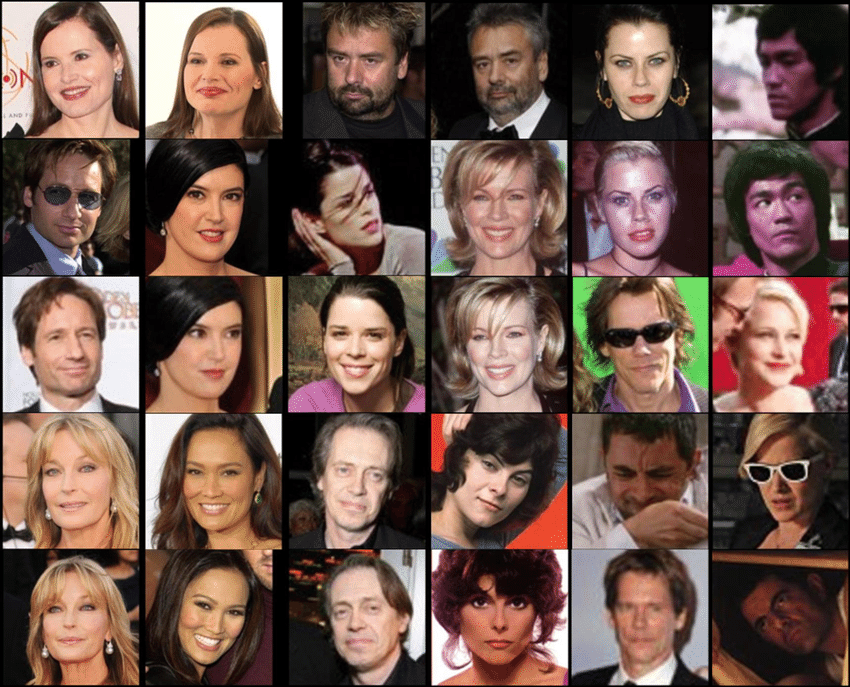
**Hướng dẫn huấn luyện mô hình MobileFacenet**

1. Chuẩn bị dữ liệu

Chúng ta sử dụng tập dữ liệu CASIA WebFace chứa 494.414 hình ảnh khuôn mặt của 10.575 người được thu thập từ web. Bấm vào đường dẫn bên dưới để tải về.

<https://drive.google.com/file/d/1KxNCrXzln0lal3N4JiYl9cFOIhT78y1l/view>



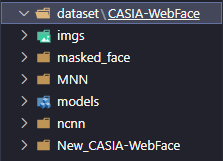
Hình 1. 1 Một vài hình ảnh trong tập dữ liệu CASIA WebFace

Trước khi sử dụng bộ dữ liệu ta cần xử lý bộ dữ liệu. Ta đưa toàn bộ ảnh trong bộ dữ liệu qua một mô hình Face Detection để thu được ảnh chỉ chứa khuôn mặt mà không bao gồm background. Ở đây chúng ta sử dụng mô hình Ultra-lightweight face detection, đây là một mô hình Face Detection rất nhẹ nhưng có độ chính xác cao, tốc độ inference nhanh nên ta sẽ sử dụng để giảm thời gian xử lý dữ liệu

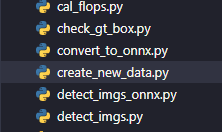
Đầu tiên vào repo <https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB> và tải repo về sau đó giải nén file đã tải về.



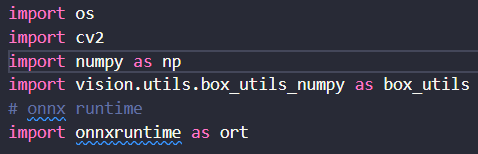
Trong folder đã giải nén tạo một folder tên dataset sau đó copy folder CASIA-WebFace vào, tạo một folder tên New\_CASIA-WebFace để lưu dữ liệu sau khi xử lý.



Tạo file create\_new\_data.py



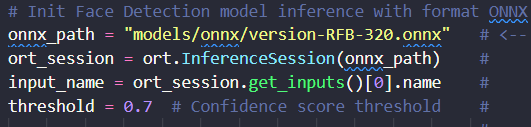
Trong file vừa tạo import những thư viện sau



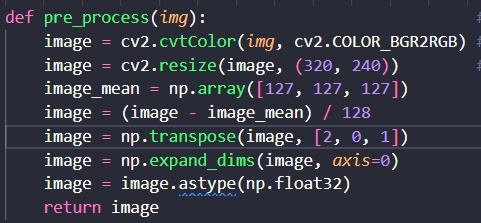
Tiếp theo khởi tạo hai đường dẫn một là đường dẫn tới dataset hiện tại, một là đường dẫn mới chứa dataset sau khi xử lý.



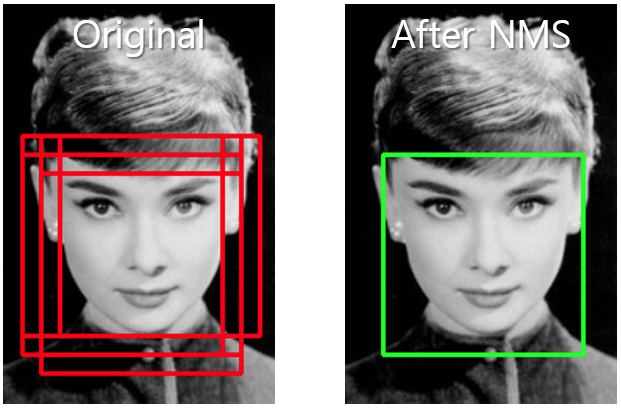
Khởi tạo mô hình



Tạo hàm tiền xử lý ảnh trước khi đưa vào mô hình. Đầu tiên chuyển ảnh được đọc từ OpenCV sẽ chuyển sang hệ màu BGR do đó ta phải chuyển lại sang hệ màu RGB để đúng với ảnh gốc. Tiếp theo thay đổi kích thước ảnh theo mô hình mà ta sử dụng, ở trên ta dùng mô hình RFB-320 do đó ta thay đổi kích thước ảnh về 320-240 (nếu dùng mô hình RFB-640 thì thay đổi thành 640-480). Để đưa được ảnh vào mô hình ta cần chuyển shape của ảnh về định dạng (batch size, chanel, height, weight).

