**人脸表情识别实验**

目录

[1 实验介绍 2](#_Toc16494)

[2 实验目的 2](#_Toc23117)

[3 预备知识 2](#_Toc29480)

[4 实验环境 2](#_Toc18463)

[5 实验原理及流程 3](#_Toc12756)

[5.1 实验原理 3](#_Toc9564)

[5.2 实验流程 4](#_Toc14714)

[6 实验任务及步骤 5](#_Toc28397)

[6.1 应用代码编写/编译 5](#_Toc32608)

[6.2模型转换 25](#_Toc26897)

[7 实验小结 34](#_Toc4751)

人脸表情识别实验

1 实验介绍

本实验主要介绍人脸表情识别项目代码开发并部署在Atlas 200 DK开发板上执行的方法。通过Atlas 200 DK开发板来实现人脸表情识别推理实验，使用Atlas 200 DK外接的摄像头获取的视频数据作为输入，实时检测视频画面中的人脸表情，并将检测后的结果展示出来。用户可以通过人脸表情识别项目对Atlas 200 DK开发板在AI方面的应用有全面的认识。

2 实验目的

了解熟悉人脸表情识别应用代码的编写(C++语言)。

掌握将应用部署在Atlas 200 DK开发板上的操作。

3 预备知识

具备一定的深度学习理论知识，对业界主流的深度学习框架（Caffe、TensorFlow等）有一定了解。

具备C++开发能力，具备一定的Shell脚本、Python脚本开发能力。

了解Cmake的语法知识和google-protobuf的数据格式

了解Linux操作系统的基本使用。

4 实验环境

实验环境需要从硬件和软件两个方面进行准备：

（1） 硬件配件准备环境：

使用Atlas 200 DK前，需自行购买相关配件，包含制作Atlas 200 DK启动系统的micro SD卡、读卡器，与Ubuntu虚拟机相连接的Type-C数据线及摄像头等配件，详细的配件信息如表4.1所示：

表4-1硬件配件清单

|  |  |  |
| --- | --- | --- |
| 配件名称 | 描述 | 推荐型号 |
| SD卡 | 用于制作Atlas 200 DK开发者板启动系统。 | 推荐使用经过测试的SD卡：  三星UHS-I U3 CLASS 10 64G  金士顿UHS-I U1 CLASS 10 64G |
| 读卡器 | 使用读卡器制作SD卡的场景。 | 支持USB3.0协议 |
| Type-C连接线 | 用于将开发板与Mind Studio所在服务器通过USB方式连接。 | 支持USB3.0的Type-C连接线 |
| 摄像头 | 用于与Atlas 200 DK连接获取视频。 | 兼容树莓派摄像头，如果使用树莓派摄像头，需要额外购买黄色排线 |
| 摄像头支架  （可选） | 用于固定摄像头。 | 树莓派透明摄像头支架 |

（2）软件部署环境：

已在Ubuntu机器上搭建好Mind Studio1.3版本的环境（参考文档：https://ascend.huawei.com/doc/mindstudio/2.1.0(beta)/zh/zh-cn\_topic\_0188160926.html）。

已配置部署好Atlas 200DK环境（参考文档：<https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn_topic_0199239495.html>）。

5 实验原理及流程

5.1 实验原理

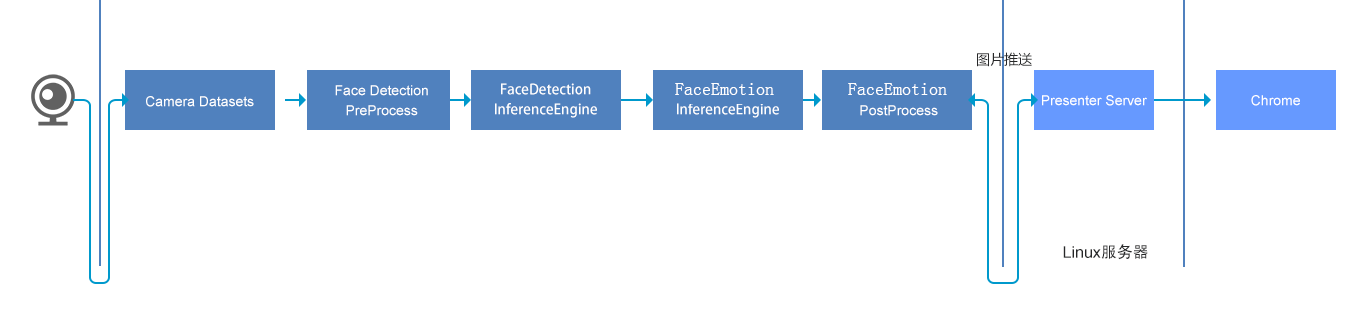


图5.1 人脸表情识别实验原理图

由原理图可知，本实验的数据采集、图片数据预处理、推理计算及检测结果后处理环节均在Atlas 200 DK上完成，在Linux主机PC端调用PresenterServer服务器，将人脸检测的结果显示在PC的网页上，其中，在主机PC端实现检测结果的展示可以有多种处理方式，读者可根据实际情况，自定义人脸检测结果的显示方式，例如直接将检测数据结果打印在终端里显示，或者将检测数据结果保存在文件中。

在本实验中，主要聚焦在Atlas 200 DK开发板上的应用案例移植环节，因此读者需要重点关注摄像头采集图片数据处理、图片数据预处理及推理图片、检测结果后处理环节的操作。

完整的实验流程涉及到的模块介绍如下：

1. CameraDatasets模块与Camera驱动进行交互，从摄像头获取YUV420SP格式的视频数据。Atlas 200 DK提供了一套帮助开发者轻松获取摄像头图像的API接口媒体库，详细的接口使用方法可参考Media API ( https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0190813674.html ) 。
2. FaceDetectionPreProcess是图像预处理模块，当输入图片的分辨率与网络模型要求的分辨率不匹配时，对图片进行resize预处理。AscendDK提供了EZDVPP接口，对DVPP接口进行了封装，帮助开发者更方便地对图片进行预处理，详细的接口使用方法可参考EZDVPP (https://ascend.huawei.com/doc/Atlas200DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0195891320.html)。
3. FaceDetectionlnferenceEngine会加载已训练好的人脸检测网络模型及其权值，对图片做推理，并将图片转化为JPEG格式，推理接口详细使用方法可参考Matrix API (https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0195891333.html)。
4. FaceEmotionlnferenceEngine会对上一个人脸检测的推理引擎结果分割得到人脸图片，然后对图片做推理
5. FaceEmotionPostProcess将接收到人脸表情推理结果通过调用Presenter Agent的API发送到主机端UI host上部署的Presenter Server服务进程，Presenter Agent API的详细使用方法可参考Presenter Agent (https://ascend.huawei.com/doc/atlas200dk/1.3.0.0/zh/zh-cn\_topic\_0165841897.html）。
6. Presenter Server根据接收到的推理结果，在JPEG图片上进行人脸位置及置信度的标记，并将图像信息推送给主机端Web Ul，用户可通过Chrome浏览器访问Presenter Server,实时查看视频中的人脸表情识别结果信息。

5.2 实验流程

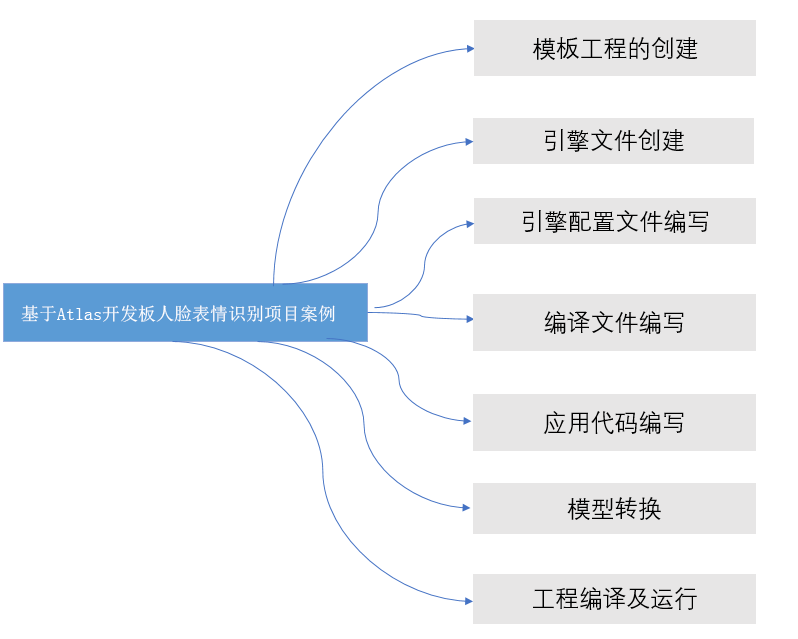


图 5.2 人脸表情识别应用案例移植流程图

在本实验中，默认已完成硬件环境和软件环境的准备工作，在此基础上进行人脸检测应用项目的实验操作，由上图可知，本实验需要分别在Ubuntu主机PC端完成基于C++的人脸表情识别应用代码的编写和编译工作，以及人脸检测和人脸表情识别模型转换，最后在Atlas 200 DK开发板上进行项目部署执行工作。

