# RT-AK 教育套件实验手册



**WWW.RT-THREAD.ORG**

**Tuesday 23rd November, 2021**

# 目录

# 第 1 章

# 实验七人脸关键点检测

# 第 2 章

# 实验介绍和目的

## 1.1 实验介绍

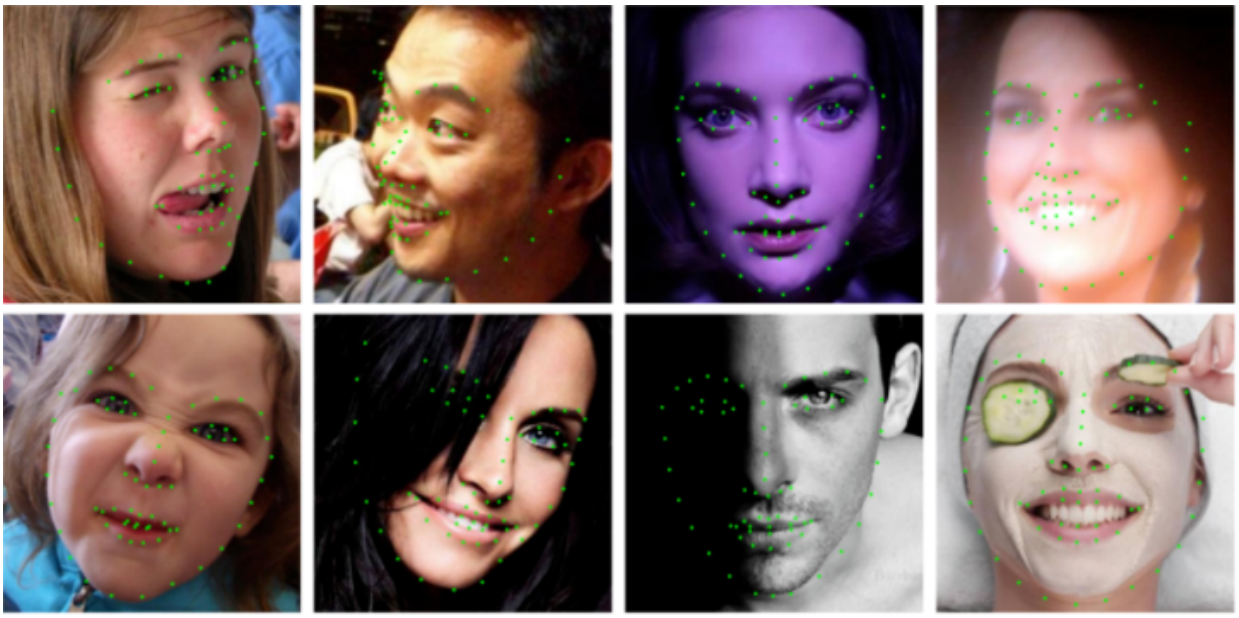
本实验是基于 Tensorflow 训练的第七个 AI 模型：人脸关键点检测。

基于深度学习的人脸关键点检测程序，可以应用在表情识别等，在日常生活中的使用非常广泛。

人脸关键点检测的工作流程可以分为两部分：

- 面部/非面部分类
- 人脸关键点回归

人脸检测和对齐虽然基础，但对于许多面部其它应用至关重要，例如面部识别和面部表情分析。实际应用中会存在遮挡、大的姿态变化和极端明暗变化，对这些任务构成了巨大的挑战。