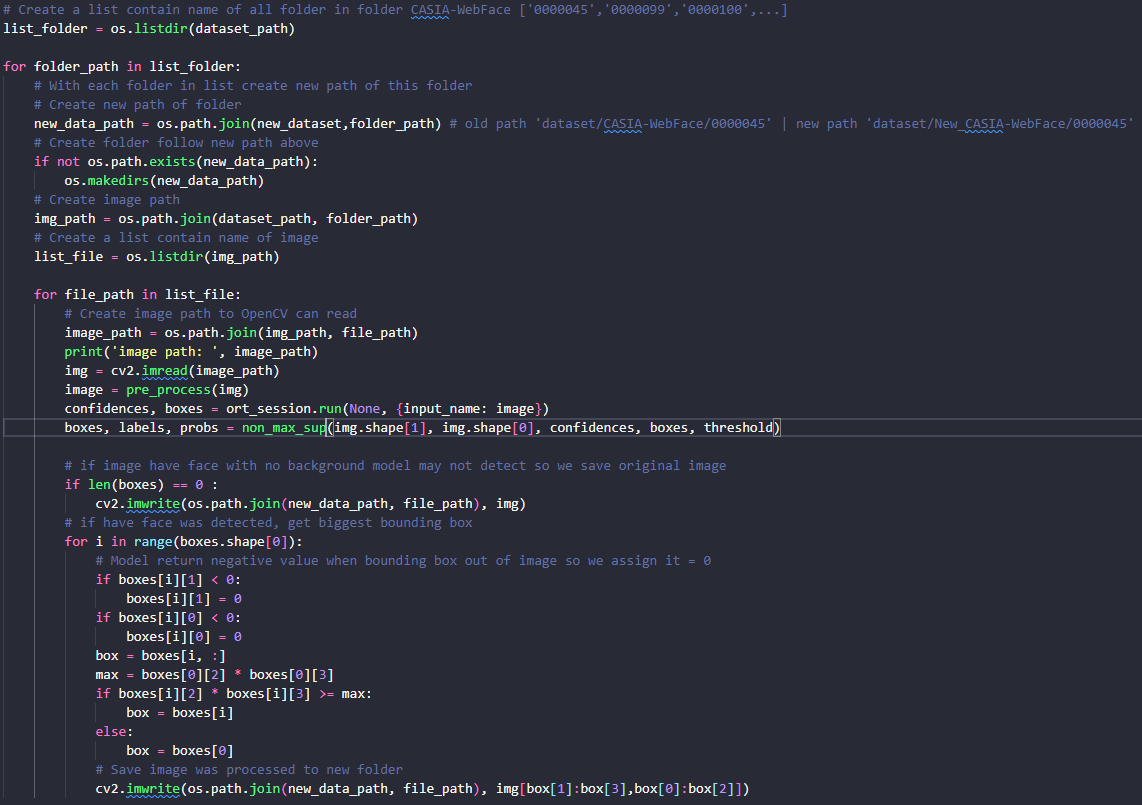


Tạo hàm non-maximum suppression có chức năng loại bỏ những bounding box dư thừa.





Cuối cùng ta đưa lần lượt ảnh trong từng folder trong dataset vào mô hình



Kết quả thu được như hình dưới đây



1. Huấn luyện mô hình

Giống như bài toán phân loại, ta không thể dùng tên của từng người để làm nhãn cho việc huấn luyện được. Có tất cả 10 575 người khác nhau trong tập dữ liệu, do đó, ta gán nhãn cho mỗi người từ 0 🡪 10 574. Ta lập trình đoạn code create\_csv\_file.py để tạo file data\_train.csv chứa đường dẫn đến hình ảnh khuôn và đồng thời gán nhãn cho khuôn mặt đó, như sau:

import csv

import os

root = 'dataset/New-CASIA-WebFace/'

folder\_name\_list = os.listdir(root)

folder\_name\_list.sort()

print('number of classes (or nuber of person ID): ', len(folder\_name\_list))

with open('data\_train.csv', 'a') as csvfile:

    writer = csv.writer(csvfile)

    for i, folder\_name in enumerate(folder\_name\_list):

        folder\_path = root + folder\_name

        img\_name\_list = os.listdir(folder\_path)

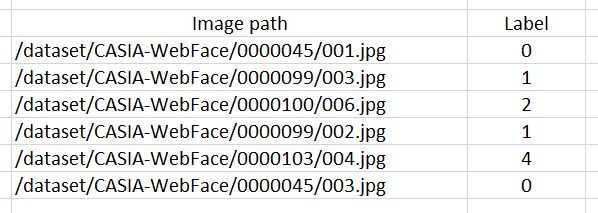
        for img\_name in img\_name\_list:

            img\_path = os.path.join(root, folder\_name, img\_name)

            label = i

            writer.writerow([img\_path, label])

File data\_train.csv được tạo ra có cấu trúc như sau:



Tiếp đến, lập trình file CASIA\_Face\_loader.py để đọc hình ảnh khuôn mặt và nhãn tương ứng từ file data\_train.csv như trong phần 1 đã làm. Ta sử dụng thư viện pandas để đọc địa chỉ hình ảnh và nhãn từ file data\_train.csv và sử dụng thư viện opencv để đọc hình ảnh lên. Lưu ý, một hình ảnh thường có 3 kênh màu theo thứ tự là R, G và B. Tuy nhiên, lệnh cv2.imread (của opencv) khi đọc hình ảnh lên sẽ lưu hình ảnh với thứ tự 3 kênh màu bị đảo ngược. Cụ thể là ảnh màu RGB khi opencv đọc lên sẽ được lưu dưới dạng BGR. Do đó, để ảnh đầu vào của mô hình là ảnh màu có 3 kênh theo thứ tự RGB, ta dùng lệnh cv2.cvtColor(img, cv2.COLOR\_BGR2RGB) để chuyển đổi.

Hình ảnh khuôn mặt trong tập dữ liệu CASIA-WebFace có thể có những hình ảnh có kích thước khác nhau, do đó, để đồng bộ kích thước hình ảnh đầu vào trong quá trình huấn luyện, ta chọn một kích thước duy nhất là 112x112 để làm kích thước chuẩn. Ta sử phép tăng cường dữ liệu quen thuộc đó là phép lật (flip). Chuẩn hóa dữ liệu hình ảnh từ phạm vi giá trị [0 255] về [0 1]. File CASIA\_Face\_loader.py được lập trình như sau:

**CASIA\_Face\_loader.py**

import numpy as np

import os

import torch

import pandas as pd

import cv2

class CASIA\_Face(object):

    def \_\_init\_\_(*self*, *root\_path*='./data\_train.csv', *image\_size*=(112, 112)):

*self*.data\_path = pd.read\_csv(*root\_path*)

*self*.num\_images = len(*self*.data\_path)

*self*.image\_size = *image\_size*

    def \_\_getitem\_\_(*self*, *index*):

        # read image path from cvs file

        image\_path = os.path.join(*self*.data\_path.iloc[*index*, 0])

        # read image from image path

        img = cv2.imread(image\_path)

        # Convert image from BGR to RGB

        img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

        # Resize all image into same size 112x112

        img = cv2.resize(img, *self*.image\_size)

         # read label to train cross\_entropy loss

        label\_cross = *self*.data\_path.iloc[*index*, 1]

        # if image is gray image --> stack gray image 3 times

        # make gray image become 3 channels

        if len(img.shape) == 2:

            img = np.stack([img] \* 3, 2)

        # Augmentation using flip technique

        flip = np.random.choice(2)\*2-1

        img = img[:, ::flip, :]

        # Normalize data from range [0 255] into range [0 1]

        img = img/255

        # Convert image from format [h, w, channel] into format [channel, h, w]

        img = img.transpose(2, 0, 1)

        # Convert image from array to torch

        img = torch.from\_numpy(img).float()

        return img, torch.from\_numpy(np.array(label\_cross, *dtype*=np.long))

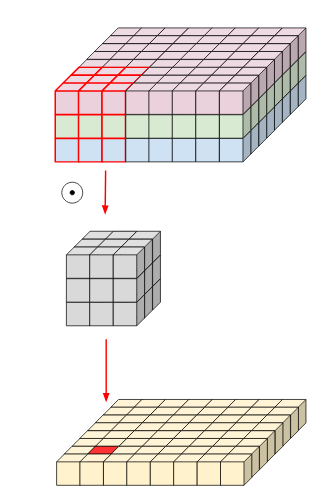
    def \_\_len\_\_(*self*):

        return *self*.num\_images

Chúng ta sẽ dùng mô hình MobileFaceNet [1] cho bài toán nhận diện này. Ý tưởng chính của mô hình này là sử dụng lớp “Depth-wise Convolution” (DW Conv) thay thế những lớp “Convolution” (Conv) thông thường chúng ta được biết trước đây. Với cùng một kernel 3x3, DW Conv tính toán nhanh gấp 9 lần Conv thông thường. Do đó, chúng thường được triển khai trên các thiết bị nhỏ như di động, hoặc thiết bị nhúng.