本案例移植的源代码编写及编译以码云源码为例进行说明，实验任务及步骤将围绕图5.2所示五个方面分别展开介绍。

6 实验任务及步骤

6.1 应用代码编写/编译

将打包下载后的代码包解压，可以看到如图6.1所示的文件目录结构，主要说明如下：

./faceemotionapp：存放人脸表情识别应用工程源码相关文件

./MyDataset：存放数据集相关属性文件

./MyModel：执行部署脚本文件时，存入转换后的模型文件

./presenterserver：存放检测结果展示相关文件

./script：存放相关部署脚本文件

deploy.sh：部署应用运行环境的脚本文件

run\_faceemotionapp.sh：执行人脸表情识别应用的脚本文件

stop\_faceemotionapp.sh：停止人脸表情识别应用运行的脚本文件

选择“faceemotionapp”进入人脸表情识别应用工程源码文件夹，如图7.1所示，接下来将围绕main.cpp、Mind\_camera\_datasets.cpp、face\_detection\_inference.cpp、face\_emotion\_inference.cpp、face\_emotion\_postprocess.cpp等cpp文件进行代码说明：

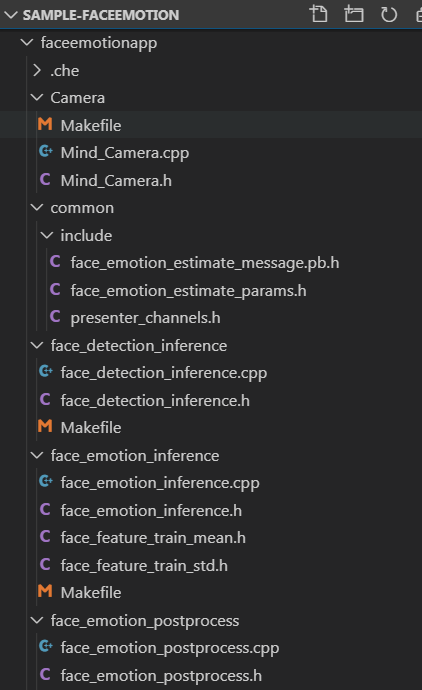


图6.1 faceemotion项目工程文件目录结构图

main.cpp编写

main.cpp文件主要功能如图7.2所示，具体描述及代码实现详见下文：

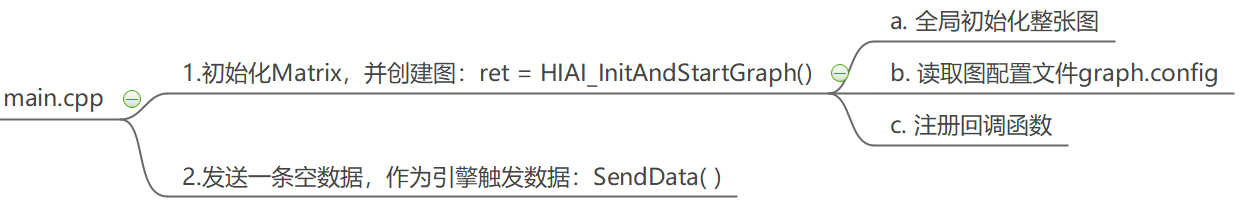


图6.2 main.cpp函数文件功能

初始化Matrix、创建图，main函数主体如下所示：

int main(int argc, char\* argv[])  
{  
 HIAI\_StatusT ret = HIAI\_OK;  
 char \* dirc = strdup(argv[0]);  
 if (dirc)  
 {  
 char \* dname = ::dirname(dirc);  
 chdir(dname);  
 HIAI\_ENGINE\_LOG("chdir to %s", dname);  
 free(dirc);  
 }  
 // 1.创建图  
 ret = HIAI\_InitAndStartGraph();  
  
 // 2.发送触发数据：从main方法中，发送一个空数据给数据集引擎，触发数据传送

// 根据图的ID（GRAPH\_ID），拿到当前工程的图对象 graph  
 std::shared\_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(GRAPH\_ID);  
   
 // 将数据送到SourceEngine的0号端口  
 hiai::EnginePortID engine\_id;  
 engine\_id.graph\_id = GRAPH\_ID;  
 engine\_id.engine\_id = 958; // 数据集引擎Mind\_camera\_datasets的编号  
 engine\_id.port\_id = 0; //数据集引擎Mind\_camera\_datasets的输入端口：0  
 std::shared\_ptr<std::string> src\_data(new std::string);  
 graph->SendData(engine\_id, "string", std::static\_pointer\_cast<void>(src\_data));  
 for (;;)  
 {  
 if(flag <= 0)  
 {  
 break;  
 }else  
 {  
 usleep(100000);  
 }  
 }  
 hiai::Graph::DestroyGraph(GRAPH\_ID);  
 return 0;  
}

其中，HIAI\_InitAndStartGraph()函数展开内容如下所示：

HIAI\_StatusT HIAI\_InitAndStartGraph()  
{  
 // Step1: 在运行引擎之前，需要全局初始化整张图  
 HIAI\_StatusT status = HIAI\_Init(0);  
  
 // Step2: 读取图配置文件graph.config并创建图  
 status = hiai::Graph::CreateGraph("./graph.config");  
  
 // Step3 注册回调函数：向三个engine中的最后一个engine（本实验为后处理引擎）注册一个回调方法  
 std::shared\_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(GRAPH\_ID);  
 int leaf\_array[1] = {601}; //最后一个引擎的id，即后处理引擎  
 //这里只注册一个回调方法，因此for循环只执行一次，后续可以根据实际需求，选择注册多个回调函数方法  
 for(int i = 0;i < 1;i++){  
 hiai::EnginePortID target\_port\_config;  
 target\_port\_config.graph\_id = GRAPH\_ID;  
 target\_port\_config.engine\_id = leaf\_array[i];   
 target\_port\_config.port\_id = 0;//最后一个引擎的输出端口：0号端口  
 graph->SetDataRecvFunctor(target\_port\_config,  
 std::shared\_ptr<CustomDataRecvInterface>(  
 new CustomDataRecvInterface("")));  
 }  
 return HIAI\_OK;  
}

其中，回调函数方法如下所示：

HIAI\_StatusT CustomDataRecvInterface::RecvData  
 (const std::shared\_ptr<void>& message)  
 //message指最后一个引擎（本实验中为后处理引擎）中输出的信息  
{  
 std::shared\_ptr<std::string> data =  
 std::static\_pointer\_cast<std::string>(message);  
 mt.lock();  
 flag--;   
 mt.unlock();  
 return HIAI\_OK;  
}

至此，main.cpp的主要代码实现及说明已介绍完毕，接下来将分别介绍本实验的3个引擎实现的功能代码。

Mind\_camera\_datasets.cpp编写

在本实验项目中，不需要用到main函数传入的数据，只需要从main函数文件中获得一个触发数据激活Mind\_camera\_dataset引擎即可，Camera会自动采集获取外部数据。

在本实验项目中，3个引擎文件.cpp中，都会包含一个初始化函数和一个数据处理函数，Mind\_camera\_datasets.cpp文件结构框架如图6.3所示：

