class Flatten(nn.Module):

def forward(self, x):

return x.view(x.size(0), -1)

class FaceNetModel(nn.Module):

def \_\_init\_\_(self, embedding\_size, num\_classes, pretrained=False):

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

self.model = resnet50(pretrained)

self.embedding\_size = embedding\_size

self.cnn = nn.Sequential(

self.model.conv1,

self.model.bn1,

self.model.relu,

self.model.maxpool,

self.model.layer1,

self.model.layer2,

self.model.layer3,

self.model.layer4)

self.model.fc = nn.Sequential(

Flatten(),

nn.Linear(100352, self.embedding\_size))

self.model.classifier = nn.Linear(self.embedding\_size, num\_classes)

def l2\_norm(self, input):

input\_size = input.size()

buffer = torch.pow(input, 2)

normp = torch.sum(buffer, 1).add\_(1e-10)

norm = torch.sqrt(normp)

\_output = torch.div(input, norm.view(-1, 1).expand\_as(input))

output = \_output.view(input\_size)

return output

def freeze\_all(self):

for param in self.model.parameters():

param.requires\_grad = False

def unfreeze\_all(self):

for param in self.model.parameters():

param.requires\_grad = True

def freeze\_fc(self):

for param in self.model.fc.parameters():

param.requires\_grad = False

def unfreeze\_fc(self):

for param in self.model.fc.parameters():

param.requires\_grad = True

def freeze\_classifier(self):

for param in self.model.classifier.parameters():

param.requires\_grad = False

def unfreeze\_classifier(self):

for param in self.model.classifier.parameters():

param.requires\_grad = True

def freeze\_only(self, freeze):

for name, child in self.model.named\_children():

if name in freeze:

for param in child.parameters():

param.requires\_grad = False

else:

for param in child.parameters():

param.requires\_grad = True

def unfreeze\_only(self, unfreeze):

for name, child in self.model.named\_children():

if name in unfreeze:

for param in child.parameters():

param.requires\_grad = True

else:

for param in child.parameters():

param.requires\_grad = False

# returns face embedding(embedding\_size)

def forward(self, x):

x = self.cnn(x)

x = self.model.fc(x)

features = self.l2\_norm(x)

# Multiply by alpha = 10 as suggested in https://arxiv.org/pdf/1703.09507.pdf

alpha = 10

features = features \* alpha

return features

def forward\_classifier(self, x):

features = self.forward(x)

res = self.model.classifier(features)

return res

class TripletLoss(torch.nn.Module):

def \_\_init\_\_(self, margin):

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

self.margin = margin

self.pdist = PairwiseDistance(2)

def forward(self, anchor, positive, negative):

pos\_dist = self.pdist.forward(anchor, positive)

neg\_dist = self.pdist.forward(anchor, negative)

hinge\_dist = torch.clamp(self.margin + pos\_dist - neg\_dist, min=0.0)

loss = torch.mean(hinge\_dist)

return loss

import os

import numpy as np

import pandas as pd

import torch

from skimage import io

from torch.utils.data import Dataset

from torchvision import transforms

class TripletFaceDataset(Dataset):

def \_\_init\_\_(self, root\_dir, csv\_name, num\_triplets, transform=None):

self.root\_dir = root\_dir

self.df = pd.read\_csv(csv\_name)

self.num\_triplets = num\_triplets

self.transform = transform

self.training\_triplets = self.generate\_triplets(self.df, self.num\_triplets)

@staticmethod

def generate\_triplets(df, num\_triplets):

def make\_dictionary\_for\_face\_class(df):

'''

- face\_classes = {'class0': [class0\_id0, ...], 'class1': [class1\_id0, ...], ...}

'''

face\_classes = dict()

for idx, label in enumerate(df['class']):

if label not in face\_classes:

face\_classes[label] = []

face\_classes[label].append((df.iloc[idx]['id'], df.iloc[idx]['ext']))

return face\_classes

triplets = []

classes = df['class'].unique()

face\_classes = make\_dictionary\_for\_face\_class(df)

for \_ in range(num\_triplets):

'''