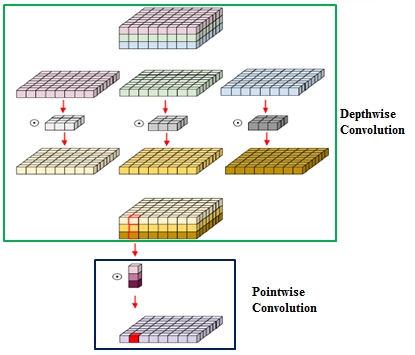
Cụ thể một lớp chập thông thường được biểu diễn ở hình bên dưới, ở đây, ta giả sử có một hình ảnh màu có 3 kênh (channel\_in = 3), và có kích thước là 8x8. Nếu sử dụng một lớp chập có kích thước kernel 3x3 để tạo ngõ ra chỉ có 1 kênh (channel\_out = 1) thì cần dùng một số lượng weights là 3 \*3 \* channel\_in \* channel\_out = 3\*3\*3\*1 = 27. Tương tự, nếu muốn tạo ngõ ra có 3 kênh (channel\_out = 3) thì ta cần 3\*3\*3\*3 = 81.

Còn nếu dụng “Depth-wise Convolution” thì sẽ như thế nào? Ta cùng tìm hiểu một ví dụ tương tự như sau (biểu diễn như hình bên dưới). Ta cũng sử dụng một hình ảnh ngõ vào có 3 kênh và kích thước là 8x8. Ta tách từng kênh ở hình ảnh ngõ vào sau đó nhân chập từng kênh với kernel 3x3. Xếp chồng từng kênh ngõ ra lại với nhau theo thứ tự giống với ngõ vào, ta sẽ thu được hình ảnh ngõ ra có 3 kênh. Tổng số lượng weight cần dùng cho DW Conv để tạo ngõ ra 3 kênh là 3x3x3 = 27 < 81. Số lượng weight nhỏ hơn, sẽ làm giảm kích thước mô hình, đồng thời tăng tốc độ tính toán sẽ giúp mô hình chạy nhanh hơn.



Hình: Sơ đồ biểu diễn lớp chập thông thường (normal convolution) với kích thước kernel 3x3

Tuy nhiên, DW Conv tạo ngõ ra có số lượng kênh đúng bằng ngõ vào. Do đó, muốn điểu chỉnh số lượng kênh ngõ ra theo ý muốn, thông thường ta sẽ cho ngõ ra của lớp DW Conv đi qua một lớp chập thông thường có kích thước kernel = 1x1 (được gọi là “Point-wise Convolution”) để điều chỉnh số lượng kênh ngõ ra. Lớp chập có kích thước kernel =1x1 có số lượng weight khá nhỏ nên thường được sử dụng trong những trường hợp như thế này mà không làm mô hình trở nên quá nặng. Kết hợp DW Conv và Point-wise Convolution, ta gọi là “Depth-wise Separable Convolution”.



Hình : Sơ đồ biểu diễn minh họa “Depth-wise Separable Convolution”

Các bạn có thể tham khảo [2], [3] để biết thêm về lớp “Depth-wise Separable Convolution” này nhé.

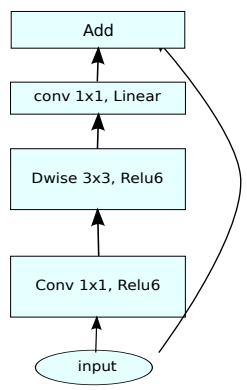
Trong thư viện pytorch, để tách riêng lẻ cách kênh ở ngõ vào cho việc DW Conv, ta sử dụng tham số *groups* trong hàm *nn.Conv2d.* Cụ thể là, một lớp chập thông thường được định nghĩa trong pytorch như sau:

self.conv = Conv2d(channel\_in, channel\_out, kernel\_size=kernel, **groups**=groups, stride=stride, padding=padding, bias=False)

Giá trị mặc định của tham số groups = 1, tương ứng với lớp chập thông thường. Để tách riêng lẻ các kênh ở ngõ vào, ta chỉnh sửa giá trị của tham số groups. Như ví dụ phía trên, ta muốn tách hình ảnh ở ngõ vào làm 3 kênh riêng biệt, ta gán giá trị groups = 3. Để khởi tạo DW Conv như ở ví dụ trên, ta làm như sau:

self.conv = Conv2d(3, 3, kernel\_size=3, **groups**=3, stride=1, padding=1, bias=False)

MobileFaceNet được sử dụng trong ứng dụng này sử dụng ý tưởng kết hợp DW Conv và khối resnet cở bản ở phần 2. Chi tiết được thể hiện như hình bên dưới:



Hình : Khối cơ bản trong mô hình MobileFaceNet, được tạo ra bằng sự kết hợp ý tưởng DW Conv và ResNet.

Đầu tiên ta khởi tạo một vài khối cơ bản cho mô hình:

# Deifine some necessary function or block

def l2\_norm(*input*, *axis* = 1):

    norm = torch.norm(*input*, 2, *axis*, True)

    output = torch.div(*input*, norm)

    return output

class Flatten(Module):

    def forward(*self*, *input*):

        return *input*.view(*input*.size(0), -1)

class Conv\_block(Module):

    def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0), *groups*=1):

        super(Conv\_block, *self*).\_\_init\_\_()

*self*.conv = Conv2d(*in\_c*, *out\_channels*=*out\_c*, *kernel\_size*=*kernel*, *groups*=*groups*, *stride*=*stride*, *padding*=*padding*, *bias*=False)

*self*.bn = BatchNorm2d(*out\_c*)

*self*.prelu = PReLU(*out\_c*)

    def forward(*self*, *x*):

*x* = *self*.conv(*x*)

*x* = *self*.bn(*x*)

*x* = *self*.prelu(*x*)

        return *x*

class Linear\_block(Module):

    def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0), *groups*=1):

        super(Linear\_block, *self*).\_\_init\_\_()

*self*.conv = Conv2d(*in\_c*, *out\_channels*=*out\_c*, *kernel\_size*=*kernel*, *groups*=*groups*, *stride*=*stride*, *padding*=*padding*, *bias*=False)

*self*.bn = BatchNorm2d(*out\_c*)

    def forward(*self*, *x*):