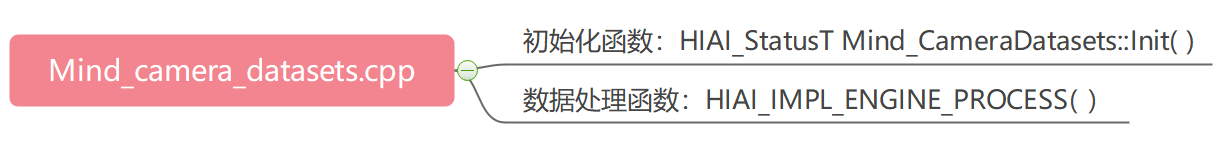


图6.3 Mind\_camera\_datasets.cpp文件结构框架

其中，数据处理函数的主要功能函数结构如图6.4所示：

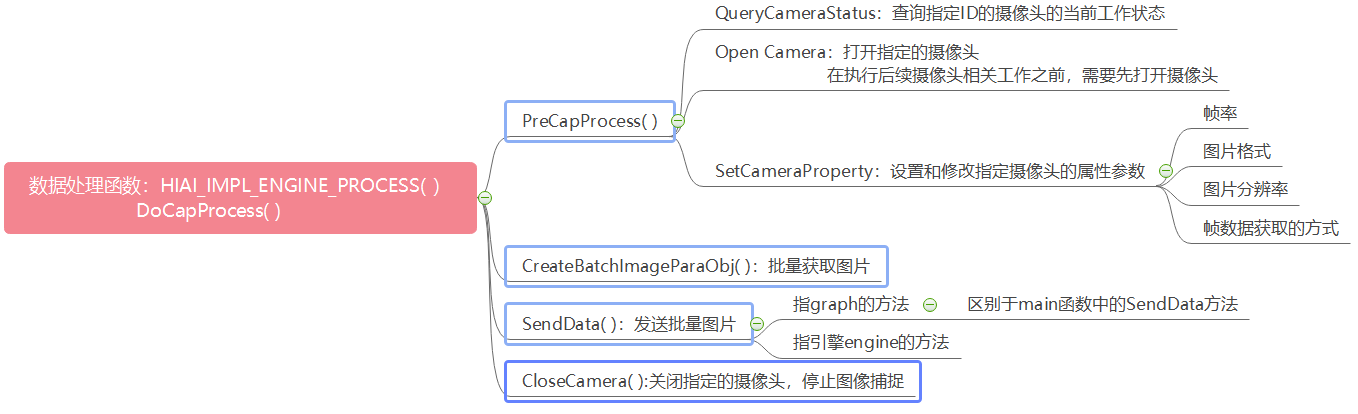


图6.4 Camera数据集引擎数据处理函数的主要功能函数结构

数据处理函数HIAI\_IMPL\_ENGINE\_PROCESS( )具体代码实现如下所示：

HIAI\_IMPL\_ENGINE\_PROCESS("Mind\_camera\_datasets", Mind\_CameraDatasets, INPUT\_SIZE)  
{  
 HIAI\_ENGINE\_LOG("[CameraDatasets] start process!");  
 DoCapProcess();//这里不需要使用main方法传进来的数据，只需要main方法做一个传输数据的动作，来触发一下这里数据处理的宏  
 HIAI\_ENGINE\_LOG("[CameraDatasets] end process!");  
 return HIAI\_OK;  
}

其中，DoCapProcess( )函数主要代码内容及相关注释如下所示：

bool Mind\_CameraDatasets::DoCapProcess() {  
 CameraOperationCode ret\_code = PreCapProcess();  
  
 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) {  
 shared\_ptr < BatchImageParaWithScaleT > pobj = CreateBatchImageParaObj();//CreateBatchImageParaObj创建批量获取图片  
 NewImageParaT\* pimg\_data = &pobj->v\_img[0];  
 uint8\_t\* pdata = pimg\_data->img.data.get();  
 read\_size = (int) pimg\_data->img.size;  
  
 // do read frame from camera  
 read\_ret = ReadFrameFromCamera(config\_->channel\_id, (void\*) pdata,  
 &read\_size);  
 // indicates failure when readRet is 1  
 read\_flag = ((read\_ret == 1) && (read\_size == (int) pimg\_data->img.size));  
 }  
  
 // "BatchImageParaWithScaleT" 表示 向下一个引擎中发送数据的类型为该类型  
 // SendData方法中的第一个参数0，表示在本引擎中，即Camera引擎中的0号端口输出数据  
 hiai\_ret = SendData(0, "BatchImageParaWithScaleT",  
 static\_pointer\_cast<void>(pobj));  
 }  
 }  
  
 // close camera：关闭指定的摄像头，停止图像捕捉  
 CloseCamera(config\_->channel\_id);  
 return true;  
}

其中，数据处理函数的主要调用函数为DoCapProcess( )函数，如图5.21所示，在DoCapProcess( )函数中，还涉及到PreCapProcess( )函数、CreateBatchImageParaObj( )函数、SendData( )函数以及CloseCamera( )函数，这些函数的功能已在图5.21中做了说明，这里不做赘述，主要列出相关函数功能的代码实现如下：

**PreCapProcess( )函数**

Mind\_CameraDatasets::CameraOperationCode Mind\_CameraDatasets::PreCapProcess() {  
 MediaLibInit();//这里调用第三方库，详见产品文档"API参考"  
 // QueryCameraStatus：查询指定ID的摄像头的当前工作状态  
 // 详见API参考 > Media API参考 > 媒体库接口 > QueryCameraStatus  
 CameraStatus status = QueryCameraStatus(config\_->channel\_id);  
 if (status != CAMERA\_STATUS\_CLOSED) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess.QueryCameraStatus "  
 "{status:%d} failed.",status);  
 return kCameraNotClosed;  
 }  
  
 // Open Camera：打开指定的摄像头，在执行后续摄像头相关工作之前，需要先打开摄像头  
 // 详见文档 Atlas 200 DK > API参考 > Media API参考 > 媒体库接口 > OpenCamera  
 int ret = OpenCamera(config\_->channel\_id);  
 // 如果返回0，表示打开摄像头失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess OpenCamera {%d} "  
 "failed.",config\_->channel\_id);  
 return kCameraOpenFailed;  
 }  
  
 // SetCameraProperty：设置和修改指定摄像头的属性参数，这里设置帧率  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_FPS,  
 &(config\_->fps));  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set fps {fps:%d} "  
 "failed.",config\_->fps);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置采集数据的图片格式  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_IMAGE\_FORMAT,  
 &(config\_->image\_format));  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set image\_fromat "  
 "{format:%d} failed.",config\_->image\_format);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置图片分辨率大小  
 CameraResolution resolution;  
 resolution.width = config\_->resolution\_width;  
 resolution.height = config\_->resolution\_height;  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_RESOLUTION,  
 &resolution);  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set resolution "  
 "{width:%d, height:%d } failed.",  
 config\_->resolution\_width, config\_->resolution\_height);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置帧数据获取的方式  
 CameraCapMode mode = CAMERA\_CAP\_ACTIVE;  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_CAP\_MODE, &mode);  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set cap mode {mode:%d}"  
 " failed.",mode);  
 return kCameraSetPropeptyFailed;  
 }  
  
 return kCameraOk;  
}

**CreateBatchImageParaObj( )函数：**

shared\_ptr<BatchImageParaWithScaleT>  
// CreateBatchImageParaObj创建批量获取图片  
Mind\_CameraDatasets::CreateBatchImageParaObj() {  
 shared\_ptr < BatchImageParaWithScaleT > pobj = make\_shared<  
 BatchImageParaWithScaleT>();  
  
 pobj->b\_info.is\_first = (frame\_id\_ == kInitFrameId);  
 pobj->b\_info.is\_last = false;  
 // handle one batch every time  
 pobj->b\_info.batch\_size = 1;  
 pobj->b\_info.max\_batch\_size = kMaxBatchSize;  
 pobj->b\_info.batch\_ID = 0;  
 pobj->b\_info.channel\_ID = config\_->channel\_id;  
 pobj->b\_info.processor\_stream\_ID = 0;  
 pobj->b\_info.frame\_ID.push\_back(frame\_id\_++);  
 pobj->b\_info.timestamp.push\_back(time(nullptr));  
  
