**人脸表情识别系统技术说明书**  
**目 录**

[1 项目开发目的和意义 2](#_Toc2743)

[2 总体设计 2](#_Toc32598)

[3 算法设计 3](#_Toc15521)

[3.1 Fer-2013数据集 3](#_Toc30008)

[3.2 数据集制作 5](#_Toc29753)

[3.3 模型训练 5](#_Toc20053)

[3.4 模型转换 15](#_Toc1620)

[3.4.1人脸模型转换 15](#_Toc26706)

[3.4.2人脸表情识别模型转换 16](#_Toc19870)

[3.5 人脸检测 17](#_Toc16470)

[3.6 人脸表情识别 17](#_Toc29447)

[3.7 流程设计 17](#_Toc8792)

[4 结果 18](#_Toc18024)

[5 关键代码说明 19](#_Toc27551)

[5.1 摄像头采集数据核心代码 19](#_Toc4470)

[5.2 人脸检测推理的核心代码 23](#_Toc17004)

[5.3 人脸表情推理的核心代码 27](#_Toc24391)

[5.4 人脸表情后处理的核心代码 31](#_Toc29683)

[6 重要问题及解决 33](#_Toc17642)

[7 后续可扩展性 34](#_Toc29443)

# 1 项目开发目的和意义

随着机器学习和深度神经网络两个领域的迅速发展以及智能设备的普及，人脸识别技术正在经历前所未有的发展，关于人脸识别技术讨论从未停歇。目前，人脸识别精度已经超过人眼，同时大规模普及的软硬件基础条件也已具备，应用市场和领域需求很大，基于这项技术的市场发展和具体应用正呈现蓬勃发展态势。人脸表情识别(facial expression recognition, FER)作为人脸识别技术中的一个重要组成部分，近年来在人机交互、安全、机器人制造、自动化、医疗、通信和驾驶领域得到了广泛的关注，成为学术界和工业界的研究热点。本文将对人脸识别中的表情识别的相关内容做一个较为详细的综述。

本项目在华为Atlas200DK上实现了借助摄像头实现了人脸表情实时检测和识别，具有完整性、代表性和实用性，满足了在实际场景下用摄像头进行人脸表情识别的需求。

# 2 总体设计

系统可以划分为3个子系统，各子系统相对独立。为了说明各系统之间的结构关系，细化的整体结构图如2-1所示。



图2-1 系统整体功能结构图

# 3 算法设计

整个算法包括训练和推断两个阶段。训练阶段在Fer-2013数据集上借助Caffe生成定制版人脸表情识别模型，人脸检测模型由于不是本文重点，所以直接采用的Atlast 200DK官方提供的模型。推断阶段包括摄像头图像采集、人脸检测、人脸表情识别等模块后给出输出反馈，具体流程见下图3-1所示。



图3-1 系统流程

## 3.1 Fer-2013数据集

使用Fer2013人脸表情数据集由35886张人脸表情图片组成，其中，测试图（Training）28708张，公共验证图（PublicTest）和私有验证图（PrivateTest）各3589张，每张图片是由大小固定为48×48的灰度图像组成，共有7种表情，分别对应于数字标签0-6，具体表情对应的标签和中英文如下：0 anger 生气； 1 disgust 厌恶； 2 fear 恐惧； 3 happy 开心； 4 sad 伤心；5 surprised 惊讶； 6 normal 中性。但是，数据集并没有直接给出图片，而是将表情、图片数据、用途的数据保存到csv文件中。所以需要我们将csv图片转成jpg格式，主要代码如下：

#encoding:utf-8

import pandas as pd

import numpy as np

import scipy.misc as sm

import os

emotions = {

'0':'anger', #生气

'1':'disgust', #厌恶

'2':'fear', #恐惧

'3':'happy', #开心

'4':'sad', #伤心

'5':'surprised', #惊讶

'6':'normal', #中性

}

#创建文件夹

def createDir(dir):

if os.path.exists(dir) is False:

os.makedirs(dir)

def saveImageFromFer2013(file):

#读取csv文件

faces\_data = pd.read\_csv(file)

imageCount = 0

#遍历csv文件内容，并将图片数据按分类保存

for index in range(len(faces\_data)):

#解析每一行csv文件内容

emotion\_data = faces\_data.loc[index][0]

image\_data = faces\_data.loc[index][1]

usage\_data = faces\_data.loc[index][2]

#将图片数据转换成48\*48

data\_array = list(map(float, image\_data.split()))

data\_array = np.asarray(data\_array)

image = data\_array.reshape(48, 48)

#选择分类，并创建文件名

dirName = usage\_data

emotionName = emotions[str(emotion\_data)]

#图片要保存的文件夹

imagePath = os.path.join(dirName, emotionName)

# 创建“用途文件夹”和“表情”文件夹

createDir(dirName)

createDir(imagePath)

#图片文件名

imageName = os.path.join(imagePath, str(index) + '.jpg')

sm.toimage(image).save(imageName)

imageCount = index

print('总共有' + str(imageCount) + '张图片')

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

saveImageFromFer2013('fer2013.csv')

## 3.2 数据集制作

在caffe中经常使用的数据类型是lmdb，不是常见的jpg,jpeg,png,tif等格式。比起单张图片，它具有I/O效率高、支持多线程并发读写、节省内存、语义完全符合ACID性等特点。由于本项目用caffe训练数据集，我们需要对我们得到的jpg图像进行格式转换，输出成一个lmdb库文件。

生成lmdb需要准备的条件如下：1）编译好caffe并编写好convert\_imageset.exe; 2）将fer-2013数据集csv转成jpg图像; 3）一个标签文件label.txt; 4）用命令编辑好的dos脚本create-lmdb.cmd。