- randomly choose anchor, positive and negative images for triplet loss

- anchor and positive images in pos\_class

- negative image in neg\_class

- at least, two images needed for anchor and positive images in pos\_class

- negative image should have different class as anchor and positive images by definition

'''

pos\_class = np.random.choice(classes)

neg\_class = np.random.choice(classes)

while len(face\_classes[pos\_class]) < 2:

pos\_class = np.random.choice(classes)

while pos\_class == neg\_class:

neg\_class = np.random.choice(classes)

pos\_name = df.loc[df['class'] == pos\_class, 'name'].values[0]

neg\_name = df.loc[df['class'] == neg\_class, 'name'].values[0]

if len(face\_classes[pos\_class]) == 2:

ianc, ipos = np.random.choice(2, size=2, replace=False)

else:

ianc = np.random.randint(0, len(face\_classes[pos\_class]))

ipos = np.random.randint(0, len(face\_classes[pos\_class]))

while ianc == ipos:

ipos = np.random.randint(0, len(face\_classes[pos\_class]))

ineg = np.random.randint(0, len(face\_classes[neg\_class]))

anc\_id = face\_classes[pos\_class][ianc][0]

anc\_ext = face\_classes[pos\_class][ianc][1]

pos\_id = face\_classes[pos\_class][ipos][0]

pos\_ext = face\_classes[pos\_class][ipos][1]

neg\_id = face\_classes[neg\_class][ineg][0]

neg\_ext = face\_classes[neg\_class][ineg][1]

triplets.append(

[anc\_id, pos\_id, neg\_id, pos\_class, neg\_class, pos\_name, neg\_name, anc\_ext, pos\_ext, neg\_ext])

return triplets

def \_\_getitem\_\_(self, idx):

anc\_id, pos\_id, neg\_id, pos\_class, neg\_class, pos\_name, neg\_name, anc\_ext, pos\_ext, neg\_ext = \

self.training\_triplets[idx]

anc\_img = os.path.join(self.root\_dir, str(pos\_name), str(anc\_id) + f'.{anc\_ext}')

pos\_img = os.path.join(self.root\_dir, str(pos\_name), str(pos\_id) + f'.{pos\_ext}')

neg\_img = os.path.join(self.root\_dir, str(neg\_name), str(neg\_id) + f'.{neg\_ext}')

anc\_img = io.imread(anc\_img)

pos\_img = io.imread(pos\_img)

neg\_img = io.imread(neg\_img)

pos\_class = torch.from\_numpy(np.array([pos\_class]).astype('long'))

neg\_class = torch.from\_numpy(np.array([neg\_class]).astype('long'))

sample = {'anc\_img': anc\_img, 'pos\_img': pos\_img, 'neg\_img': neg\_img, 'pos\_class': pos\_class,

'neg\_class': neg\_class}

if self.transform:

sample['anc\_img'] = self.transform(sample['anc\_img'])

sample['pos\_img'] = self.transform(sample['pos\_img'])

sample['neg\_img'] = self.transform(sample['neg\_img'])

return sample

def \_\_len\_\_(self):

return len(self.training\_triplets)

def get\_dataloader(train\_root\_dir, valid\_root\_dir,

train\_csv\_name, valid\_csv\_name,

num\_train\_triplets, num\_valid\_triplets,

batch\_size, num\_workers):

data\_transforms = {

'train': transforms.Compose([

transforms.ToPILImage(),

transforms.RandomRotation(15),

transforms.RandomResizedCrop(224),

transforms.RandomHorizontalFlip(),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406],

std=[0.229, 0.224, 0.225])]),

'valid': transforms.Compose([

transforms.ToPILImage(),

transforms.Resize(224),

transforms.CenterCrop(224),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406],

std=[0.229, 0.224, 0.225])])}

face\_dataset = {

'train': TripletFaceDataset(root\_dir=train\_root\_dir,

csv\_name=train\_csv\_name,

num\_triplets=num\_train\_triplets,

transform=data\_transforms['train']),

'valid': TripletFaceDataset(root\_dir=valid\_root\_dir,

csv\_name=valid\_csv\_name,

num\_triplets=num\_valid\_triplets,

transform=data\_transforms['valid'])}

dataloaders = {

x: torch.utils.data.DataLoader(face\_dataset[x], batch\_size=batch\_size, shuffle=False, num\_workers=num\_workers)