 NewImageParaT img\_data;  
 // channel begin from zero  
 img\_data.img.channel = 0;  
 img\_data.img.format = YUV420SP;  
 img\_data.img.width = config\_->resolution\_width;  
 img\_data.img.height = config\_->resolution\_height;  
 // YUV size in memory is width\*height\*3/2  
 img\_data.img.size = config\_->resolution\_width \* config\_->resolution\_height \* 3  
 / 2;  
  
 shared\_ptr <uint8\_t> data(new uint8\_t[img\_data.img.size],  
 default\_delete<uint8\_t[]>());  
 img\_data.img.data = data;  
  
 pobj->v\_img.push\_back(img\_data);  
  
 return pobj;  
}

**SendData( )函数**：

// "BatchImageParaWithScaleT" 表示 向下一个引擎中发送数据的类型为该类型  
// SendData方法中的第一个参数0，表示在本引擎中，即Camera引擎中的0号端口输出数据  
hiai\_ret = SendData(0, "BatchImageParaWithScaleT",  
 static\_pointer\_cast<void>(pobj));

**CloseCamera( )函数：**

// close camera：关闭指定的摄像头，停止图像捕捉  
CloseCamera(config\_->channel\_id);

face\_detection\_inference.cpp编写

face\_detection\_inference.cpp文件结构框架如图6.5所示，依然是包含了2个主要的函数：初始化函数和数据处理函数，其中数据处理函数主要功能结构如图6.6所示，下文将对各个函数功能的代码实现做详细呈现。



图6.5 face\_detection\_inference.cpp文件结构框架

在推理引擎中，有需要对图片数据进行预处理操作，本实验项目将预处理嵌入在了推理引擎当中，读者也可以为图片预处理操作定义一个专门的预处理引擎，即在实验步骤3流程编排的环节，设置一个图片数据预处理引擎，这里体现了软件设计模式之“单一职责原则”。本实验项目将数据预处理环节与推理过程编都统一排在了推理引擎节点，如图7.6所示，在数据处理函数功能实现框架中，执行推理过程之前，需要对图片进行相关的预处理操作。

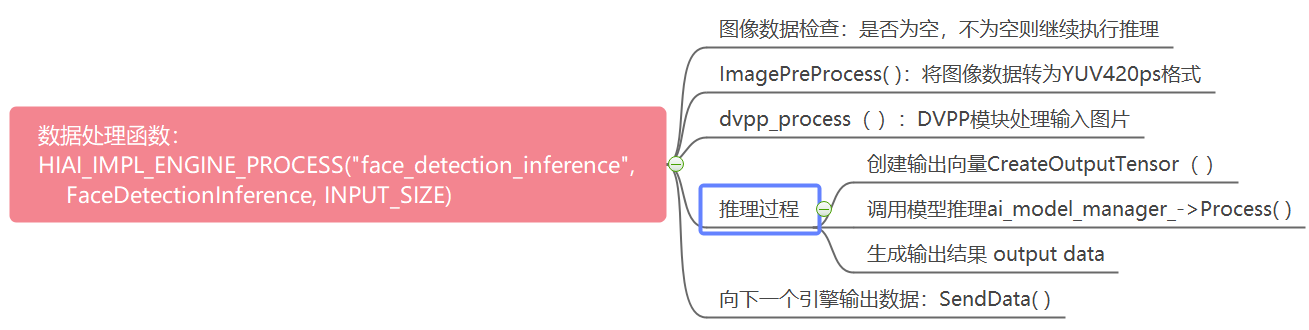


图6.6 推理引擎数据处理函数的主要功能结构

其中，推理过程的代码实现及相关注释说明如下所示：

bool face\_detection\_inference::Inference(

const ImageData<u\_int8\_t> &resized\_image,

vector<shared\_ptr<hiai::IAITensor>> &output\_data\_vec) {

// neural buffer

shared\_ptr<hiai::AINeuralNetworkBuffer> neural\_buf = shared\_ptr <

hiai::AINeuralNetworkBuffer > (

new hiai::AINeuralNetworkBuffer(),

default\_delete<hiai::AINeuralNetworkBuffer>());

neural\_buf->SetBuffer((void\*) resized\_image.data.get(), resized\_image.size);

// input data

shared\_ptr<hiai::IAITensor> input\_data = static\_pointer\_cast<hiai::IAITensor>(

neural\_buf);

vector<shared\_ptr<hiai::IAITensor>> input\_data\_vec;

input\_data\_vec.push\_back(input\_data);

// Call Process

// 1. create output tensor

hiai::AIContext ai\_context;

hiai::AIStatus ret = ai\_model\_manager\_->CreateOutputTensor(input\_data\_vec,

output\_data\_vec);

// create failed

if (ret != hiai::SUCCESS) {

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

"call CreateOutputTensor failed");

return false;

}

// 2. process

HIAI\_ENGINE\_LOG("aiModelManager->Process start!");

ret = ai\_model\_manager\_->Process(ai\_context, input\_data\_vec, output\_data\_vec,

AI\_MODEL\_PROCESS\_TIMEOUT);

// process failed, also need to send data to post process

if (ret != hiai::SUCCESS) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT, "call Process failed");

return false;

}

HIAI\_ENGINE\_LOG("aiModelManager->Process end!");

return true;

}

face\_emotion\_inference.cpp文件

face\_detection\_inference.cpp文件结构框架如图6.7所示，依然是包含了2个主要的函数：初始化函数和数据处理函数，其中数据处理函数主要功能结构如图6.8所示，下文将对各个函数功能的代码实现做详细呈现。



图6.7 face\_emotion\_inference.cpp文件结构框架

在推理引擎中，人脸表情识别接收上一个引擎发来的人脸图片数据，然后将图片经过预处理转换为推理需要的图像格式，最后得出推理结果，如图7.8所示，在数据处理函数功能实现框架中，执行推理过程之前，需要对图片进行相关的预处理操作。

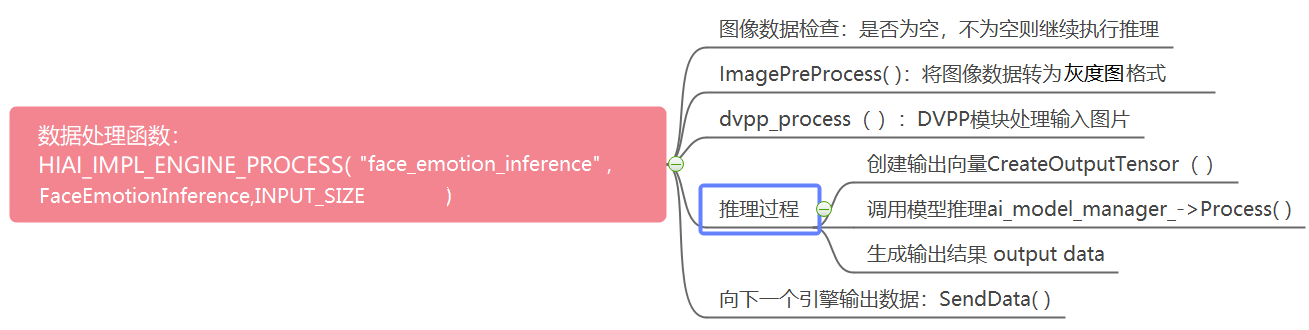


图6.8 推理引擎数据处理函数的主要功能结构

其中，推理过程的代码实现及相关注释说明如下所示：

//-- region  BGRInfarence

bool face\_emotion\_inference::Inference(vector<Mat> &normalized\_image,

                                       vector<FaceImage> &face\_imgs) {

  // Define the ai model's data

  AIContext ai\_context;

  int normalized\_image\_size = normalized\_image.size();

  int normalized\_image\_mod = normalized\_image\_size % batch\_size\_;

  // calcuate the iter number

  // calcuate the value by batch

  int iter\_num = normalized\_image\_mod == 0 ?