将数据集jpg转换成lmdb十分简单，执行如下命令即可：

convert\_imageset --shuffle --gray -resize\_height=48 --resize\_width=48 ./datasets/train/ ./train.txt ./train\_lmdb

## 3.3 模型训练

在数据集lmdb制作以后就可以开始训练模型了，我们采用的windows-cpu训练，训练只需要调用以下脚本即可执行：caffe train --solver=./solver.prototxt

其中我们使用的的网络配置文件如表3.1：

网络是在AlexNet上修改得到，减少了3个卷积层，现在的网络包含3个卷积层、3个池化

层和3个全连接层细节如下：

表3.1 FacialNet model architecture

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| type | kernel\_size | kernel\_num | stride | pad | output | dropout |
| Data |  |  |  |  | 42 x 42 x 1 |  |
| Convolution | 5 x 5 | 32 | 1 | 2 | 42 x 42 x 32 |  |
| Pooling | 3 x 3 |  | 2 |  | 21 x 21 x 32 |  |
| Convolution | 4 x 4 | 32 | 1 | 1 | 20 x 20 x 32 |  |
| Pooling | 3 x 3 |  | 2 |  | 10 x 10 x 32 |  |
| Convolution | 5 x 5 | 64 | 1 | 2 | 10 x 10 x 64 |  |
| Pooling | 3 x 3 |  | 2 |  | 5 x 5 x 64 |  |
| InnerProduct |  |  |  |  | 1 x 1 x 2048 | 0.5 |
| InnerProduct |  |  |  |  | 1 x 1 x 1024 | 0.5 |
| InnerProduct |  |  |  |  | 1 x 1 x 7 |  |

type：网络中每一层的类型。我们的网络首先是一个Data层，然后紧跟3个卷积层和3个池化层，最后是3个全连接层

kernel\_size：卷积核的尺寸

kernel\_num：卷积核的个数

stride：步长，即卷积核每次移动的长度

pad：扩充边缘，使得图像经过卷积之后得到的特征图象不会改变尺寸

output：经过该层处理后，输出结果的维度

dropout：这个不多说了，减少过拟合

得到output的计算过程：

    1）在Data层，我们的原始图像数据是48\*48\*1的，其中1代表只有一个通道，因为原始图像是灰度图像。经过一个裁剪操作后，将48\*48的图像裁剪成42\*42的大小（裁剪操作在上述表格中没有体现，后面在介绍Data层代码时会介绍到），所以Data层的output为42\*42\*1。

    2）在第1个Convolution层，pad为2，即图像的宽高各增加2个像素，特征图像尺寸变为46\*46，经过卷积之后，特征图像尺寸变为(46-5)/1+1=42，即42\*42，发现经过卷积之后图像尺寸还是42\*42没有改变，这其实是pad=2的功劳。由于卷积核的个数是32，因此在第1个Convolution层，output为42\*42\*32。

    这里介绍一个特征图像的计算公式：特征图像尺寸=[(原特征图像尺寸-卷积核尺寸)/步长]+1

    3）在第1个Pooling层，使用上面公式计算得到特征图像尺寸=(42-3)/2+1=20.5，进1得到21，经过池化层的特征图像大小为21\*21。该层的output为21\*21\*32。

    4）后面的第2个第3个Convolution层和Pooling层的计算方法跟上面相同，不再过多介绍。

    5）全连接层的output不再过多介绍。

上面简单介绍了神经网络的结构，得到output的过程可能比较繁琐，需要读者有一定的卷积神经网络基础。output是网络自己得出的，并不需要我们手动来规定，这里介绍一下每一层得到的数据结果（也就是output）是为了让读者理解更清晰卷积神经网络的工作原理，不然下次自己去搭建一个网络时还是无从下手。至于说为什么第1个卷积层的卷积核尺寸是5x5，第2个是4x4等等这类的问题，只能说这是人们不断实验得到的比较理想的结果，如果有时间可以改变这些取值，看看最后的结果有什么影响，这里就不多说了。

网络配置文件如下：

name: "FacialNet"

layer {

name: "data"

type: "Data"

top: "data"

top: "label"

include {

phase: TRAIN

}

transform\_param {

mirror: true

crop\_size: 48

mean\_file: "./binaryproto/meandata.binaryproto"

}

data\_param {

source: "./fer2013train\_lmdb"

batch\_size: 64

backend: LMDB

}

}

layer {

name: "data"

type: "Data"

top: "data"

top: "label"

include {

phase: TEST

}

transform\_param {

mirror: false

crop\_size: 48

mean\_file: "./binaryproto/meandata.binaryproto"

}

data\_param {

source: "./fer2013test\_lmdb"

batch\_size: 32

backend: LMDB

}

}

layer {

name: "conv1"

type: "Convolution"

bottom: "data"

top: "conv1"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

convolution\_param {

num\_output: 32

kernel\_size: 5

pad: 2

stride: 1

weight\_filler {

type: "gaussian"

std: 0.01

}

bias\_filler {

type: "constant"

value: 0

}

}

}

layer {

name: "relu1"

type: "ReLU"

bottom: "conv1"

top: "conv1"

}

layer {

name: "norm1"

type: "LRN"

bottom: "conv1"

top: "norm1"

lrn\_param {

local\_size: 5

alpha: 0.0001

beta: 0.75

}

}

layer {

name: "pool1"

type: "Pooling"

bottom: "norm1"

top: "pool1"

pooling\_param {

pool: MAX

kernel\_size: 3

stride: 2

}

}

layer {

name: "conv2"

type: "Convolution"

bottom: "pool1"

top: "conv2"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

convolution\_param {

num\_output: 32

pad: 1

kernel\_size: 4

stride: 2

weight\_filler {

type: "gaussian"

std: 0.01

}

bias\_filler {

type: "constant"

value: 0.1

}

}

}

layer {

name: "relu2"

type: "ReLU"

bottom: "conv2"

top: "conv2"

