binarize probs

probs\_all = probs\_all.argmax(axis=1)

probs\_all = np.stack([1 - probs\_all, probs\_all], axis=1)

true = (probs\_all \* labels\_all).sum(axis=0)

accuracy\_multiclass = true / labels\_all.sum(axis=0)

return loss / (i + 1), auc\_multiclass, accuracy\_multiclass

def save\_ckpt(model, ckpt\_path):

torch.save(model, ckpt\_path)

print('Checkpoint saved: ' + ckpt\_path)

def get\_df\_for\_patients(df\_full, patient\_ids\_indexes):

patient\_ids = df\_full['patientid'].unique()

picked = pd.DataFrame()

for idx in patient\_ids[patient\_ids\_indexes]:

picked = pd.concat([picked, df\_full[df\_full['patientid'] == idx]], ignore\_index=True)

return picked

if \_\_name\_\_ == '\_\_main\_\_':

images\_dir = './data/images'

checkpoints\_dir = './checkpoints'

if not os.path.exists(checkpoints\_dir):

os.makedirs(checkpoints\_dir)

data\_transforms = {

'train': torchvision.transforms.Compose([

dataset.RandomRotate(15),

dataset.RandomCrop(512),

dataset.RandomColorShift(),

dataset.Normalize(config.mean, config.std),

dataset.ToTensor(dtype=config.dtype)

]),

'eval': torchvision.transforms.Compose([

dataset.CenterCrop(512),

dataset.Normalize(config.mean, config.std),

dataset.ToTensor(dtype=config.dtype),

]),

}

df = pd.read\_csv('./data/all.csv', usecols=['patientid', 'finding', 'filename'])

covid\_df = df[df.apply(lambda x: 'COVID-19' in str(x), axis=1)]

other\_df = df[df.apply(lambda x: 'COVID-19' not in str(x), axis=1)]

print('Unique COVID-19 images:', len(covid\_df))

print('Unique COVID-19 patients:', len(covid\_df['patientid'].unique()))

print('Unique other images:', len(other\_df))

print('Unique other patients:', len(other\_df['patientid'].unique()))

n\_covid\_unique\_patients = len(covid\_df['patientid'].unique())

n\_covid\_val\_patients = n\_covid\_unique\_patients // config.num\_folds

n\_covid\_train\_patients = n\_covid\_unique\_patients - n\_covid\_val\_patients

n\_other\_unique\_patients = len(other\_df['patientid'].unique())

n\_other\_val\_patients = n\_other\_unique\_patients // config.num\_folds

n\_other\_train\_patients = n\_other\_unique\_patients - n\_other\_val\_patients

print('nubmer of COVID-19 validation patients: ', n\_covid\_val\_patients)

print('nubmer of COVID-19 train patients: ', n\_covid\_train\_patients)

print('nubmer of Other validation patients: ', n\_other\_val\_patients)

print('nubmer of Other train patients: ', n\_other\_train\_patients)

writer = SummaryWriter()

checkpoints\_best\_list = list()

current\_fold = 0

cv\_covid\_gen = cv.get\_folds(n\_covid\_unique\_patients, config.num\_folds)

cv\_other\_gen = cv.get\_folds(n\_other\_unique\_patients, config.num\_folds)

for current\_fold in range(config.num\_folds):

patient\_indices\_covid\_train, patient\_indices\_covid\_val = next(cv\_covid\_gen)

patient\_indices\_other\_train, patient\_indices\_other\_val = next(cv\_other\_gen)

covid\_train\_df = get\_df\_for\_patients(covid\_df, patient\_indices\_covid\_train)

covid\_val\_df = get\_df\_for\_patients(covid\_df, patient\_indices\_covid\_val)

other\_train\_df = get\_df\_for\_patients(other\_df, patient\_indices\_other\_train)

other\_val\_df = get\_df\_for\_patients(other\_df, patient\_indices\_other\_val)

total = len(covid\_train\_df) + len(other\_train\_df)

weights = torch.tensor([len(other\_train\_df) / total, len(covid\_train\_df) / total], dtype=config.dtype, device=config.device)

criterion = torch.nn.BCEWithLogitsLoss(reduction='mean', pos\_weight=weights)

train\_df = pd.concat([covid\_train\_df, other\_train\_df], ignore\_index=True)