for x in ['train', 'valid']}

data\_size = {x: len(face\_dataset[x]) for x in ['train', 'valid']}

return dataloaders, data\_size

import argparse

import time

import numpy as np

import torch

import torch.optim as optim

from torch.nn.modules.distance import PairwiseDistance

from torch.optim import lr\_scheduler

from data\_loader import get\_dataloader

from datasets.write\_csv\_for\_making\_dataset import write\_csv

from eval\_metrics import evaluate, plot\_roc

from loss import TripletLoss

from models import FaceNetModel

from utils import ModelSaver, init\_log\_just\_created

# from utils import VisdomLinePlotter

parser = argparse.ArgumentParser(description='Face Recognition using Triplet Loss')

parser.add\_argument('--num-epochs', default=200, type=int, metavar='NE',

help='number of epochs to train (default: 200)')

parser.add\_argument('--num-classes', default=10000, type=int, metavar='NC',

help='number of clases (default: 10000)')

parser.add\_argument('--num-train-triplets', default=10000, type=int, metavar='NTT',

help='number of triplets for training (default: 10000)')

parser.add\_argument('--num-valid-triplets', default=10000, type=int, metavar='NVT',

help='number of triplets for vaidation (default: 10000)')

parser.add\_argument('--embedding-size', default=128, type=int, metavar='ES',

help='embedding size (default: 128)')

parser.add\_argument('--batch-size', default=64, type=int, metavar='BS',

help='batch size (default: 128)')

parser.add\_argument('--num-workers', default=8, type=int, metavar='NW',

help='number of workers (default: 8)')

parser.add\_argument('--learning-rate', default=0.001, type=float, metavar='LR',

help='learning rate (default: 0.001)')

parser.add\_argument('--margin', default=0.5, type=float, metavar='MG',

help='margin (default: 0.5)')

parser.add\_argument('--train-root-dir',

default='/run/media/hoosiki/WareHouse2/home/mtb/datasets/vggface2/test\_mtcnnpy\_182', type=str,

help='path to train root dir')

parser.add\_argument('--valid-root-dir', default='/run/media/hoosiki/WareHouse2/home/mtb/datasets/lfw/lfw\_mtcnnpy\_182',

type=str,

help='path to valid root dir')

parser.add\_argument('--train-csv-name', default='./datasets/test\_vggface2.csv', type=str,

help='list of training images')

parser.add\_argument('--valid-csv-name', default='./datasets/lfw.csv', type=str,

help='list of validtion images')

parser.add\_argument('--step-size', default=50, type=int, metavar='SZ',

help='Decay learning rate schedules every --step-size (default: 50)')

parser.add\_argument('--unfreeze', type=str, metavar='UF', default='',

help='Provide an option for unfreezeing given layers')

parser.add\_argument('--freeze', type=str, metavar='F', default='',

help='Provide an option for freezeing given layers')

parser.add\_argument('--pretrain', action='store\_true')

parser.add\_argument('--fc-only', action='store\_true')

parser.add\_argument('--except-fc', action='store\_true')

parser.add\_argument('--load-best', action='store\_true')

parser.add\_argument('--load-last', action='store\_true')

parser.add\_argument('--continue-step', action='store\_true')

parser.add\_argument('--train-all', action='store\_true', help='Train all layers')

args = parser.parse\_args()

device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

l2\_dist = PairwiseDistance(2)

modelsaver = ModelSaver()