*x* = *self*.conv(*x*)

*x* = *self*.bn(*x*)

        return *x*

Trong thư viện pytorch, ta lập trình khối cơ bản này như sau (class Depth\_Wise\_Res):

class Depth\_Wise\_Res(Module):

     def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *residual* = False, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=1):

        super(Depth\_Wise\_Res, *self*).\_\_init\_\_()

*self*.conv = Conv\_block(*in\_c*, *out\_c*=*groups*, *kernel*=(1, 1), *padding*=(0, 0), *stride*=(1, 1))

*self*.conv\_dw = Conv\_block(*groups*, *groups*, *groups*=*groups*, *kernel*=*kernel*, *padding*=*padding*, *stride*=*stride*)

*self*.project = Linear\_block(*groups*, *out\_c*, *kernel*=(1, 1), *padding*=(0, 0), *stride*=(1, 1))

*self*.residual = *residual*

     def forward(*self*, *x*):

        if *self*.residual:

            short\_cut = *x*

*x* = *self*.conv(*x*)

*x* = *self*.conv\_dw(*x*)

*x* = *self*.project(*x*)

        if *self*.residual:

            output = short\_cut + *x*

        else:

            output = *x*

        return output

MobileFacenet sử dụng rất nhiều khối cơ bản Depth\_Wise\_Res, để code được gọn gàng hơn, ta nhóm các khối cơ bản Depth\_Wise\_Res có chung kích thước ngõ vào và ngõ ra lại với nhau. Ta định nghĩa nhiều khối Depth\_Wise\_Res trong một class Multi\_Res như sau:

class Multi\_Res(Module):

    def \_\_init\_\_(*self*, *c*, *num\_block*, *groups*, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1)):

        super(Multi\_Res, *self*).\_\_init\_\_()

        modules = []

        for \_ in range(*num\_block*):

            modules.append(Depth\_Wise\_Res(*c*, *c*, *residual*=True, *kernel*=*kernel*, *padding*=*padding*, *stride*=*stride*, *groups*=*groups*))

*self*.model = Sequential(\*modules)

    def forward(*self*, *x*):

        return *self*.model(*x*)

Từ đó, mô hình MobileFaceNet được lập trình như sau:

class MobileFaceNet(Module):

    def \_\_init\_\_(*self*, *embedding\_size*=512, *class\_num*=1):

        super(MobileFaceNet, *self*).\_\_init\_\_()

*self*.conv1 = Conv\_block(3, 64, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1))

*self*.conv2\_dw = Conv\_block(64, 64, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1), *groups*=64)

*self*.conv\_23 = Depth\_Wise\_Res(64, 64, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=128)

*self*.conv\_3 = Multi\_Res(64, *num\_block*=4, *groups*=128, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_34 = Depth\_Wise\_Res(64, 128, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=256)

*self*.conv\_4 = Multi\_Res(128, *num\_block*=6, *groups*=256, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_45 = Depth\_Wise\_Res(128, 128, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=512)

*self*.conv\_5 = Multi\_Res(128, *num\_block*=2, *groups*=256, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_6\_sep = Conv\_block(128, 512, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0))

*self*.conv\_6\_dw = Linear\_block(512, 512, *groups*=512, *kernel*=(7,7), *stride*=(1, 1), *padding*=(0, 0))

*self*.conv\_6\_flatten = Flatten()

*self*.linear = Linear(512, *embedding\_size*, *bias*=False)

    def forward(*self*, *x*):

        out = *self*.conv1(*x*)

        out = *self*.conv2\_dw(out)

        out = *self*.conv\_23(out)

        out = *self*.conv\_3(out)

        out = *self*.conv\_34(out)

        out = *self*.conv\_4(out)

        out = *self*.conv\_45(out)

        out = *self*.conv\_5(out)

        out = *self*.conv\_6\_sep(out)

        out = *self*.conv\_6\_dw(out)

        out = *self*.conv\_6\_flatten(out)

        out = *self*.linear(out)

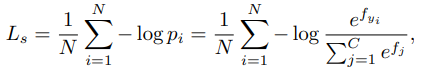
        return out

Trong giai đoạn huấn luyện, bài toán này là bài toán phân loại khuôn mặt người, số lượng ngõ ra của mô hình phải đúng bằng số lượng lớp (số lượng người) trong tập dữ liệu CASIA-WebFace. Do số lượng người cần phân loại trong tập dữ liệu là 10 575 nên số lượng ngõ ra phải là 10 575. Ta cần phải thiết kế thêm một kiến trúc mở rộng để thay đổi kích thước ngõ ra đúng bằng 10 575 cho việc phân loại. Đơn giản nhất kiến trúc mở rộng chỉ cần 1 lớp kết nối đủ như sau:

self.linear = Linear(512, 10575, bias=False)

# Trong đó, 512 là số lượng ngõ ra của mô hình MobleFaceNet

Tuy nhiên, để mô hình có thể học “embedding features” tốt hơn (nghĩa là “embedding features” của khuôn mặt này khác biệt với “embedding features” của khuôn mặt khác), ta áp dụng một kĩ thuật có tên là CosFace [4]. Ý tưởng cơ bản của CosFace là chuyển đổi đặc trưng (embedding features) từ miền vector sang miền góc để tính mất mát (loss).

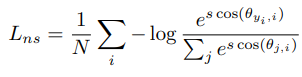
Để hiểu một cách chi tiết hơn, ta quay lại với bài toán phân loại như trong phần 1, ta dùng hàm mất mát Softmax loss (một dạng của Cross Entropy loss) để học các đặc trưng tương ứng với những lớp khác nhau. Giả sử ta có một vector đặc trưng là xi, và nhãn tương ứng là yi, ta có softmax loss được định nghĩa như sau:

Trong đó, pi là xác xuất xi được phân loại đúng với nhãn. N là số lượng mẫu được huấn luyện (trong một batch). fi là ngõ ra của lớp kết nối đủ cuối cùng của mô hình. fi được tạo ra như sau:

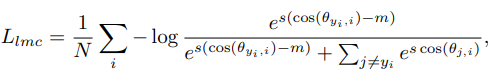


Trong đó, Wj là weights của lớp kết nối đủ cuối cùng

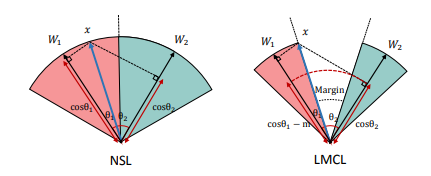
Ta cố định độ dài vector ||Wj|| = 1, và ||x|| = s (s là hằng số). Như vậy, việc học weight ở lớp kết nối đủ cuối cùng được chuyển từ miền vector sang miền góc. Thay vì huấn luyện để mô hình học giá trị W, giờ đây, ta huấn luyện mô hình để học giá trị góc *theta*. Hàm mất mát được tính lại như sau:



Để các đặc trưng của những khuôn mặt khác nhau được tách biệt một cách rõ ràng hơn, ta sử dụng một kĩ thuật được gọi là “margin”. Đơn giản là ta thêm giá trị “margin” vào hàm mất mát để ràng buộc mô hình học “embedding features” của các lớp (class) khác nhau phải cách nhau một khoảng “margin”. Thêm giá trị “margin” vào hàm mất mát như sau:



Để hiểu rõ hơn về ý nghĩa của việc thêm “margin”, hãy xem hình bên dưới:



Hình : Bên trái, đặc trưng được học ở miền góc của 2 “embedding features” nếu không dùng kĩ thuật “margin”. Ngược lại, ở bên phải có sử dụng kĩ thuật “margin”. Ta nhận thấy, với việc áp dụng kĩ thuật “margin”, đặc trưng của những lớp khác nhau sẽ tách rời nhau. Điều này tốt cho việc phân loại.