val\_df = pd.concat([covid\_val\_df, other\_val\_df], ignore\_index=True)

dataset\_train = dataset.CovidDataset(train\_df, images\_dir, transform=data\_transforms['train'])

dataset\_val = dataset.CovidDataset(val\_df, images\_dir, transform=data\_transforms['eval'])

loader\_train = torch.utils.data.DataLoader(dataset=dataset\_train, batch\_size=config.batch\_size, shuffle=True, num\_workers=config.dataloader\_workers, pin\_memory=True)

loader\_val = torch.utils.data.DataLoader(dataset=dataset\_val, batch\_size=config.batch\_size, shuffle=False, num\_workers=config.dataloader\_workers, pin\_memory=True)

model = torchvision.models.densenet121(pretrained=True)

if config.use\_pretrained\_lungs:

weights = torch.load(config.pretrained\_path, map\_location=config.device)

model.classifier = torch.nn.Linear(1024, 14)

model.load\_state\_dict(weights)

model.classifier = torch.nn.Linear(1024, 2)

model = model.to(device=config.device)

model.eval()

best\_model\_wts = copy.deepcopy(model.state\_dict())

model.train()

optimizer = torch.optim.Adam(model.parameters(), lr=config.learning\_rate, amsgrad=True)

milestones = [100]

scheduler = torch.optim.lr\_scheduler.MultiStepLR(optimizer, milestones, gamma=0.1)

best\_epoch = 0

no\_improvement\_epochs\_passed = 0

best\_auc = 0.

best\_covid\_acc = 0.

for current\_epoch in range(config.num\_epochs):

loss = train\_one\_epoch\_accumulated(model, criterion, loader\_train, optimizer, config.batch\_size\_to\_update)

print('FOLD {}/{} EPOCH {}/{}: '.format(current\_fold, config.num\_folds, current\_epoch, config.num\_epochs), end='')

print('Running loss:', loss)

if (current\_epoch + 1) % 1 == 0: # and current\_epoch >= 100:

model.eval()

val\_loss, val\_auc, val\_acc = evaluate\_performance(model, criterion, loader\_val)

print('Validation loss = {}, mean auc = {}, mean acc = {}'.format(val\_loss, val\_auc.mean(), val\_acc.mean()))

writer.add\_scalars('fold\_' + str(current\_fold) + '/losses', {'train': loss, 'val': val\_loss}, current\_epoch)

for i in range(len(val\_auc)):

writer.add\_scalars('fold\_' + str(current\_fold) + '/aucs/roc\_auc', {'val-' + dataset.label\_names[i]: val\_auc[i]}, current\_epoch)

writer.add\_scalars('fold\_' + str(current\_fold) + '/accs/accs', {'val-' + dataset.label\_names[i]: val\_acc[i]}, current\_epoch)

writer.add\_scalars('fold\_' + str(current\_fold) + '/aucs/mean', {'val': val\_auc.mean()}, current\_epoch)

writer.add\_scalars('fold\_' + str(current\_fold) + '/accs/mean', {'val': val\_acc.mean()}, current\_epoch)

if val\_auc.mean() > best\_auc:

best\_auc = val\_auc.mean()

best\_covid\_acc = val\_acc[0]

best\_epoch = current\_epoch

best\_model\_wts = copy.deepcopy(model.state\_dict())

no\_improvement\_epochs\_passed = 0

else:

no\_improvement\_epochs\_passed += 1

if no\_improvement\_epochs\_passed >= config.plateau\_epochs\_num:

print('Training is stopped because auc metric has reached a plateau. Stopped at epoch {}. Best auc metric at epoch {}.'.format(

current\_epoch, current\_epoch - config.plateau\_epochs\_num))

break

model.train()

scheduler.step()

# break # for testing purposes

# saving best ckpt

checkpoint\_path = os.path.join(checkpoints\_dir, 'ckpt\_fold\_' + str(current\_fold) + '\_epoch\_' + str(best\_epoch) + '\_bestonval\_auc\_' + str(best\_auc) + '\_acc\_covid\_' + str(best\_covid\_acc) + '.pt')

save\_ckpt(best\_model\_wts, checkpoint\_path)

current\_fold += 1

# break # for testing purposes

writer.close()