# plotter = VisdomLinePlotter('Siamese Triplet')

def save\_if\_best(state, acc):

modelsaver.save\_if\_best(acc, state)

def main():

init\_log\_just\_created("log/valid.csv")

init\_log\_just\_created("log/train.csv")

pretrain = args.pretrain

fc\_only = args.fc\_only

except\_fc = args.except\_fc

train\_all = args.train\_all

unfreeze = args.unfreeze.split(',')

freeze = args.freeze.split(',')

start\_epoch = 0

print(f"Transfer learning: {pretrain}")

print("Train fc only:", fc\_only)

print("Train except fc:", except\_fc)

print("Train all layers:", train\_all)

print("Unfreeze only:", ', '.join(unfreeze))

print("Freeze only:", ', '.join(freeze))

print(f"Learning rate will decayed every {args.step\_size}th epoch")

model = FaceNetModel(embedding\_size=args.embedding\_size, num\_classes=args.num\_classes, pretrained=pretrain).to(

device)

triplet\_loss = TripletLoss(args.margin).to(device)

if fc\_only:

model.freeze\_all()

model.unfreeze\_fc()

model.unfreeze\_classifier()

if except\_fc:

model.unfreeze\_all()

model.freeze\_fc()

model.freeze\_classifier()

if train\_all:

model.unfreeze\_all()

if len(unfreeze) > 0:

model.unfreeze\_only(unfreeze)

if len(freeze) > 0:

model.freeze\_only(freeze)

optimizer = optim.Adam(filter(lambda p: p.requires\_grad, model.parameters()), lr=args.learning\_rate)

scheduler = lr\_scheduler.StepLR(optimizer, step\_size=args.step\_size, gamma=0.1)

if args.load\_best or args.load\_last:

checkpoint = './log/best\_state.pth' if args.load\_best else './log/last\_checkpoint.pth'

print('loading', checkpoint)

checkpoint = torch.load(checkpoint)

modelsaver.current\_acc = checkpoint['accuracy']

start\_epoch = checkpoint['epoch'] + 1

model.load\_state\_dict(checkpoint['state\_dict'])

print("Stepping scheduler")

try:

optimizer.load\_state\_dict(checkpoint['optimizer\_state'])

except ValueError as e:

print("Can't load last optimizer")

print(e)

if args.continue\_step:

scheduler.step(checkpoint['epoch'])

print(f"Loaded checkpoint epoch: {checkpoint['epoch']}\n"

f"Loaded checkpoint accuracy: {checkpoint['accuracy']}\n"

f"Loaded checkpoint loss: {checkpoint['loss']}")

model = torch.nn.DataParallel(model)

for epoch in range(start\_epoch, args.num\_epochs + start\_epoch):

print(80 \* '=')

print('Epoch [{}/{}]'.format(epoch, args.num\_epochs + start\_epoch - 1))

time0 = time.time()

data\_loaders, data\_size = get\_dataloader(args.train\_root\_dir, args.valid\_root\_dir,

args.train\_csv\_name, args.valid\_csv\_name,

args.num\_train\_triplets, args.num\_valid\_triplets,

args.batch\_size, args.num\_workers)

train\_valid(model, optimizer, triplet\_loss, scheduler, epoch, data\_loaders, data\_size)

print(f' Execution time = {time.time() - time0}')

print(80 \* '=')

def save\_last\_checkpoint(state):

torch.save(state, 'log/last\_checkpoint.pth')

def train\_valid(model, optimizer, triploss, scheduler, epoch, dataloaders, data\_size):

for phase in ['train', 'valid']:

labels, distances = [], []

triplet\_loss\_sum = 0.0

if phase == 'train':

scheduler.step()

if scheduler.last\_epoch % scheduler.step\_size == 0:

print("LR decayed to:", ', '.join(map(str, scheduler.get\_lr())))

model.train()

else:

model.eval()

for batch\_idx, batch\_sample in enumerate(dataloaders[phase]):

anc\_img = batch\_sample['anc\_img'].to(device)

pos\_img = batch\_sample['pos\_img'].to(device)

neg\_img = batch\_sample['neg\_img'].to(device)

pos\_cls = batch\_sample['pos\_class'].to(device)

neg\_cls = batch\_sample['neg\_class'].to(device)

with torch.set\_grad\_enabled(phase == 'train'):

# anc\_embed, pos\_embed and neg\_embed are encoding(embedding) of image

anc\_embed, pos\_embed, neg\_embed = model(anc\_img), model(pos\_img), model(neg\_img)