}

layer {

name: "norm2"

type: "LRN"

bottom: "conv2"

top: "norm2"

lrn\_param {

local\_size: 5

alpha: 0.0001

beta: 0.75

}

}

layer {

name: "pool2"

type: "Pooling"

bottom: "norm2"

top: "pool2"

pooling\_param {

pool: MAX

kernel\_size: 3

stride: 2

}

}

layer {

name: "conv3"

type: "Convolution"

bottom: "pool2"

top: "conv3"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

convolution\_param {

num\_output: 64

pad: 2

kernel\_size: 5

stride: 1

weight\_filler {

type: "gaussian"

std: 0.01

}

bias\_filler {

type: "constant"

value: 0

}

}

}

layer {

name: "relu3"

type: "ReLU"

bottom: "conv3"

top: "conv3"

}

layer {

name: "norm3"

type: "LRN"

bottom: "conv3"

top: "norm3"

lrn\_param {

local\_size: 5

alpha: 0.0001

beta: 0.75

}

}

layer {

name: "pool3"

type: "Pooling"

bottom: "norm3"

top: "pool3"

pooling\_param {

pool: MAX

kernel\_size: 3

stride: 2

}

}

layer {

name: "fc4"

type: "InnerProduct"

bottom: "pool3"

top: "fc4"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

inner\_product\_param {

num\_output: 2048

weight\_filler {

type: "gaussian"

std: 0.005

}

bias\_filler {

type: "constant"

value: 0.1

}

}

}

layer {

name: "relu4"

type: "ReLU"

bottom: "fc4"

top: "fc4"

}

layer {

name: "drop4"

type: "Dropout"

bottom: "fc4"

top: "fc4"

dropout\_param {

dropout\_ratio: 0.5

}

}

layer {

name: "fc5"

type: "InnerProduct"

bottom: "fc4"

top: "fc5"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

inner\_product\_param {

num\_output: 1024

weight\_filler {

type: "gaussian"

std: 0.005

}

bias\_filler {

type: "constant"

value: 0.1

}

}

}

layer {

name: "relu5"

type: "ReLU"

bottom: "fc5"

top: "fc5"

}

layer {

name: "drop5"

type: "Dropout"

bottom: "fc5"

top: "fc5"

dropout\_param {

dropout\_ratio: 0.5

}

}

layer {

name: "fc6"

type: "InnerProduct"

bottom: "fc5"

top: "fc6"

param {

lr\_mult: 1

decay\_mult: 1

}

param {

lr\_mult: 2

decay\_mult: 0

}

inner\_product\_param {

num\_output: 7

weight\_filler {

type: "gaussian"

std: 0.005

}

bias\_filler {

type: "constant"

value: 0.1

}

}

}

layer {

name: "accuracy"

type: "Accuracy"

bottom: "fc6"

bottom: "label"

top: "accuracy"

include {

phase: TEST

}

}

layer {

name: "loss"

type: "SoftmaxWithLoss"

bottom: "fc6"

bottom: "label"

top: "loss"

}

超参数文件solver.prototxt配置如下：

net: "./train\_val.prototxt" #用于训练或预测的网络

test\_iter: 110 #测试阶段迭代次数，测试阶段的batch\_size为32,在次设置迭代次数为110,能包含3520个测试样本

test\_interval: 1000 #训练时，每经过1000次迭代，进行一次预测

base\_lr: 0.001 #网络学习率

momentum: 0.9 #网络冲量

weight\_decay: 0.0005 #网络权衰量

#学习速率的衰减策略

lr\_policy: "fixed"

gamma: 0.1

stepsize: 50000

display: 1000 #每经过1000次迭代，在屏幕上打印一次日志

max\_iter: 200000 #训练阶段最大迭代次数

snapshot: 200000 #每两万次迭代得出一个模型

snapshot\_prefix: "./myfacialnet"