                 (normalized\_image\_size / batch\_size\_) : (normalized\_image\_size / batch\_size\_ + 1);

  // Invoke interface to do the inference

  for (int i = 0; i < iter\_num; i++) {

    HIAI\_ENGINE\_LOG("Batch data's number is %d!", i);

    int start\_index = batch\_size\_ \* i;

    int end\_index = start\_index + batch\_size\_;

    // Last group data, need to fulfill the extra data

    // fulfill with last Mat in the vector

    if (i == iter\_num - 1 && normalized\_image\_mod != 0) {

      int fulfill\_number = batch\_size\_ - normalized\_image\_mod;

      EnrichDataByLastMat(normalized\_image, fulfill\_number);

      end\_index = i \* batch\_size\_ + normalized\_image\_mod;

    }

    float \*tensor\_buffer = new(std::nothrow) float[batch\_size\_ \* kResizedImgWidth \* kResizedImgHeight \* kRgbChannel];

    if (tensor\_buffer == nullptr) {

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

                      "New the tensor buffer error.");

      return false;

    }

    printf("CopyDataToBuffer...\n");

    int last\_size = CopyDataToBuffer(normalized\_image, start\_index, tensor\_buffer);

    if (last\_size == -1) {

      return false;

    }

    vector<shared\_ptr<IAITensor>> input\_data\_vec;

    shared\_ptr<AINeuralNetworkBuffer> neural\_buffer = shared\_ptr <

        AINeuralNetworkBuffer > (new(std::nothrow) AINeuralNetworkBuffer());

    shared\_ptr<IAITensor> input\_data = static\_pointer\_cast <

                                       IAITensor > (neural\_buffer);

    neural\_buffer->SetBuffer((void \*)tensor\_buffer,  last\_size \* sizeof(float));

    input\_data\_vec.push\_back(input\_data);

    printf("last\_size=%d...\n",last\_size);

    printf("CreateOutputTensor...\n");

    vector<shared\_ptr<IAITensor>> output\_data\_vec;

    AIStatus ret = ai\_model\_manager\_->CreateOutputTensor(input\_data\_vec, output\_data\_vec);

    if (ret != SUCCESS) {

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

                      "Fail to create output tensor");

      delete [] tensor\_buffer;

      return false;

    }

    printf("Process...\n");

    ret = ai\_model\_manager\_->Process(ai\_context, input\_data\_vec, output\_data\_vec, 0);

    if (ret != SUCCESS) {

      printf("Fail to process the data in FWK\n");

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

                      "Fail to process the data in FWK\n");

      delete [] tensor\_buffer;

      return false;

    }

    std::cout<<"output data vec size:"<<output\_data\_vec.size();

    printf("Inference successed...\n");

    HIAI\_ENGINE\_LOG("Inference successed!\n");

    // Get the inference result.

    shared\_ptr<AISimpleTensor> result\_tensor = static\_pointer\_cast <

        AISimpleTensor > (output\_data\_vec[kResult1Index]);//第0路输出 136个float

   // shared\_ptr<hiai::AISimpleTensor> result\_tensor1 = static\_pointer\_cast <

      //  AISimpleTensor > (output\_data\_vec[kResult2Index]);//第1路输出 3个float

    printf("copy data to float array...\n");

    // copy data to float array

    int32\_t  size= result\_tensor->GetSize() / sizeof(float);

   // int32\_t size1 = result\_tensor1->GetSize() / sizeof(float);

    float result[size];

    printf("size=%d...\n",size);

    errno\_t mem\_ret = memcpy\_s(result, sizeof(result), result\_tensor->GetBuffer(),

                                result\_tensor->GetSize());

    if (mem\_ret != EOK) {

      printf("post process call memcpy\_s() error=%d\n", mem\_ret);

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

                      "post process call memcpy\_s() error=%d", mem\_ret);

      return false;

    }

    printf("copy result start\n");

    for (int k = 0; k < 7; k++)

    {

      face\_imgs[i].result[k] = result[k];

      }

      printf("copy result over\n");

      printf("%f=%f=%f=%f=%f=%f=%f", result[0], result[1], result[2], result[3], result[4], result[5], result[6]);

      input\_data\_vec.clear();

      output\_data\_vec.clear();

      delete[] tensor\_buffer;

  }

  return true;

}

face\_emotion\_postprocess.cpp文件

本实验项目的后处理引擎，是在接收到人脸表情识别推理引擎发送的数据后，整理数据格式并调用PresenterServer的接口，进行检测结果在主机侧的回显。如同上述2）、3）所述文件结构，face\_emotion\_postprocess.cpp文件结构框架也有2个，如图6.9所示，其中，初始化函数主要是为了获取图配置信息，建立PresenterServer链接，便于检测结果的回显。

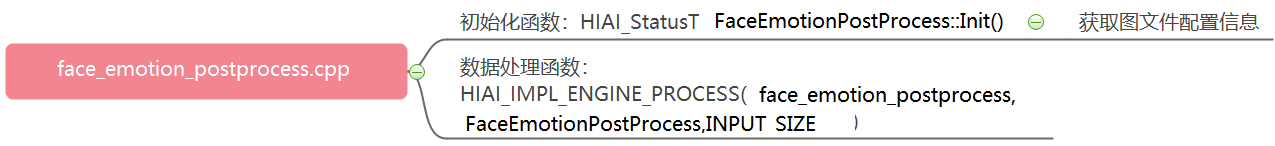


图6.9 face\_emotion\_postprocess.cpp文件结构框架



图6.10 后处理引擎数据处理函数的主要功能结构

后处理引擎数据处理函数HIAI\_IMPL\_ENGINE\_PROCESS( )代码实现如下所示：

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;

}

// check original image is empty or not

std::shared\_ptr<FaceRecognitionInfo> inference\_res = std::static\_pointer\_cast<

FaceRecognitionInfo>(arg0);

// inference success, dealing inference results

return HandleResults(inference\_res);

//return HIAI\_OK;

}

这里，主要介绍推理成功后，处理推理结果的函数代码实现，如下所示：

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;

      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;

}

处理推理成功的检测结果时，这里的SendImage( )不是向下一个引擎发送数据，而是将检测结果中的关键数据向PresenterServer发送，在主机侧接收SendImage( )的数据后，利用坐标、置信度、属性等数据进行检测结果的可视化，本实验项目是利用PresenterServer实现检测结果的显示，读者也可根据实际情况，自定义人脸检测结果的显示方式，例如直接将检测数据结果打印在终端里显示，或者将检测数据结果保存在文件中。SendImage( )函数具体实现代码如下所示：

// 函数接口参数：图片的高、宽、尺寸、数据

int32\_t face\_emotion\_postprocess::SendImage(uint32\_t height, uint32\_t width,

                                            uint32\_t size, u\_int8\_t \*data, std::vector<DetectionResult>& detection\_results) {

  int32\_t status = kFdFunSuccess;

  // parameter

  ImageFrame image\_frame\_para;

  image\_frame\_para.format = ImageFormat::kJpeg;

  image\_frame\_para.width = width;

  image\_frame\_para.height = height;

  image\_frame\_para.size = size;

  image\_frame\_para.data = data;

  image\_frame\_para.detection\_results = detection\_results;

  PresenterErrorCode p\_ret = PresentImage(presenter\_channel\_.get(),

                                            image\_frame\_para);

  // send to presenter failed

  if (p\_ret != PresenterErrorCode::kNone) {

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

                      "Send JPEG image to presenter failed, error code=%d",

                      p\_ret);

    status = kFdFunFailed;

  }

  return status;

}

至此，在主机侧的项目代码编写实现的大体思路已介绍完毕，编译的过程在终端执行对每个.pp文件的make指令即可完成应用代码的编译，接下来需要将本实验项目用到的模型进行转换，以便后续部署在Atlas开发板上，进行人脸检测推理。

6.2模型转换

该任务将Caffe模型转换为Ascend 310芯片支持的Davinci模型文件，首先需要通过Mind Studio创建模型转换工程，通过转换模型功能，实现人脸检测模型的转换。

步骤1 创建模型转换工程项目

从Web端输入<https://hostIP:Port>，登录MindStudio，以本机的IP信息为例：浏览器输入<https://192.168.1.10:8888>，注意要用https协议进行访问，并加上端口号8888，登录Mind Studio界面的用户名默认为“MindStudioAdmin”，不支持修改和新建。初始密码为“Huawei123@，输入密码登录首次要求修改密码，输入修改后的密码即可登入；

登入Mind Studio之后，依次选择“File -> New -> New Project”，弹出“New Project”窗口，创建一个新的工程，显示窗口配置界面，并依次设置好工程项目初始配置，点击“create”完成创建，配置信息如图6.11所示。