Việc lập trình kiến trúc mở rộng CosFace bằng thư viện pytorch được thể hiện như sau:

class ArcMarginProduct(*nn*.Module):

    def \_\_init\_\_(*self*, *in\_features*=512, *out\_features*=10575, *s*=32.0, *m*=0.50):

        super(ArcMarginProduct, *self*).\_\_init\_\_()

*self*.in\_features = *in\_features*

*self*.out\_features = *out\_features*

*self*.s = *s*  # gia tri ||s||

*self*.m = *m* # gia tri margin

*self*.weight = Parameter(torch.Tensor(*out\_features*, *in\_features*))

        nn.init.xavier\_uniform\_(*self*.weight)

*self*.cos\_m = math.cos(*m*)

*self*.sin\_m = math.sin(*m*)

        # make the function cos(theta+m) monotonic decreasing while theta in [0°,180°]

*self*.th = math.cos(math.pi - *m*)

*self*.mm = math.sin(math.pi - *m*) \* *m*

    def forward(*self*, *x*, *label*):

        cosine = F.linear(F.normalize(*x*), F.normalize(*self*.weight))

        sine = torch.sqrt(1.0 - torch.pow(cosine, 2))

        phi = cosine \* *self*.cos\_m - sine \* *self*.sin\_m

        phi = torch.where((cosine - *self*.th) > 0, phi, cosine - *self*.mm)

        one\_hot = torch.zeros(cosine.size(), *device*='cuda')

        one\_hot.scatter\_(1, *label*.view(-1, 1).long(), 1)

        output = (one\_hot \* phi) + ((1.0 - one\_hot) \* cosine)

        output \*= *self*.s

        return output

Tổng kết tất cả những đoạn code thiết kế mô hình huấn luyện trong một file model.py duy nhất, ta được kết quả như sau:

# Import some necessary libraries

from torch import nn

import torch

import torch.nn.functional as F

import math

from torch.nn import Parameter

from torch.nn import Linear, Conv2d, BatchNorm2d, PReLU, Sequential, Module, Parameter

# Deifine some necessary function or block

def l2\_norm(*input*, *axis* = 1):

    norm = torch.norm(*input*, 2, *axis*, True)

    output = torch.div(*input*, norm)

    return output

class Flatten(Module):

    def forward(*self*, *input*):

        return *input*.view(*input*.size(0), -1)

class Conv\_block(Module):

    def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0), *groups*=1):

        super(Conv\_block, *self*).\_\_init\_\_()

*self*.conv = Conv2d(*in\_c*, *out\_channels*=*out\_c*, *kernel\_size*=*kernel*, *groups*=*groups*, *stride*=*stride*, *padding*=*padding*, *bias*=False)

*self*.bn = BatchNorm2d(*out\_c*)

*self*.prelu = PReLU(*out\_c*)

    def forward(*self*, *x*):

*x* = *self*.conv(*x*)

*x* = *self*.bn(*x*)

*x* = *self*.prelu(*x*)

        return *x*

class Linear\_block(Module):

    def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0), *groups*=1):

        super(Linear\_block, *self*).\_\_init\_\_()

*self*.conv = Conv2d(*in\_c*, *out\_channels*=*out\_c*, *kernel\_size*=*kernel*, *groups*=*groups*, *stride*=*stride*, *padding*=*padding*, *bias*=False)

*self*.bn = BatchNorm2d(*out\_c*)

    def forward(*self*, *x*):

*x* = *self*.conv(*x*)

*x* = *self*.bn(*x*)

        return *x*

# Basic block of MobileFacenet, combining the idea from DW Conv and ResNet

class Depth\_Wise\_Res(Module):

    def \_\_init\_\_(*self*, *in\_c*, *out\_c*, *residual* = False, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=1):

        super(Depth\_Wise\_Res, *self*).\_\_init\_\_()

*self*.conv = Conv\_block(*in\_c*, *out\_c*=*groups*, *kernel*=(1, 1), *padding*=(0, 0), *stride*=(1, 1))

*self*.conv\_dw = Conv\_block(*groups*, *groups*, *groups*=*groups*, *kernel*=*kernel*, *padding*=*padding*, *stride*=*stride*)

*self*.project = Linear\_block(*groups*, *out\_c*, *kernel*=(1, 1), *padding*=(0, 0), *stride*=(1, 1))

*self*.residual = *residual*

    def forward(*self*, *x*):

        if *self*.residual:

            short\_cut = *x*

*x* = *self*.conv(*x*)

*x* = *self*.conv\_dw(*x*)

*x* = *self*.project(*x*)

        if *self*.residual:

            output = short\_cut + *x*

        else:

            output = *x*

        return output

class Multi\_Res(Module):

    def \_\_init\_\_(*self*, *c*, *num\_block*, *groups*, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1)):

        super(Multi\_Res, *self*).\_\_init\_\_()

        modules = []

        for \_ in range(*num\_block*):

            modules.append(Depth\_Wise\_Res(*c*, *c*, *residual*=True, *kernel*=*kernel*, *padding*=*padding*, *stride*=*stride*, *groups*=*groups*))

*self*.model = Sequential(\*modules)

    def forward(*self*, *x*):

        return *self*.model(*x*)

class MobileFaceNet(Module):

    def \_\_init\_\_(*self*, *embedding\_size*=512, *class\_num*=1):

        super(MobileFaceNet, *self*).\_\_init\_\_()

*self*.conv1 = Conv\_block(3, 64, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1))

*self*.conv2\_dw = Conv\_block(64, 64, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1), *groups*=64)

*self*.conv\_23 = Depth\_Wise\_Res(64, 64, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=128)

*self*.conv\_3 = Multi\_Res(64, *num\_block*=4, *groups*=128, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_34 = Depth\_Wise\_Res(64, 128, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=256)

*self*.conv\_4 = Multi\_Res(128, *num\_block*=6, *groups*=256, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_45 = Depth\_Wise\_Res(128, 128, *kernel*=(3, 3), *stride*=(2, 2), *padding*=(1, 1), *groups*=512)

*self*.conv\_5 = Multi\_Res(128, *num\_block*=2, *groups*=256, *kernel*=(3, 3), *stride*=(1, 1), *padding*=(1, 1))

*self*.conv\_6\_sep = Conv\_block(128, 512, *kernel*=(1, 1), *stride*=(1, 1), *padding*=(0, 0))

*self*.conv\_6\_dw = Linear\_block(512, 512, *groups*=512, *kernel*=(7,7), *stride*=(1, 1), *padding*=(0, 0))

       # self.adaptivePooling = AdaptiveAvgPool2d((1,1))

*self*.conv\_6\_flatten = Flatten()