solver\_mode: CPU

我们共计迭代了20万次，训练大概需要十几个小时便可结束，以下我们训练模型的accuracy和loss随着迭代次数变化关系：

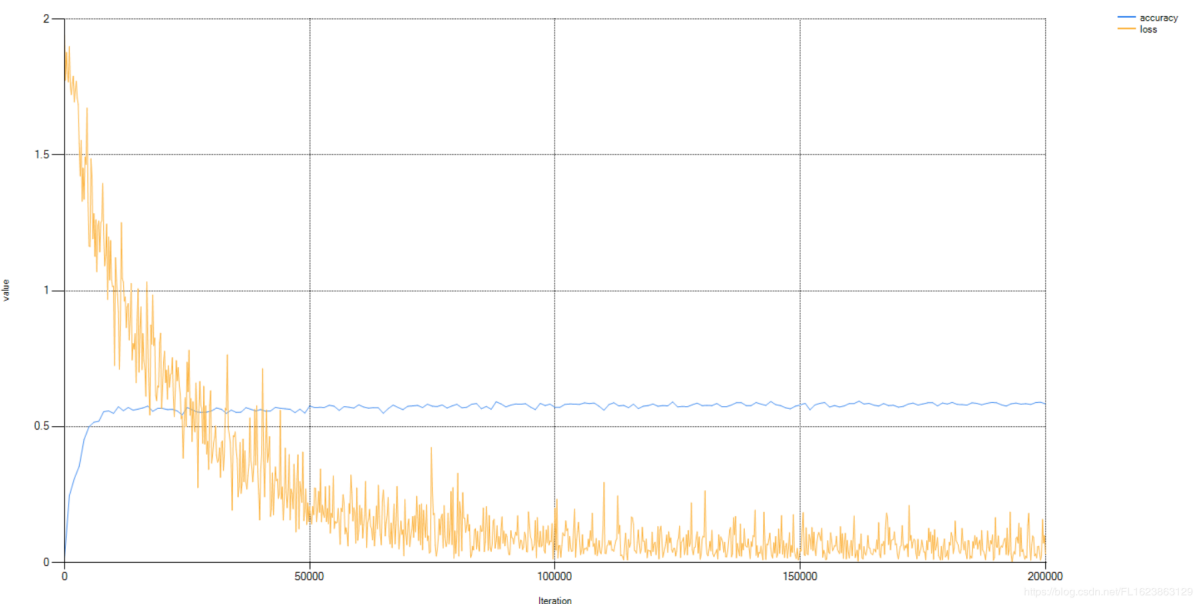


图3.2 人脸表情模型Accuracy和Loss变化趋势图

从上面可以看到我们的模型在迭代10万次左右已经收敛，准确率最终稳定在60多。由于fer数据集的问题，目前网上准确率大多也是60%多，这个已经是符合我们训练的实际需求。

**3.4 模型转换**

**3.4.1人脸模型转换**

人脸检测网络模型是基于Caffe的Resnet10-SSD300模型转换后的网络模型。请参考https://github.com/Ascend/models/tree/master/computer\_vision/object\_detect/face\_detection目录中README.md下载原始网络模型文件及其对应的权重文件。人脸模型转换是在MindSpore Studio进行的，转换参数设置如下图所示：

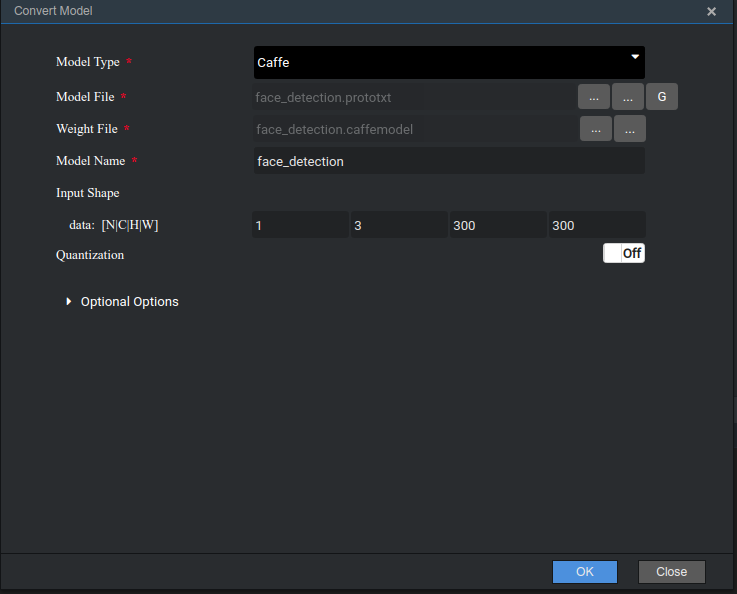


图3.3 人脸检测模型转换

经过转换后可以得到一个OM模型，这个模型就是我们需要的Davinci模型

**3.4.2人脸表情识别模型转换**

同样，我们使用Mindpore Studio将人脸表情识别的caffe模型转为OM模型，具体转换参数如图所示：

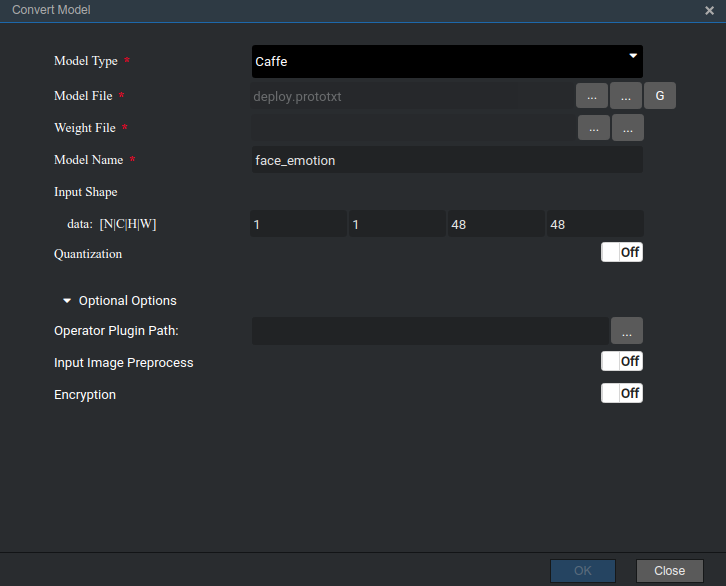


图3.4 人脸表情模型转换

## 3.5 人脸检测



图3.5 人脸检测流程图

人脸检测模块流程：树莓派摄像头捕捉的实时画面后发送到人脸推理引擎，然后经过图像预处理后，经过推理引擎处理会得到人脸框的信息，我们可以借助推理后的信息将人脸区域得到，然后交给人脸表情识别模型进行再次推理。

**3.6 人脸表情识别**



图3.6 人脸表情识别流程图

人脸表情识别模块流程：由人脸检测得到人脸图像后，送至表情推理引擎，这时候会得到人脸表情的推理结果；将推理结果交给表情后处理引擎处理，其中人脸表情是一个7位向量，每个元素是一个推理的结果值，我们将概率最高的值作为最终预测结果，并将结果和图像发送Presenter Server展示。

**3.7 流程设计**

根据人脸表情识别的需求，共设计了三个Engine，分别为摄像头模块、推理模块、后处理模块。

摄像头模块与Camera驱动进行交互，设置摄像头的帧率、图像分辨率、图像格式等相关参数，从摄像头中获取图像数据，每一帧传给推理引擎进行计算。以此工程为例，其中帧率fps为5，图像分辨率取1280x720，摄像头图像格式为默认的YUV420SP。