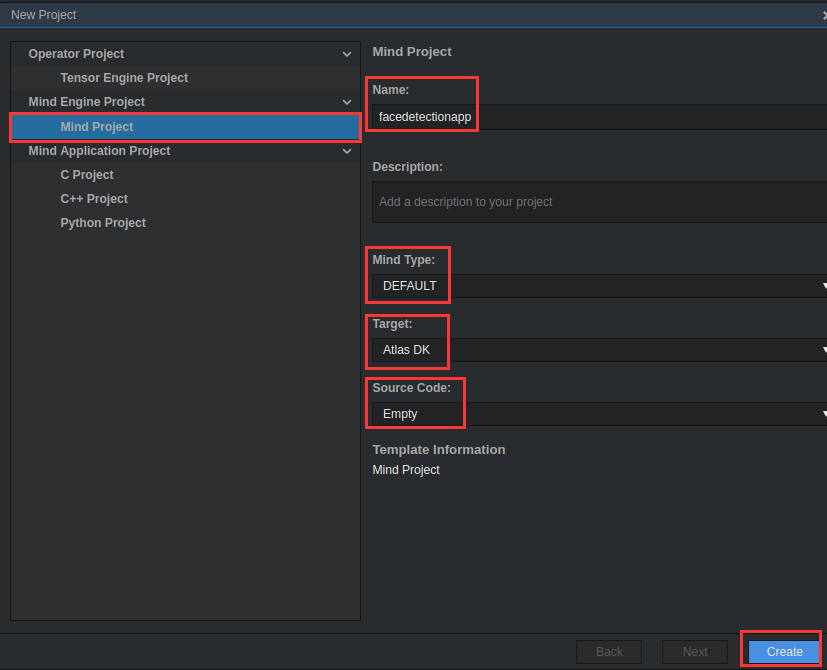


图6.11 创建工程项目配置

此工程示例中“Mind Type”选择“DEFAULT”，“Target”选择“Atlas DK”（开发者板），“Source Code”选择“Empty”。具体创建工程的设置参数如表6.1所示。

表6-1 创建工程设置的参数说明表

|  |  |
| --- | --- |
| 参数 | 参数说明 |
| Name | 工程名称，自行配置。  名字需要为字符串，不包含空格，空格会自动填充为“-”。 |
| Mind Type | 包含如下两种选项：  DEFAULT：创建的工程会生成画布，用户可以使用拖拽的方式进行工程编排。  CUSTOM：自定义工程，不会生成画布。 |
| Target | 选择运行环境：  Local：仿真环境，用于caffe推理引擎。  ASIC：连接EVB、PCIe单板。  Atlas DK：连接开发者板。 |
| Source Code | 代码来源：  Empty：选择该项，表示工程非外部导入。  Local(Web Client)：从本地Windows中导入源文件，选择该项，下方出现文件上传输入框。  Local(Web Server)：从后端服务器导入源文件，选择该项，则下方出现输入框填写Mind Studio服务器端代码路径。 |

步骤2 模型转换

通常可以在ModelArts平台上完成所需深度学习模型的生成，再将训练好的模型导入Mind Studio中进行模型转换，获得Ascend 310芯片支持的Davinci模型文件。

在本项目中，人脸检测模型是利用官方提供的已有caffe模型，人脸表情网络模型是训练得到的Caffe模型，因此，需要将人脸检测和人脸表情识别的Caffe模型转换为Davinci模型文件，操作步骤如下：

1. 获取人脸检测模型的网络和权重文件：

prototxt网络文件下载：从https://bbs.huaweicloud.com/forum/thread-22650-1-1.html处获取已验证的Prototxt下载地址文件，，在.xlsx文件中下载本项目所需的face\_detection.prototxt文件（注意：需要在已验证prototxt文件路径进行下载）。

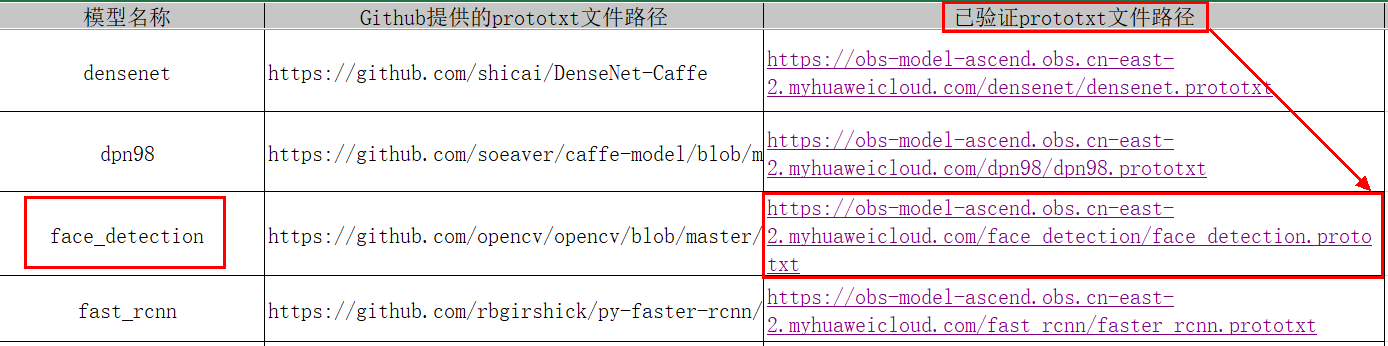
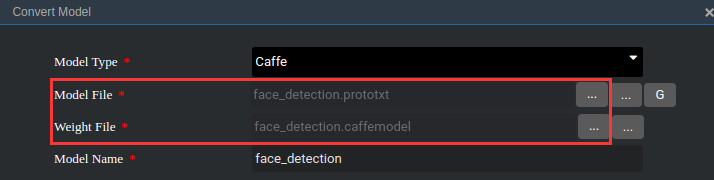


图6.12 已验证prototxt文件下载路径选择

权重文件下载：从https://obs-model-ascend.obs.cn-east-2.myhuaweicloud.com/face\_detection/face\_detection.caffemodel 下载权重文件。

1. 在步骤1创建的“faceemotionnapp”工程目录中，先选中faceemotionapp工程目录，再右击选择“Convert Model”，或者先选中faceemotionapp工程目录，在菜单栏依次选择“Tools > Convert Model”，进入模型转换界面，设置模型转换的配置参数，单击OK进行模型转换，如图6.13所示：



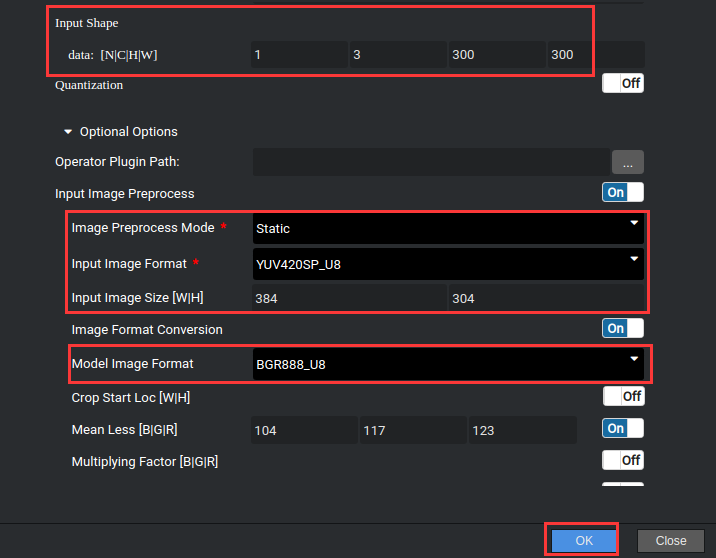


图6.13 离线模型转换示意图

1. 转换时会出现如下报错信息，此时在DetectionOutput层的Suggestion中选择SSDDetectionOutput，并点击Retry，如图6.14所示。

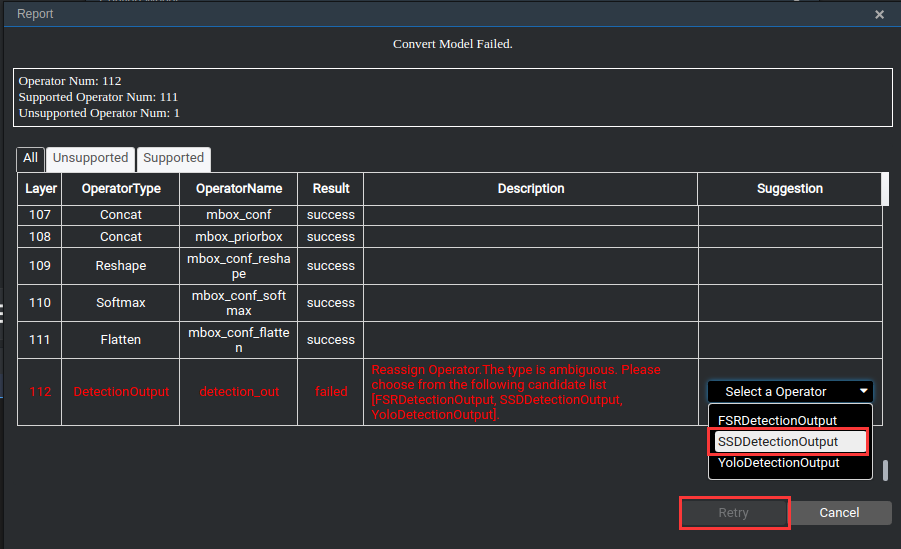


图6.14 离线模型转换报错后调整

1. 模型转换成功后显示如图6.15所示，模型转换成功后，转换好的后缀为.om的模型文件存放地址为：$HOME/tools/che/model-zoo/my-model/face\_detection/device：