*self*.linear = Linear(512, *embedding\_size*, *bias*=False)

    def forward(*self*, *x*):

        out = *self*.conv1(*x*)

        out = *self*.conv2\_dw(out)

        out = *self*.conv\_23(out)

        out = *self*.conv\_3(out)

        out = *self*.conv\_34(out)

        out = *self*.conv\_4(out)

        out = *self*.conv\_45(out)

        out = *self*.conv\_5(out)

        out = *self*.conv\_6\_sep(out)

        out = *self*.conv\_6\_dw(out)

        out = *self*.conv\_6\_flatten(out)

        out = *self*.linear(out)

        return out

class ArcMarginProduct(*nn*.Module):

    def \_\_init\_\_(*self*, *in\_features*=512, *out\_features*=10575, *s*=32.0, *m*=0.50):

        super(ArcMarginProduct, *self*).\_\_init\_\_()

*self*.in\_features = *in\_features*

*self*.out\_features = *out\_features*

*self*.s = *s*  # gia tri ||s||

*self*.m = *m* # gia tri margin

*self*.weight = Parameter(torch.Tensor(*out\_features*, *in\_features*))

        nn.init.xavier\_uniform\_(*self*.weight)

*self*.cos\_m = math.cos(*m*)

*self*.sin\_m = math.sin(*m*)

        # make the function cos(theta+m) monotonic decreasing while theta in [0°,180°]

*self*.th = math.cos(math.pi - *m*)

*self*.mm = math.sin(math.pi - *m*) \* *m*

    def forward(*self*, *x*, *label*):

        cosine = F.linear(F.normalize(*x*), F.normalize(*self*.weight))

        sine = torch.sqrt(1.0 - torch.pow(cosine, 2))

        phi = cosine \* *self*.cos\_m - sine \* *self*.sin\_m

        phi = torch.where((cosine - *self*.th) > 0, phi, cosine - *self*.mm)

        one\_hot = torch.zeros(cosine.size(), *device*='cuda')

        one\_hot.scatter\_(1, *label*.view(-1, 1).long(), 1)

        output = (one\_hot \* phi) + ((1.0 - one\_hot) \* cosine)

        output \*= *self*.s

        return output

Để dễ dàng quản lí các file trong quá trình lập trình, và giúp code trở nên gọn gàng hơn, ta định nghĩa tất cả các siêu tham số trong một file, có tên là config.py như sau:

# define some hyper-parameter and some configurations

BATCH\_SIZE = 512

TOTAL\_EPOCH = 500

IMG\_SIZE = (112, 112)

SAVE\_DIR = './saved\_models'

NUM\_CLASSES = 10575

SAVED\_EPOCH = 5 # save model's weight every 5 epochs

CASIA\_DATA\_DIR = './data\_train.csv'

LEARNING\_RATE = 0.1

Ta lập trình việc huấn luyện mô hình trong file train.py. Đầu tiên, ta khởi tạo các thư viện cần thiết, đồng thời nhập (import) các lớp và các tham số đã được định nghĩa ở những file khác vào file train.py như sau:

import os

import torch.utils.data

from torch import nn

# import hyper-parameters from file config

from config import BATCH\_SIZE, SAVE\_DIR, TOTAL\_EPOCH, IMG\_SIZE, NUM\_CLASSES, LEARNING\_RATE, SAVED\_EPOCH, CASIA\_DATA\_DIR

# import model for training

from model import MobileFaceNet, ArcMarginProduct

# import dataloader to load data and label for training

from CASIA\_Face\_loader import CASIA\_Face

# this function help adjust learning during training

from torch.optim import lr\_scheduler

# other lib

import torch.optim as optim

import time

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

Tiếp theo ta khởi tạo mô hình chính (MobileFaceNet) và kiến trúc mở rộng như sau:

# define model

net = MobileFaceNet()

ArcMargin = ArcMarginProduct(512, NUM\_CLASSES)

net = net.to(device)

ArcMargin = ArcMargin.to(device)

Tiếp đến, ta định nghĩa hàm mất mát và giải thuật tối ưu. Ta sử dụng hàm Cross\_Entropy loss cho bài toán phân loại và giải thuật tối ưu phổ biến là SGD (stochastic gradient descent). Khác với những bài toán trước, tất cả các tham số (weights) của mô hình được huấn luyện với cùng một “weight\_decay”. Trong phần này, ta tách một số “module” trong mô hình và thiết lập “weight\_decay” khác nhau. Ta làm như sau:

# define optimizers

ignored\_params = list(map(id, net.linear.parameters()))

ignored\_params += list(map(id, ArcMargin.weight))

prelu\_params\_id = []

prelu\_params = []

for m in net.modules():

    if isinstance(m, nn.PReLU):

        ignored\_params += list(map(id, m.parameters()))

        prelu\_params += m.parameters()

base\_params = filter(lambda *p*: id(*p*) not in ignored\_params, net.parameters())

optimizer\_ft = optim.SGD([

    {'params': base\_params, 'weight\_decay': 4e-5},

    {'params': net.linear.parameters(), 'weight\_decay': 4e-4},

    {'params': ArcMargin.weight, 'weight\_decay': 4e-4},

    {'params': prelu\_params, 'weight\_decay': 0.0}

], *lr*=LEARNING\_RATE, *momentum*=0.9, *nesterov*=True)

exp\_lr\_scheduler = lr\_scheduler.MultiStepLR(optimizer\_ft, *milestones*=[200000], *gamma*=0.1)

# Using cross-entropy loss for classification human ID

criterion = torch.nn.CrossEntropyLoss()

Để thu được kết quả tốt hơn từ việc huấn luyện mô hình, giống như ứng dụng phát hiện khuôn mặt, ta cũng thực hiện kĩ thuật giảm tốc độ học (learning rate) theo thời gian huấn luyện. Thư viện pytorch có cung cấp hàm để ta thực hiện việc này vô cùng dễ dàng như sau:

exp\_lr\_scheduler = lr\_scheduler.MultiStepLR(optimizer\_ft, *milestones*=[200000], *gamma*=0.1)

Cuối cùng, ta tiến hành lập trình việc huấn luyện mô hình, in ra màn hình giá trị mất mát và lưu lại weights của mô hình:

best\_acc = 0.0

best\_epoch = 0

iters = 0

for epoch in range(0, TOTAL\_EPOCH+1):

    # train model

    print('Train Epoch: {}/{} ...'.format(epoch, TOTAL\_EPOCH))

    net.train()

    train\_total\_loss = 0.0

    total = 0

    since = time.time()

    print('iteration for 1 epoch: ', iters)

    for data in trainloader:

        iters = iters + 1

        img, label = data[0].cuda(), data[1].cuda()

        batch\_size = img.size(0)

        optimizer\_ft.zero\_grad()

        raw\_logits = net(img)

        output = ArcMargin(raw\_logits, label)

        total\_loss = criterion(output, label)

        total\_loss.backward()

        optimizer\_ft.step()

        exp\_lr\_scheduler.step()

        train\_total\_loss += total\_loss.item() \* batch\_size

        total += batch\_size

    train\_total\_loss = train\_total\_loss / total

    time\_elapsed = time.time() - since

    loss\_msg = 'total\_loss: {:.4f} time: {:.0f}m {:.0f}s'\

        .format(train\_total\_loss, time\_elapsed // 60, time\_elapsed % 60)

    print(loss\_msg)