推理模块共有2个，一个是人脸检测推理模块，一个是人脸表情识别模块。摄像头采集的图像数据，会经过人脸检测推理得到一个人脸位置信息，我们由人脸信息可以得到人脸区域的图像，由于人脸表情模型输入灰度图，我们需要利用Opencv将图像转为灰度图，然后送给人脸表情识别推理引擎，这样我们会得到人脸表情识别推理结果。

后处理模块接收上人脸表情识别引擎的推理结果与摄像头图像，将矩形框集合和人脸表情识别结果添加到Presenter Server记录检测目标位置信息的数据结构DetectionResult类中，作为摄像头图像的检测结果，通过调用Presenter Agent的API发送到UI Host上部署的Presenter Server服务进程。Presenter Server根据接收到的推理结果，并将图像信息发送给Web UI。

# 4 结果

推理结果展示如下：



图4.1 人脸表情识别结果



图4.2 人脸表情识别结果

推理结果可以实时展示，人脸表情识别结果比较准确而且稳定。

由于数据集自身的限制，即数据集中很多数据真实人有时候也分不清是哪一种表情，而且数据集符合真实场景中的情况，所以导致有时候也会出现错误识别结果。

# 5 关键代码说明

## 5.1 摄像头采集数据核心代码

Mind\_Camera::CameraOperationCode Mind\_Camera::PreCapProcess() {

MediaLibInit();

CameraStatus status = QueryCameraStatus(config\_->channel\_id);

if (status != CAMERA\_STATUS\_CLOSED) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess.QueryCameraStatus {status:%d} \

failed.",status);

return kCameraNotClosed;

}

//Open Camera

int ret = OpenCamera(config\_->channel\_id);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess OpenCamera {%d} failed.",

config\_->channel\_id);

return kCameraOpenFailed;

}

//set fps

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_FPS,

&(config\_->fps));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set fps {fps:%d} failed.",

config\_->fps);

return kCameraSetPropertyFailed;

}

// set image format

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_IMAGE\_FORMAT,

&(config\_->image\_format));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set image\_fromat {format:%d} \

failed.",config\_->image\_format);

return kCameraSetPropertyFailed;

}

// set image resolution.

CameraResolution resolution;

resolution.width = config\_->resolution\_width;

resolution.height = config\_->resolution\_height;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_RESOLUTION,

&resolution);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set resolution {width:%d, \

height:%d } failed.",config\_->resolution\_width,

config\_->resolution\_height);

return kCameraSetPropertyFailed;

}

// set work mode

CameraCapMode mode = CAMERA\_CAP\_ACTIVE;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_CAP\_MODE, &mode);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set cap mode {mode:%d} failed.",

mode);

return kCameraSetPropertyFailed;

}

return kCameraOk;

}

std::shared\_ptr<FaceRecognitionInfo>

Mind\_Camera::CreateBatchImageParaObj() {

std::shared\_ptr<FaceRecognitionInfo> pObj = make\_shared<FaceRecognitionInfo>();

// handle one image frame every time

pObj->frame.channel\_id = config\_->channel\_id;

pObj->frame.frame\_id = frame\_id\_++;

pObj->frame.timestamp = time(nullptr);

// channel begin from zero

pObj->org\_img.channel = 0;

pObj->org\_img.format = YUV420SP;

pObj->org\_img.width = config\_->resolution\_width;

pObj->org\_img.height = config\_->resolution\_height;

// YUV size in memory is width\*height\*3/2

pObj->org\_img.size = config\_->resolution\_width \* config\_->resolution\_height

\* 3 / 2;

shared\_ptr<uint8\_t> data(new uint8\_t[pObj->org\_img.size],

default\_delete<uint8\_t[]>());

pObj->org\_img.data = data;

return pObj;

}

bool Mind\_Camera::DoCapProcess() {

CameraOperationCode retCode = PreCapProcess();

if (retCode == kCameraSetPropertyFailed) {

CloseCamera(config\_->channel\_id);

HIAI\_ENGINE\_LOG( "[Mind\_Camera] DoCapProcess.PreCapProcess failed");

return false;

}

// set procedure is running.

SetExitFlag(CAMERADATASETS\_RUN);

HIAI\_StatusT hiai\_ret = HIAI\_OK;

int read\_ret = 0;

int read\_size = 0;

bool read\_flag = false;

while (GetExitFlag() == CAMERADATASETS\_RUN) {

std::shared\_ptr<FaceRecognitionInfo> p\_obj =

CreateBatchImageParaObj();

uint8\_t\* p\_data = p\_obj->org\_img.data.get();

read\_size = (int) p\_obj->org\_img.size;

// do read frame from camera, readSize maybe changed when called

read\_ret = ReadFrameFromCamera(config\_->channel\_id, (void\*) p\_data,

&read\_size);

// indicates failure when readRet is 1

read\_flag = ((read\_ret == 1) && (read\_size == (int) p\_obj->org\_img.size));

if (!read\_flag) {

HIAI\_ENGINE\_LOG("[CameraDatasets] readFrameFromCamera failed "

"{camera:%d, ret:%d, size:%d, expectsize:%d} ",

config\_->channel\_id, read\_ret, read\_size,

(int )p\_obj->org\_img.size);

break;

}

hiai\_ret = SendData(0, "FaceRecognitionInfo",

static\_pointer\_cast<void>(p\_obj));

if (hiai\_ret != HIAI\_OK) {

HIAI\_ENGINE\_LOG("[CameraDatasets] senddata failed! {frameid:%d, "

"timestamp:%lu}",

p\_obj->frame.frame\_id, p\_obj->frame.timestamp);

break;

}

}

// close camera

CloseCamera(config\_->channel\_id);

if (HIAI\_OK != hiai\_ret) {

return false;

}

return true;