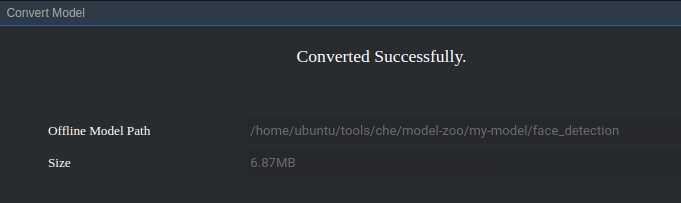


图6.15 离线模型转换成功

1. 模型转化成功后，在“model-zoo > my-model”中可以看到转化成功的离线模型文件目录，如图6.16所示。

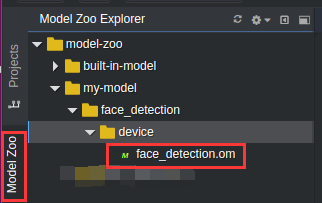


图6.16 人脸检测转化好的离线模型

1. 模型放至指定路径：将转换好的模型文件（.om文件）拷贝到人脸检测源码所在路径“./script”目录下，用于运行部署脚本时将模型文件部署在开发板的环境中。
2. 经过类似于人脸检测步骤，我们可以直接转换人脸表情识别的模型为OM模型，参数转换图如6.17所示：

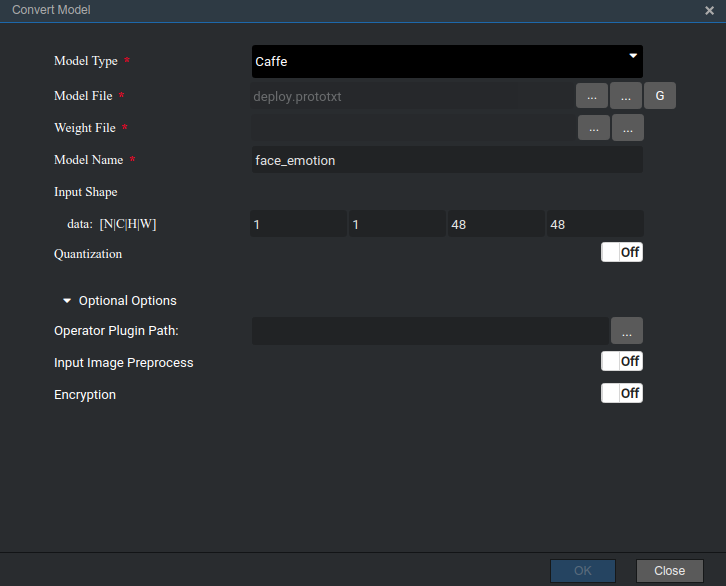


图6.17 人脸表情识别转化参数设置

7.3 开发板部署执行

步骤1设置环境变量

在安装了Mind Studio所在的Ubuntu主机侧，设置环境变量DDK\_HOME，在终端中输入如下指令：

**vim ~/.bashrc**

执行该命令后，在弹出的新窗口中，按下键盘字母“i”，通过键盘上的向下箭头按键移动光标至文本的最后一行，并在最后一行添加DDK\_HOME及LD\_LIBRARY\_PATH的环境变量，需要添加的内容如下：

**export DDK\_HOME=/home/XXX/tools/che/ddk/ddk**

**export LD\_LIBRARY\_PATH=$DDK\_HOME/uihost/lib**

说明：XXX为安装Mind Studio所在的服务器的用户名，/home/XXX/tools为DDK默认的安装路径。（如果此环境变量已经添加，则此步骤可跳过。）

完成文本编辑后，键盘按下“**esc**”按键，再继续键盘输入 **:wq!** 保存退出，即完成了环境变量配置。

执行如下命令使环境变量生效。

**source ~/.bashrc**

步骤2 移植应用案例

1. 进入facedetectionapp应用代码所在根目录，例如：

/home/ubuntu/Downloads/sample-facedetection。

1. 执行部署脚本，将人脸检测应用案例移植到Atlas开发板上，并进行工程环境准备，包括ascenddk公共库的编译与部署、Presenter Server服务器的配置等操作，其中Presenter Server用于接收人脸检测Application发送过来的数据并通过浏览器进行结果展示。执行如下指令：

**bash deploy.sh host\_ip model\_mode**

其中，host\_ip指Atlas 200 DK开发者板的IP地址。model\_mode代表模型文件及依赖软件的部署方式，默认为internet，在线下载依赖代码库。

指令示例如下所示：

**bash deploy.sh 192.168.1.2 internet**

当提示“Please choose one to show the presenter in browser(default: 127.0.0.1):“时，请输入在浏览器中访问Presenter Server服务所使用的IP地址，这里可直接输入127.0.0.1。

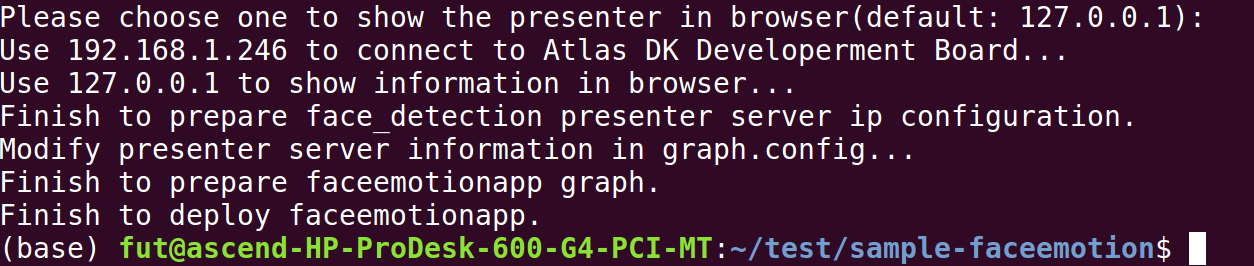


图6.18 工程部署完成

如图6.19所示，在“Current environment valid ip list“中选择浏览器访问Presenter Server服务使用的IP地址，默认IP为127.0.0.1。

3）启动Presenter Server。

执行如下命令在后台启动Face Detection应用的Presenter Server主程序。

**python3 presenterserver/presenter\_server.py --app face\_detection &**

如图7.19所示，表示presenter\_server的服务启动成功。

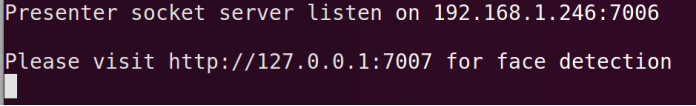


图6.19 Presenter Server进程启动

使用上图提示的URL登录Presenter Server，这里支持Chrome浏览器和火狐浏览器，在浏览器中输入IP地址和端口号，其中，IP地址为2）中输入的IP地址，端口号默为7007，如图6.20所示，表示Presenter Server启动成功。