# choose the hard negatives only for "training"

pos\_dist = l2\_dist.forward(anc\_embed, pos\_embed)

neg\_dist = l2\_dist.forward(anc\_embed, neg\_embed)

all = (neg\_dist - pos\_dist < args.margin).cpu().numpy().flatten()

if phase == 'train':

hard\_triplets = np.where(all == 1)

if len(hard\_triplets[0]) == 0:

continue

else:

hard\_triplets = np.where(all >= 0)

anc\_hard\_embed = anc\_embed[hard\_triplets]

pos\_hard\_embed = pos\_embed[hard\_triplets]

neg\_hard\_embed = neg\_embed[hard\_triplets]

anc\_hard\_img = anc\_img[hard\_triplets]

pos\_hard\_img = pos\_img[hard\_triplets]

neg\_hard\_img = neg\_img[hard\_triplets]

pos\_hard\_cls = pos\_cls[hard\_triplets]

neg\_hard\_cls = neg\_cls[hard\_triplets]

anc\_img\_pred = model.module.forward\_classifier(anc\_hard\_img)

pos\_img\_pred = model.module.forward\_classifier(pos\_hard\_img)

neg\_img\_pred = model.module.forward\_classifier(neg\_hard\_img)

triplet\_loss = triploss.forward(anc\_hard\_embed, pos\_hard\_embed, neg\_hard\_embed)

if phase == 'train':

optimizer.zero\_grad()

triplet\_loss.backward()

optimizer.step()

dists = l2\_dist.forward(anc\_embed, pos\_embed)

distances.append(dists.data.cpu().numpy())

labels.append(np.ones(dists.size(0)))

dists = l2\_dist.forward(anc\_embed, neg\_embed)

distances.append(dists.data.cpu().numpy())

labels.append(np.zeros(dists.size(0)))

triplet\_loss\_sum += triplet\_loss.item()

avg\_triplet\_loss = triplet\_loss\_sum / data\_size[phase]

labels = np.array([sublabel for label in labels for sublabel in label])

distances = np.array([subdist for dist in distances for subdist in dist])

tpr, fpr, accuracy, val, val\_std, far = evaluate(distances, labels)

print(' {} set - Triplet Loss = {:.8f}'.format(phase, avg\_triplet\_loss))

print(' {} set - Accuracy = {:.8f}'.format(phase, np.mean(accuracy)))

write\_csv(f'log/{phase}.csv', [epoch, np.mean(accuracy), avg\_triplet\_loss])

if phase == 'valid':

save\_last\_checkpoint({'epoch': epoch,

'state\_dict': model.module.state\_dict(),

'optimizer\_state': optimizer.state\_dict(),

'accuracy': np.mean(accuracy),

'loss': avg\_triplet\_loss

})

save\_if\_best({'epoch': epoch,

'state\_dict': model.module.state\_dict(),

'optimizer\_state': optimizer.state\_dict(),

'accuracy': np.mean(accuracy),

'loss': avg\_triplet\_loss

}, np.mean(accuracy))

else:

plot\_roc(fpr, tpr, figure\_name='./log/roc\_valid\_epoch\_{}.png'.format(epoch))

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

main()

import numpy as np

from scipy import interpolate

from sklearn.model\_selection import KFold

def evaluate(distances, labels, nrof\_folds=10):

# Calculate evaluation metrics

thresholds = np.arange(0, 30, 0.01)

tpr, fpr, accuracy = calculate\_roc(thresholds, distances,

labels, nrof\_folds=nrof\_folds)

thresholds = np.arange(0, 30, 0.001)

val, val\_std, far = calculate\_val(thresholds, distances,

labels, 1e-3, nrof\_folds=nrof\_folds)

return tpr, fpr, accuracy, val, val\_std, far

def calculate\_roc(thresholds, distances, labels, nrof\_folds=10):

nrof\_pairs = min(len(labels), len(distances))

nrof\_thresholds = len(thresholds)

k\_fold = KFold(n\_splits=nrof\_folds, shuffle=False)

tprs = np.zeros((nrof\_folds, nrof\_thresholds))

fprs = np.zeros((nrof\_folds, nrof\_thresholds))

accuracy = np.zeros((nrof\_folds))

indices = np.arange(nrof\_pairs)

for fold\_idx, (train\_set, test\_set) in enumerate(k\_fold.split(indices)):