}

void Mind\_Camera::SetExitFlag(int flag) {

TLock lock(mutex\_);

exit\_flag\_ = flag;

}

int Mind\_Camera::GetExitFlag() {

TLock lock(mutex\_);

return exit\_flag\_;

}

HIAI\_IMPL\_ENGINE\_PROCESS("Mind\_Camera", Mind\_Camera, INPUT\_SIZE)

{

HIAI\_ENGINE\_LOG("[Mind\_Camera] start process!");

DoCapProcess();

HIAI\_ENGINE\_LOG("[Mind\_Camera] end process!");

return HIAI\_OK;

}

## 5.2 人脸检测推理的核心代码

Mind\_Camera::CameraOperationCode Mind\_Camera::PreCapProcess() {

MediaLibInit();

CameraStatus status = QueryCameraStatus(config\_->channel\_id);

if (status != CAMERA\_STATUS\_CLOSED) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess.QueryCameraStatus {status:%d} \

failed.",status);

return kCameraNotClosed;

}

//Open Camera

int ret = OpenCamera(config\_->channel\_id);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess OpenCamera {%d} failed.",

config\_->channel\_id);

return kCameraOpenFailed;

}

//set fps

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_FPS,

&(config\_->fps));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set fps {fps:%d} failed.",

config\_->fps);

return kCameraSetPropertyFailed;

}

// set image format

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_IMAGE\_FORMAT,

&(config\_->image\_format));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set image\_fromat {format:%d} \

failed.",config\_->image\_format);

return kCameraSetPropertyFailed;

}

// set image resolution.

CameraResolution resolution;

resolution.width = config\_->resolution\_width;

resolution.height = config\_->resolution\_height;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_RESOLUTION,

&resolution);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set resolution {width:%d, \

height:%d } failed.",config\_->resolution\_width,

config\_->resolution\_height);

return kCameraSetPropertyFailed;

}

// set work mode

CameraCapMode mode = CAMERA\_CAP\_ACTIVE;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_CAP\_MODE, &mode);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set cap mode {mode:%d} failed.",

mode);

return kCameraSetPropertyFailed;

}

return kCameraOk;

}

std::shared\_ptr<FaceRecognitionInfo>

Mind\_Camera::CreateBatchImageParaObj() {

std::shared\_ptr<FaceRecognitionInfo> pObj = make\_shared<FaceRecognitionInfo>();

// handle one image frame every time

pObj->frame.channel\_id = config\_->channel\_id;

pObj->frame.frame\_id = frame\_id\_++;

pObj->frame.timestamp = time(nullptr);

// channel begin from zero

pObj->org\_img.channel = 0;

pObj->org\_img.format = YUV420SP;

pObj->org\_img.width = config\_->resolution\_width;

pObj->org\_img.height = config\_->resolution\_height;

// YUV size in memory is width\*height\*3/2

pObj->org\_img.size = config\_->resolution\_width \* config\_->resolution\_height

\* 3 / 2;

shared\_ptr<uint8\_t> data(new uint8\_t[pObj->org\_img.size],

default\_delete<uint8\_t[]>());

pObj->org\_img.data = data;

return pObj;

}

bool Mind\_Camera::DoCapProcess() {

CameraOperationCode retCode = PreCapProcess();

if (retCode == kCameraSetPropertyFailed) {

CloseCamera(config\_->channel\_id);

HIAI\_ENGINE\_LOG( "[Mind\_Camera] DoCapProcess.PreCapProcess failed");

return false;

}

// set procedure is running.

SetExitFlag(CAMERADATASETS\_RUN);

HIAI\_StatusT hiai\_ret = HIAI\_OK;

int read\_ret = 0;

int read\_size = 0;

bool read\_flag = false;

while (GetExitFlag() == CAMERADATASETS\_RUN) {

std::shared\_ptr<FaceRecognitionInfo> p\_obj =

CreateBatchImageParaObj();

uint8\_t\* p\_data = p\_obj->org\_img.data.get();

read\_size = (int) p\_obj->org\_img.size;

// do read frame from camera, readSize maybe changed when called

read\_ret = ReadFrameFromCamera(config\_->channel\_id, (void\*) p\_data,

&read\_size);

// indicates failure when readRet is 1

read\_flag = ((read\_ret == 1) && (read\_size == (int) p\_obj->org\_img.size));

if (!read\_flag) {

HIAI\_ENGINE\_LOG("[CameraDatasets] readFrameFromCamera failed "

"{camera:%d, ret:%d, size:%d, expectsize:%d} ",

config\_->channel\_id, read\_ret, read\_size,

(int )p\_obj->org\_img.size);

break;

}

hiai\_ret = SendData(0, "FaceRecognitionInfo",

static\_pointer\_cast<void>(p\_obj));

if (hiai\_ret != HIAI\_OK) {

HIAI\_ENGINE\_LOG("[CameraDatasets] senddata failed! {frameid:%d, "

"timestamp:%lu}",

p\_obj->frame.frame\_id, p\_obj->frame.timestamp);

break;

}

}

// close camera

CloseCamera(config\_->channel\_id);

if (HIAI\_OK != hiai\_ret) {

return false;

}

return true;

}

void Mind\_Camera::SetExitFlag(int flag) {

TLock lock(mutex\_);

exit\_flag\_ = flag;

}

int Mind\_Camera::GetExitFlag() {

TLock lock(mutex\_);

return exit\_flag\_;

}

HIAI\_IMPL\_ENGINE\_PROCESS("Mind\_Camera", Mind\_Camera, INPUT\_SIZE)

{

HIAI\_ENGINE\_LOG("[Mind\_Camera] start process!");

DoCapProcess();

HIAI\_ENGINE\_LOG("[Mind\_Camera] end process!");

return HIAI\_OK;