    # save model

    if not os.path.exists(SAVE\_DIR):

        os.mkdir(SAVE\_DIR)

    if epoch % SAVED\_EPOCH == 0:

        torch.save(net.state\_dict(),

            os.path.join(SAVE\_DIR, '%03d.pth' % epoch))

print('finishing training')

Tổng hợp toàn bộ các đoạn code nhỏ ở trên trong một file train.py, ta được như sau:

import os

import torch.utils.data

from torch import nn

# import hyper-parameters from file config

from config import BATCH\_SIZE, SAVE\_DIR, TOTAL\_EPOCH, IMG\_SIZE, NUM\_CLASSES, LEARNING\_RATE, SAVED\_EPOCH, CASIA\_DATA\_DIR

# import model for training

from model import MobileFaceNet, ArcMarginProduct

# import dataloader to load data and label for training

from CASIA\_Face\_loader import CASIA\_Face

# this function help adjust learning during training

from torch.optim import lr\_scheduler

# other lib

import torch.optim as optim

import time

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

# define trainloader and testloader

trainset = CASIA\_Face(*root\_path*=CASIA\_DATA\_DIR, *image\_size* = IMG\_SIZE)

trainloader = torch.utils.data.DataLoader(trainset, *batch\_size*=BATCH\_SIZE,

*shuffle*=True, *num\_workers*=2, *drop\_last*=False)

# define model

net = MobileFaceNet()

ArcMargin = ArcMarginProduct(512, NUM\_CLASSES)

net = net.to(device)

ArcMargin = ArcMargin.to(device)

# define optimizers

ignored\_params = list(map(id, net.linear.parameters()))

ignored\_params += list(map(id, ArcMargin.weight))

prelu\_params\_id = []

prelu\_params = []

for m in net.modules():

    if isinstance(m, nn.PReLU):

        ignored\_params += list(map(id, m.parameters()))

        prelu\_params += m.parameters()

base\_params = filter(lambda *p*: id(*p*) not in ignored\_params, net.parameters())

optimizer\_ft = optim.SGD([

    {'params': base\_params, 'weight\_decay': 4e-5},

    {'params': net.linear.parameters(), 'weight\_decay': 4e-4},

    {'params': ArcMargin.weight, 'weight\_decay': 4e-4},

    {'params': prelu\_params, 'weight\_decay': 0.0}

], *lr*=LEARNING\_RATE, *momentum*=0.9, *nesterov*=True)

exp\_lr\_scheduler = lr\_scheduler.MultiStepLR(optimizer\_ft, *milestones*=[200000], *gamma*=0.1)

# Using cross-entropy loss for classification human ID

criterion = torch.nn.CrossEntropyLoss()

best\_acc = 0.0

best\_epoch = 0

iters = 0

for epoch in range(0, TOTAL\_EPOCH+1):

    # train model

    print('Train Epoch: {}/{} ...'.format(epoch, TOTAL\_EPOCH))

    net.train()

    train\_total\_loss = 0.0

    total = 0

    since = time.time()

    print('iteration for 1 epoch: ', iters)

    for data in trainloader:

        iters = iters + 1

        img, label = data[0].cuda(), data[1].cuda()

        batch\_size = img.size(0)

        optimizer\_ft.zero\_grad()

        raw\_logits = net(img)

        output = ArcMargin(raw\_logits, label)

        total\_loss = criterion(output, label)

        total\_loss.backward()

        optimizer\_ft.step()

        exp\_lr\_scheduler.step()

        train\_total\_loss += total\_loss.item() \* batch\_size

        total += batch\_size

    train\_total\_loss = train\_total\_loss / total

    time\_elapsed = time.time() - since

    loss\_msg = 'total\_loss: {:.4f} time: {:.0f}m {:.0f}s'\

        .format(train\_total\_loss, time\_elapsed // 60, time\_elapsed % 60)

    print(loss\_msg)

    # save model

    if not os.path.exists(SAVE\_DIR):

        os.mkdir(SAVE\_DIR)

    if epoch % SAVED\_EPOCH == 0:

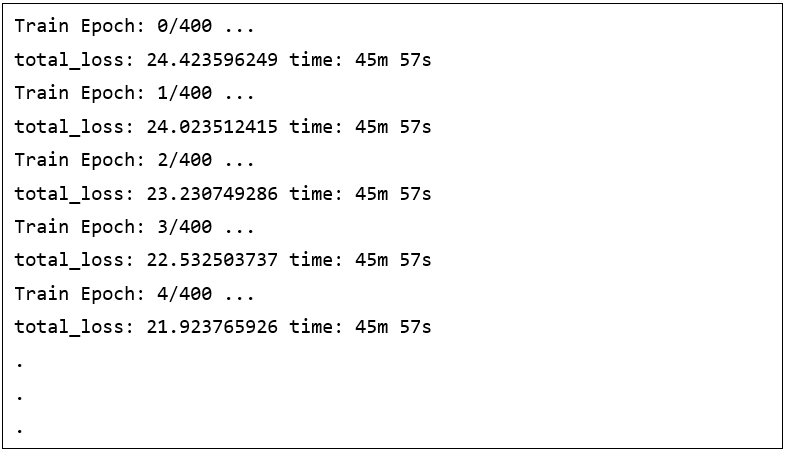
        torch.save(net.state\_dict(),

            os.path.join(SAVE\_DIR, '%03d.pth' % epoch))

print('finishing training')

Để huấn luyện mô hình, trên terminal ta chạy lệnh python train.py

Kết quả in ra màn hình như sau:



1. Kiểm tra mô hình

Quá trình kiểm tra mô hình được biểu diễn như hình phía trên. Ở đây, ta kiểm tra 2 hình ảnh khuôn mặt có phải cùng một người hay không. Ta lần lượt đưa 2 hình ảnh khuôn mặt vào mô hình để tạo ra các “embedding features” tương ứng. Tiếp theo, ta so sánh 2 “embedding features” này, nếu chúng gần giống nhau thì sẽ cùng 1 người và ngược lại.