# Find the best threshold for the fold

acc\_train = np.zeros((nrof\_thresholds))

for threshold\_idx, threshold in enumerate(thresholds):

\_, \_, acc\_train[threshold\_idx] = calculate\_accuracy(threshold, distances[train\_set], labels[train\_set])

best\_threshold\_index = np.argmax(acc\_train)

for threshold\_idx, threshold in enumerate(thresholds):

tprs[fold\_idx, threshold\_idx], fprs[fold\_idx, threshold\_idx], \_ = calculate\_accuracy(threshold,

distances[test\_set],

labels[test\_set])

\_, \_, accuracy[fold\_idx] = calculate\_accuracy(thresholds[best\_threshold\_index], distances[test\_set],

labels[test\_set])

tpr = np.mean(tprs, 0)

fpr = np.mean(fprs, 0)

return tpr, fpr, accuracy

def calculate\_accuracy(threshold, dist, actual\_issame):

predict\_issame = np.less(dist, threshold)

tp = np.sum(np.logical\_and(predict\_issame, actual\_issame))

fp = np.sum(np.logical\_and(predict\_issame, np.logical\_not(actual\_issame)))

tn = np.sum(np.logical\_and(np.logical\_not(predict\_issame), np.logical\_not(actual\_issame)))

fn = np.sum(np.logical\_and(np.logical\_not(predict\_issame), actual\_issame))

tpr = 0 if (tp + fn == 0) else float(tp) / float(tp + fn)

fpr = 0 if (fp + tn == 0) else float(fp) / float(fp + tn)

acc = float(tp + tn) / dist.size

return tpr, fpr, acc

def calculate\_val(thresholds, distances, labels, far\_target=1e-3, nrof\_folds=10):

nrof\_pairs = min(len(labels), len(distances))

nrof\_thresholds = len(thresholds)

k\_fold = KFold(n\_splits=nrof\_folds, shuffle=False)

val = np.zeros(nrof\_folds)

far = np.zeros(nrof\_folds)

indices = np.arange(nrof\_pairs)

for fold\_idx, (train\_set, test\_set) in enumerate(k\_fold.split(indices)):

# Find the threshold that gives FAR = far\_target

far\_train = np.zeros(nrof\_thresholds)

for threshold\_idx, threshold in enumerate(thresholds):

\_, far\_train[threshold\_idx] = calculate\_val\_far(threshold, distances[train\_set], labels[train\_set])

if np.max(far\_train) >= far\_target:

f = interpolate.interp1d(far\_train, thresholds, kind='slinear')

threshold = f(far\_target)

else:

threshold = 0.0

val[fold\_idx], far[fold\_idx] = calculate\_val\_far(threshold, distances[test\_set], labels[test\_set])

val\_mean = np.mean(val)

far\_mean = np.mean(far)

val\_std = np.std(val)

return val\_mean, val\_std, far\_mean

def calculate\_val\_far(threshold, dist, actual\_issame):

predict\_issame = np.less(dist, threshold)

true\_accept = np.sum(np.logical\_and(predict\_issame, actual\_issame))

false\_accept = np.sum(np.logical\_and(predict\_issame, np.logical\_not(actual\_issame)))

n\_same = np.sum(actual\_issame)

n\_diff = np.sum(np.logical\_not(actual\_issame))

if n\_diff == 0:

n\_diff = 1

if n\_same == 0:

return 0, 0

val = float(true\_accept) / float(n\_same)

far = float(false\_accept) / float(n\_diff)

return val, far

def plot\_roc(fpr, tpr, figure\_name="roc.png"):

import matplotlib.pyplot as plt

plt.switch\_backend('Agg')

from sklearn.metrics import auc

roc\_auc = auc(fpr, tpr)

fig = plt.figure()

lw = 2

plt.plot(fpr, tpr, color='#16a085',

lw=lw, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.plot([0, 1], [0, 1], color='#2c3e50', lw=lw, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc="lower right", frameon=False)

fig.savefig(figure\_name, dpi=fig.dpi)