}

## 5.3 人脸表情推理的核心代码

Mind\_Camera::CameraOperationCode Mind\_Camera::PreCapProcess() {

MediaLibInit();

CameraStatus status = QueryCameraStatus(config\_->channel\_id);

if (status != CAMERA\_STATUS\_CLOSED) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess.QueryCameraStatus {status:%d} \

failed.",status);

return kCameraNotClosed;

}

//Open Camera

int ret = OpenCamera(config\_->channel\_id);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess OpenCamera {%d} failed.",

config\_->channel\_id);

return kCameraOpenFailed;

}

//set fps

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_FPS,

&(config\_->fps));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set fps {fps:%d} failed.",

config\_->fps);

return kCameraSetPropertyFailed;

}

// set image format

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_IMAGE\_FORMAT,

&(config\_->image\_format));

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set image\_fromat {format:%d} \

failed.",config\_->image\_format);

return kCameraSetPropertyFailed;

}

// set image resolution.

CameraResolution resolution;

resolution.width = config\_->resolution\_width;

resolution.height = config\_->resolution\_height;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_RESOLUTION,

&resolution);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set resolution {width:%d, \

height:%d } failed.",config\_->resolution\_width,

config\_->resolution\_height);

return kCameraSetPropertyFailed;

}

// set work mode

CameraCapMode mode = CAMERA\_CAP\_ACTIVE;

ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_CAP\_MODE, &mode);

if (ret == 0) {

HIAI\_ENGINE\_LOG(

"[Mind\_Camera] PreCapProcess set cap mode {mode:%d} failed.",

mode);

return kCameraSetPropertyFailed;

}

return kCameraOk;

}

std::shared\_ptr<FaceRecognitionInfo>

Mind\_Camera::CreateBatchImageParaObj() {

std::shared\_ptr<FaceRecognitionInfo> pObj = make\_shared<FaceRecognitionInfo>();

// handle one image frame every time

pObj->frame.channel\_id = config\_->channel\_id;

pObj->frame.frame\_id = frame\_id\_++;

pObj->frame.timestamp = time(nullptr);

// channel begin from zero

pObj->org\_img.channel = 0;

pObj->org\_img.format = YUV420SP;

pObj->org\_img.width = config\_->resolution\_width;

pObj->org\_img.height = config\_->resolution\_height;

// YUV size in memory is width\*height\*3/2

pObj->org\_img.size = config\_->resolution\_width \* config\_->resolution\_height

\* 3 / 2;

shared\_ptr<uint8\_t> data(new uint8\_t[pObj->org\_img.size],

default\_delete<uint8\_t[]>());

pObj->org\_img.data = data;

return pObj;

}

bool Mind\_Camera::DoCapProcess() {

CameraOperationCode retCode = PreCapProcess();

if (retCode == kCameraSetPropertyFailed) {

CloseCamera(config\_->channel\_id);

HIAI\_ENGINE\_LOG( "[Mind\_Camera] DoCapProcess.PreCapProcess failed");

return false;

}

// set procedure is running.

SetExitFlag(CAMERADATASETS\_RUN);

HIAI\_StatusT hiai\_ret = HIAI\_OK;

int read\_ret = 0;

int read\_size = 0;

bool read\_flag = false;

while (GetExitFlag() == CAMERADATASETS\_RUN) {

std::shared\_ptr<FaceRecognitionInfo> p\_obj =

CreateBatchImageParaObj();

uint8\_t\* p\_data = p\_obj->org\_img.data.get();

read\_size = (int) p\_obj->org\_img.size;

// do read frame from camera, readSize maybe changed when called

read\_ret = ReadFrameFromCamera(config\_->channel\_id, (void\*) p\_data,

&read\_size);

// indicates failure when readRet is 1

read\_flag = ((read\_ret == 1) && (read\_size == (int) p\_obj->org\_img.size));

if (!read\_flag) {

HIAI\_ENGINE\_LOG("[CameraDatasets] readFrameFromCamera failed "

"{camera:%d, ret:%d, size:%d, expectsize:%d} ",

config\_->channel\_id, read\_ret, read\_size,

(int )p\_obj->org\_img.size);

break;

}

hiai\_ret = SendData(0, "FaceRecognitionInfo",

static\_pointer\_cast<void>(p\_obj));

if (hiai\_ret != HIAI\_OK) {

HIAI\_ENGINE\_LOG("[CameraDatasets] senddata failed! {frameid:%d, "

"timestamp:%lu}",

p\_obj->frame.frame\_id, p\_obj->frame.timestamp);

break;

}

}

// close camera

CloseCamera(config\_->channel\_id);

if (HIAI\_OK != hiai\_ret) {

return false;

}

return true;

}

void Mind\_Camera::SetExitFlag(int flag) {

TLock lock(mutex\_);

exit\_flag\_ = flag;

}

int Mind\_Camera::GetExitFlag() {

TLock lock(mutex\_);

return exit\_flag\_;

}

HIAI\_IMPL\_ENGINE\_PROCESS("Mind\_Camera", Mind\_Camera, INPUT\_SIZE)

{

HIAI\_ENGINE\_LOG("[Mind\_Camera] start process!");

DoCapProcess();

HIAI\_ENGINE\_LOG("[Mind\_Camera] end process!");

return HIAI\_OK;

}

## 5.4 人脸表情后处理的核心代码

int face\_emotion\_postprocess::GetMaxIndex(float data[7])

{

float maxValue = data[0];

int index=0;

for(int i=1;i<=6;i++)

{

if(maxValue<data[i])

{

maxValue=data[i];

index=i;

}

}

return index;

}

HIAI\_StatusT face\_emotion\_postprocess::HandleResults(

const std::shared\_ptr<FaceRecognitionInfo> &inference\_res) {

HIAI\_StatusT status = HIAI\_OK;

std::vector<FaceImage> face\_img\_vec = inference\_res->face\_imgs;