Đầu tiên, ta cũng khởi tạo những thư viện cần thiết và “import” mô hình MobileFaceNet đã được thiết kế trong file model.py.

import torch

from model import MobileFaceNet

import cv2

import numpy as np

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

Tiếp đến, ta khởi tạo mô hình MobileFaceNet và tải weights đã được huấn luyện vào lại mô hình, như sau:

# init MobileFaceNet model

    net = MobileFaceNet()

    # load trained model from folder ./saved\_model

    net.load\_state\_dict(torch.load('./saved\_models/400.pth'))

    #

    net.eval()

    net = net.to(device)

Ta nạp đường dẫn đến 2 hình ảnh khuôn mặt và đọc 2 hình ảnh đó lên bằng thư viện opencv như sau:

# path to image of two face we want to verify (same or different person)

    face1\_path = './test\_images/Ronaldo\_1.jpg'

    face2\_path = './test\_images/Ronaldo\_2.JPG'

    # read images

    img1 = cv2.imread(face1\_path)

    img2 = cv2.imread(face2\_path)

Những hình ảnh này sau khi đọc lên cần phải được tiền xử lí (pre-processing) trước khi đưa vào mô hình. Việc tiền xử lí hình ảnh ngõ vào cho tiến trình kiểm tra phải giống quá trình tiền xử lí trong tiến trình huấn luyện. Ta lập trình một hàm tiền xử lí hình ảnh ngõ vào như sau:

def processing\_image(*img*):

*img* = cv2.cvtColor(*img*, cv2.COLOR\_BGR2RGB)

*img* = cv2.resize(*img*, (112, 112))

*img* = *img*/255

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

*img* = *img*[None, :]

*img* = torch.from\_numpy(*img*).float()

    return *img*

Khi đưa hình ảnh khuôn mặt vào ngõ vào của mô hình, ta thu được “embedding features” tương ứng của khuôn mặt đó. Trước khi sử dụng các vector “embedding features” này để so sánh, ta phải chuẩn hóa nó. Ý nghĩa của việc chuẩn hóa là để đồng nhất tất cả các vector “embedding features” này về chung một phân phối. Ở đây, ta chọn phương pháp chuẩn hóa Norm-2. Giả sử một vector “embedding features” 512 chiều, gọi là F, có 512 phần tử fi (với i = 1🡪 512). Ta thiết lập công thức chuẩn hóa từng phần tử fi như sau:

Với f’i là phần thử thứ i trong vector “embedding features” sau khi được chuẩn hóa.

Ta lập trình một hàm chuẩn hóa vector theo công thức trên bằng cách sử dụng thư viện *numpy* như sau:

def feature\_normalization(*embedding\_features*):

    normalized\_features = *embedding\_features*/np.linalg.norm(*embedding\_features*)

    return normalized\_features

Cuối cùng, để biết được 2 hình ảnh khuôn mặt có phải cùng một người hay không, ta so sánh 2 vector “emnedding features” tương ứng của 2 khuôn mặt đó. Để so sánh 2 vector, ta dùng phép tính cosine, nếu cosine giữa 2 vector gần bằng 1 thì 2 vector đó gần giống nhau, có nghĩa là 2 khuôn mặt tương ứng là cùng một người. Ngược lại, nếu cosine giữa 2 vector gần bằng 0, thì 2 hình ảnh khuôn mặt ngõ vào là của 2 người khác nhau. Để tính cosine giữa 2 vector, ta lập trình một hàm như sau:

def feature\_comparison(*feature\_1*, *feature\_2*):

    cosine = np.dot(*feature\_1*, *feature\_2*)

    cosine = np.clip(cosine, -1.0, 1.0)

    return cosine

Trong quá trình kiểm tra, ta chọn một giá trị ngưỡng (threshold). Nếu consine lớn hơn giá trị ngưỡng thì ta xem như cosine = 1 (2 hình ảnh khuôn mặt là cùng một người) và ngược lại. Trong phần này, ta chọn threshold = 0.5.

Tổng hợp những đoạn code trên, cùng với việc lập trình thêm vài dòng code để in ra màn hình kết quả so sánh, ta được file test.py như sau:

import torch

from model import MobileFaceNet

import cv2

import numpy as np

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

def feature\_normalization(*embedding\_features*):

    normalized\_features = *embedding\_features*/np.linalg.norm(*embedding\_features*)

    return normalized\_features

def feature\_comparison(*feature\_1*, *feature\_2*):

    cosine = np.dot(*feature\_1*, *feature\_2*)

    cosine = np.clip(cosine, -1.0, 1.0)

    return cosine

def processing\_image(*img*):

*img* = cv2.cvtColor(*img*, cv2.COLOR\_BGR2RGB)

*img* = cv2.resize(*img*, (112, 112))

*img* = *img*/255

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

*img* = *img*[None, :]

*img* = torch.from\_numpy(*img*).float()

    return *img*

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

    threshold = 0.5

    # path to image of two face we want to verify (same or different person)

    face1\_path = './test\_images/Ronaldo\_1.jpg'

    face2\_path = './test\_images/Ronaldo\_2.JPG'

    # read images

    img1 = cv2.imread(face1\_path)

    img2 = cv2.imread(face2\_path)

    # init MobileFaceNet model

    net = MobileFaceNet()

    # load trained model from folder ./saved\_model

    net.load\_state\_dict(torch.load('./saved\_models/400.pth'))

    #

    net.eval()

    net = net.to(device)

    with torch.no\_grad():

        # pre-processing image before input to model

        img1 = processing\_image(img1)

        img2 = processing\_image(img2)

        img1 = img1.to(device)

        img2 = img2.to(device)

        # input face image to model to get embedding features

        embedding\_features1 = net(img1)

        embedding\_features1 = embedding\_features1.squeeze()

        embedding\_features2 = net(img2)

        embedding\_features2 = embedding\_features2.squeeze()

        # convert features from torch to numpy

        embedding\_features1 = embedding\_features1.data.cpu().numpy()

        embedding\_features2 = embedding\_features2.data.cpu().numpy()

        # normalize embedding features

        normalized\_features1 = feature\_normalization(embedding\_features1)

        normalized\_features2 = feature\_normalization(embedding\_features2)

        cosine = feature\_comparason(normalized\_features1, normalized\_features2)

        if cosine > threshold:

            print('Cosin = {}: TWO FACE ARE SAME PERSON'.format(cosine))

        else:

            print('Cosin = {}: TWO FACE ARE DIFFERENT PERSON'.format(cosine))

Ta thử test 2 hình ảnh khuôn mặt đưới đây và xem kết quả in ra màn hình:



Hình: Kiểm tra 2 hình ảnh cùng một người

Ta thu được kết quả như bên dưới:

|  |
| --- |
| $ python test.py |
| Cosin = 0.5976020097732544: TWO FACE ARE SAME PERSON |

Ta kiểm tra thử 2 hình ảnh khuôn mặt khác nhau:



Hình: Kiểm tra 2 hình ảnh của 2 khuôn mặt khác nhau

Ta thu được kết quả như bên dưới:

|  |
| --- |
| $ python test.py |
| Cosin = 0.034748874604701996: TWO FACE ARE DIFFERENT PERSON |

Thế là ta đã cơ bản xây dựng thành công một ứng dụng nhận dạng khuôn mặt. Để xây dựng một bài toán tổng quả hơn, ta cần thu thập hình ảnh khuôn mặt của nhiều người, tạo “embedding features” của từng khuôn mặt và lưu vào kho dữ liệu “embedding features” cùng với tên tương ứng của người đó. Khi muốn nhận dạng một khuôn mặt bất kì, ta đưa hình ảnh khuôn mặt đó vào mô hình để tạo “embedding features”. Sau đó, ta tính cosine giữa “embedding features” này với tất cả các “embedding features” ta đã lưu trong kho dữ liệu. Cosine lớn nhất và đồng thời cosine > 0.5 sẽ được chọn để nhận dạng tên của người đó.

**Tài liệu tham khảo**

[1] MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices, CCBR 2018.

[2]<https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728>

[3]<https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215>

[4] CosFace: Large Margin Cosine Loss for Deep Face Recognition, CVPR 2018.