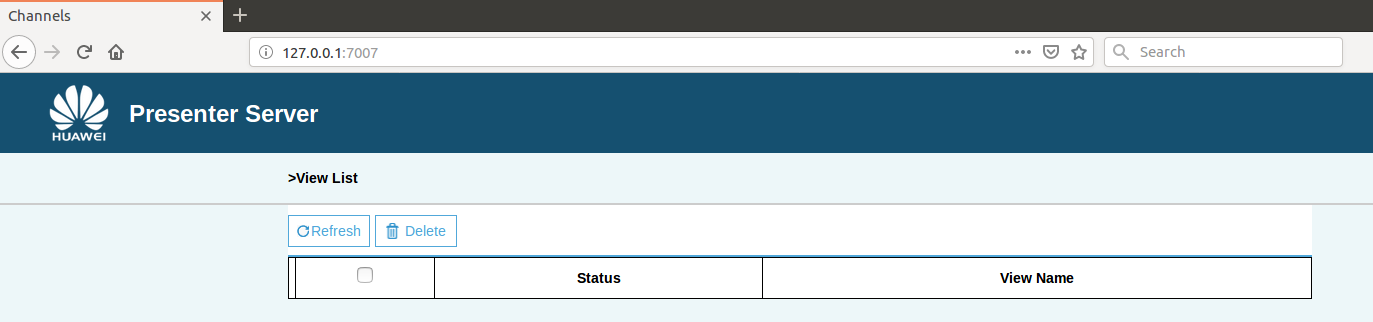


图6.20 Presenter Server主页显示

步骤3 查看摄像头所属Channel

为了使用镜头在开发板上进行数据的采集，需要获取摄像头在Atlas 200 DK开发板上所属的通道编号，此时需要借助Mind Studio工具进行查阅。

在Mind Studio界面中单击“Tools > Atlas DK Configuration”，进入“Atlas DK Configuration”界面：

若开发板还未与Mind Studio连接，可参照产品文档中“在Mind Studio中添加Atlas 200 DK开发者板”一节 ( <https://ascend.huawei.com/doc/atlas200dk/1.3.0.0/zh/zh-cn_topic_0195272402.html> ) ，将开发板与Mind Studio相连。

若已配置好开发板与MindStudio连接，此时只需选中使用的开发者板，单击“Connect”，查看开发者板连接状态，如图6.21所示，本例中，摄像头Camera1显示“online”状态，可确定摄像头所属Channel为1。

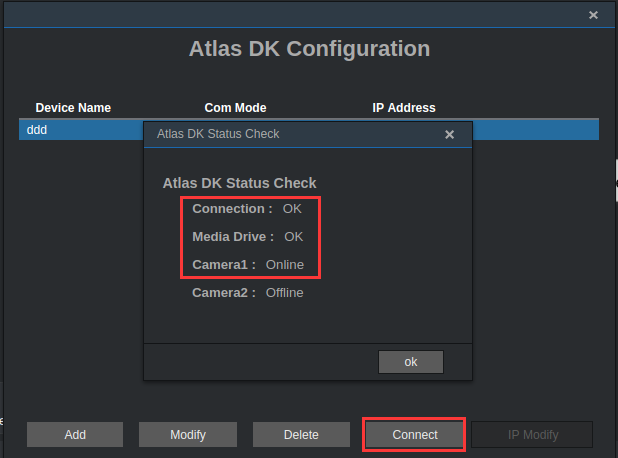


图6.21 Atlas 200 DK状态界面

步骤4 执行人脸表情识别应用

1. 运行Face Emotion程序。

在sample-faceemotion目录下执行如下命令运行Face Emotion应用程序。

**bash run\_faceemotionapp.sh host\_ip presenter\_view\_app\_name camera\_channel\_name &**

说明：

**host\_ip：**对于Atlas 200 DK开发者板，即为开发者板的IP地址。

**presenter\_view\_app\_name：**用户自定义的在PresenterServer界面展示的View Name，此View Name需要在Presenter Server展示界面唯一，只能是大小写字母、数字、“\_”的组合，位数3-20。

**camera\_channel\_name：**摄像头所属Channel，取值为“Channel-1“或者“Channel-2“。

命令示例如下：

**bash run\_faceemotionapp.sh 192.168.1.2 video Channel-1 &**

1. 使用启动Presenter Server服务时提示的URL登录 Presenter Server 网站，等待Presenter Agent传输数据给服务端，单击“Refresh“刷新，当有数据时相应的Channel 的Status变成绿色，如图6.22所示。

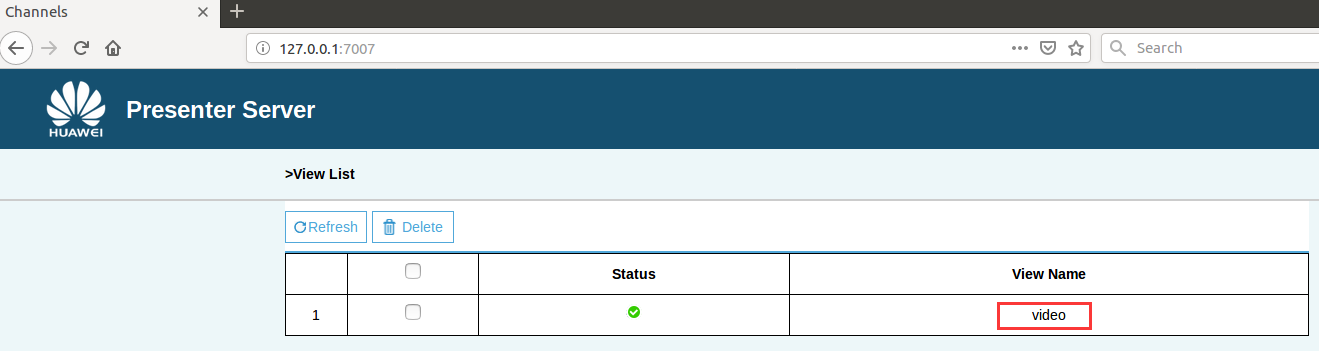


图6.22 有视频数据输入的Presenter Server界面

1. 单击右侧对应的View Name链接，比如图6.23的“video”，查看结果，对于检测到的人脸表情，会给出标注结果，人脸表情识别实验效果如图6.23所示。



图6.23 人脸表情是被结果显示示例

7 实验小结

本实验完成了基于Atlas 200 DK开发板的人脸表情识别应用案例移植实践，开发者可以将本应用部署至Atlas 200DK上，实现对摄像头数据的实时采集、并对视频中的人脸表情信息进行预测的功能。其中，主要涉及的功能代码函数、目标模型转换、环境部署编译等流程总结如图7.1所示：

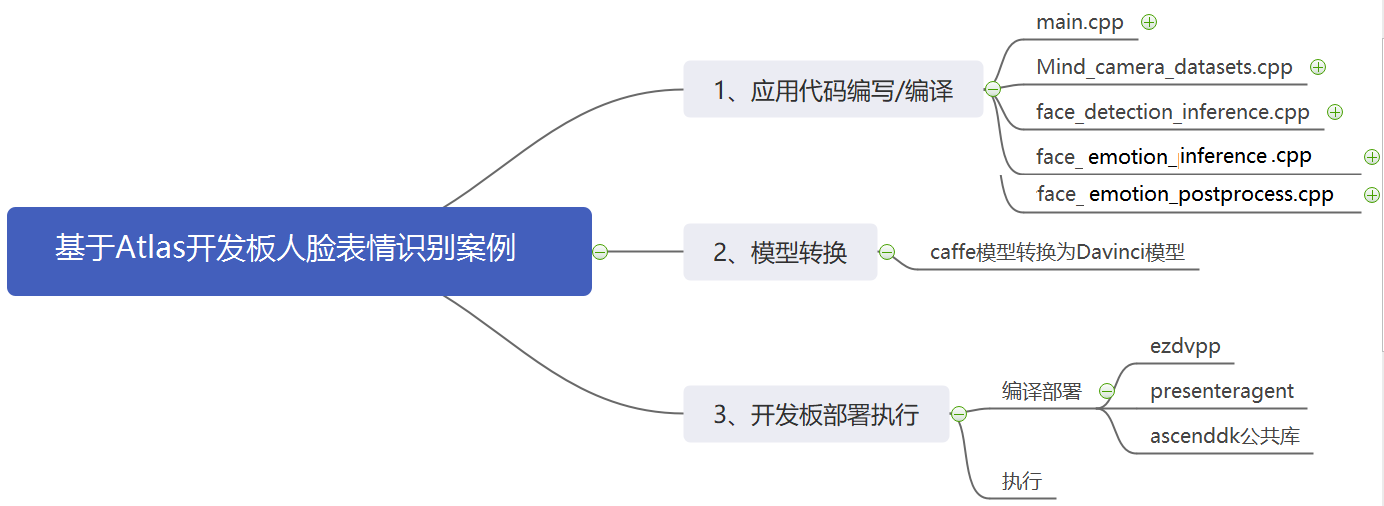


图7.1 实验小结导图