ConvertImage(inference\_res->org\_img);//转换为jpeg格式

uint32\_t width = inference\_res->org\_img.width;

uint32\_t height = inference\_res->org\_img.height;

uint32\_t img\_size = inference\_res->org\_img.size;

std::vector<DetectionResult> detection\_results;//发送的结果

for(int myind = 0; myind < inference\_res->face\_imgs.size(); myind++)

{

DetectionResult oneResult;//存储一张面孔的左上、右下坐标

Point point\_lt, point\_rb;

point\_lt.x = inference\_res->face\_imgs[myind].rectangle.lt.x;

point\_lt.y= inference\_res->face\_imgs[myind].rectangle.lt.y;

point\_rb.x= inference\_res->face\_imgs[myind].rectangle.rb.x;

point\_rb.y= inference\_res->face\_imgs[myind].rectangle.rb.y;

oneResult.lt = point\_lt;

oneResult.rb = point\_rb;

//char \*ret;

//GetEmotionString(face\_img\_vec[myind].result,7,ret);

//printf("EmotionString:%s\n",ret);

//{"sad", "disgust", "fear", "happy", "angry", "surprise", "neutral"};

std:string retString = "unkonw";

//printf("%f=%f=%f=%f=%f=%f=%f", face\_img\_vec[myind].result[0], face\_img\_vec[myind].result[1], face\_img\_vec[myind].result[2], face\_img\_vec[myind].result[3], face\_img\_vec[myind].result[4], face\_img\_vec[myind].result[5], face\_img\_vec[myind].result[6]);

int index = GetMaxIndex(face\_img\_vec[myind].result);

switch (index)

{

case 0:

retString = "sad";

break;

case 1:

retString = "disgust";

break;

case 2:

retString = "fear";

break;

case 3:

retString = "happy";

break;

case 4:

retString = "angry";

break;

case 5:

retString = "surprise";

break;

case 6:

retString = "neutral";

break;

default:

retString = "unknown";

break;

}

oneResult.result\_text.append("emotion result:");

oneResult.result\_text.append(retString);

printf("append over");

// oneResult.result\_text.append(",yaw:");

// oneResult.result\_text.append(to\_string(inference\_res->face\_imgs[myind].infe\_res.face\_emotion[1]));

// oneResult.result\_text.append(",roll:");

// oneResult.result\_text.append(to\_string(inference\_res->face\_imgs[myind].infe\_res.face\_emotion[2]));

detection\_results.emplace\_back(oneResult);

}

int32\_t ret;

ret = SendImage(height, width, img\_size, inference\_res->org\_img.data.get(), detection\_results);

// check send result

if (ret == kFdFunFailed) {

status = HIAI\_ERROR;

}

return status;

}

HIAI\_IMPL\_ENGINE\_PROCESS("face\_emotion\_postprocess", face\_emotion\_postprocess, INPUT\_SIZE) {

if (arg0 == nullptr) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Failed to process invalid message.");

return HIAI\_ERROR;

}

# 6 重要问题及解决

问题1：caffe环境配置提示无法打开libcaffe.lib。

解答：提示此问题是因为编译未成功，应检查前面其他文件，可忽略此error。

问题2：人脸表情识别展示在Presenter Server的人脸位置是歪的。

解答：由于转换的人脸检测模型与对应的Mind Studio和DDK版本不匹配，重新转换对应的版本即可解决问题。

问题3：人脸表情训练模型loss已经收敛了，但是准确率为什么只有60%?

解答：由于数据集采用的fer2013数据集符合真实场景的人脸表情，即使是真人也未必能分清楚具体表情是属于哪一种，再加上网络设计的限制，最终也只有60%多，根据资料

问题4：atlas200DK配置usb ip地址后，每次开机都需要重新配置。

解答：其实电脑已经配置完成，在每次开机后执行网络重启，刷新网络即可。

问题5：ubuntu电脑连接开发板后不能上网问题。

解答：因为是实验室电脑，IP为192.168.1.\*，与开发板USB端口IP存在冲突，修改开发板USB端口IP为192.168.2.2，同时在ubuntu电脑中配置相同网段的USB端口虚拟IP，问题解决。

问题6：人脸模型转换时候出错。

解答：人脸表情模型采用的灰度图像输入,为48x48，通道数为1而不是3。

问题7：人脸表情推理过程崩溃。

解答：推理输入时灰度图像，所以在人脸检测后需要把得到的图像转为灰度图送入推理引擎。

问题8：配置开发板环境有哪些注意事项

解答：

1. ubuntu主机与开发板系统版本保持一致，均为16.04.3。
2. Ubuntu主机要求安装64位谷歌浏览器，且版本不低于67.0.3396。
3. Mindstudio的ip应设置为主机ip。
4. 每次开关机最运行stop.sh命令，关闭 mindstudio程序，避免下次使用出现数据库冲突等问题。
5. 首次启动或者升级atlas200DK开发者板时不能断电，以免对开发板造成损坏，再次上电与上次上电保持2s以上安全时间间隔。
6. 若ubuntu主机使用SSH登录开发板提示无信任关系，在ubuntu主机端执行以下命令回复信任关系sh-keygen -f "$HOME/.ssh/known\_hosts" -R *192.168.1.2。*
7. 若更换开发者板或者SD卡，或重置系统，则容易出现信任关系过期问题，无法登陆开发板，输入以下命令删除历史信任关系即可：ssh-keygen -R 192.168.1.2。
8. 对配置环境过程中以及随后登录过程中设置和修改的密码做记录，备份，防止忘记。

# 7 后续可扩展性

本项目中期任务聚焦于几万张的人脸表情识别分类任务，我们利用更多的数据集然后重新训练增加人脸识别的准确率。