## 1.2 实验目的

1. 进一步掌握 RT-AK 命令部署 AI 模型的使用

2. 学习卷积神经网络
3. 实现端到端的人脸特征点检测任务

# 第 3 章

# 实验器材

1. 上位机（电脑）
2. EgdeAI 实验板
3. Camera 使用的为 GC0308
4. LCD 驱动 IC 为 ILI9341

# 第 4 章

# 实验步骤

确保环境安装无问题之后做后续实验，环境安装请参考以往实验教程

## 3.1 AI 模型训练

本节模型是参考 RetinaFace 模型进行开发的，由于模型较大，本教程只给出模型主要部分，对模型训练感兴趣的同学可以参考具体的 RetinaFace 项目。

RetinaFace 项目地址：https://github.com/deepinsight/insightface

RetinaFace 论文地址：https://arxiv.org/abs/1905.00641

1. 加载模型和权重

这一部分是模型的主干，主要是加载模型和权重，并将模型的结构打印出来

```
#-------------------------------------#
#   载入模型与权值
#   请注意主干网络与预训练权重的对应
#-------------------------------------#
model = RetinaFace(cfg)
model.summary()
model_path = "model_data/retinaface_mobilenet025.h5"
model.load_weights(model_path, by_name=True, skip_mismatch=True)

#-------------------------------#
#   获得先验框和工具箱
#-------------------------------#
anchors = Anchors(cfg, image_size=(img_dim, img_dim)).get_anchors()
bbox_util = BBoxUtility(anchors)

#---------------------------------------------------------------------------------#
#   训练参数的设置
#   logging表示tensorboard的保存地址
#   checkpoint用于设置权值保存的细节，period用于修改多少epoch保存一次
```

```
#   reduce_lr用于设置学习率下降的方式
#   early_stopping用于设定早停，val_loss多次不下降自动结束训练，表示模型基本收敛
#-------------------------------------------------------------------------------#
logging = TensorBoard(log_dir="logs/")
checkpoint = ModelCheckpoint('logs/ep{epoch:03d}-loss{loss:.3f}.h5',
    monitor='loss', save_weights_only=True, save_best_only=False, period=1)
reduce_lr = ExponentDecayScheduler(decay_rate=0.92, verbose=1)
early_stopping = EarlyStopping(monitor='loss', min_delta=0, patience=6, verbose=1)
loss_history = LossHistory("logs/")

for i in range(freeze_layers): model.layers[i].trainable = False
print('Freeze the first {} layers of total {} layers.'.format(freeze_layers, len(
    model.layers)))
```

2. 训练模型

这一部分的工作是根据模型本身的特性进行部分训练还是全部训练，本例程由于使用了 mobilenet 主干网络，所以采用了预训练权重，所以训练分为两部分。

```
#-------------------------------------------------------#
#   主干特征提取网络特征通用，冻结训练可以加快训练速度
#   也可以在训练初期防止权值被破坏。
#   Init_Epoch为起始世代
#   Freeze_Epoch为冻结训练的世代
#   Epoch总训练世代
#   提示OOM或者显存不足请调小Batch_size
#-------------------------------------------------------#
if True:
    batch_size          = 4
    Init_epoch          = 0
    Freeze_epoch        = 10
    learning_rate_base  = 1e-3

    gen                 = Generator(training_dataset_path,img_dim,batch_size,
        bbox_util)
    epoch_size          = gen.get_len() // batch_size
    print('Train on {} samples, with batch size {}.'.format(epoch_size, batch_size))
    if eager:
        gen         = tf.data.Dataset.from_generator(partial(gen.generate), (tf.
            float32, tf.float32, tf.float32, tf.float32))
        gen         = gen.shuffle(buffer_size=batch_size).prefetch(buffer_size=
            batch_size)

        lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
            initial_learning_rate=learning_rate_base, decay_steps=epoch_size,
                decay_rate=0.95, staircase=True
        )

        optimizer   = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
```

```
        for epoch in range(Init_epoch,Freeze_epoch):
            fit_one_epoch(model, optimizer, epoch, epoch_size, gen, Freeze_epoch,
                get_train_step_fn())
    else:
        model.compile(loss={
                    'bbox_reg'  : box_smooth_l1(weights=cfg['loc_weight']),
                    'cls'       : conf_loss(),
                    'ldm_reg'   : ldm_smooth_l1()
                },optimizer=keras.optimizers.Adam(lr=learning_rate_base)
        )

        model.fit(gen,
                steps_per_epoch=epoch_size,
                verbose=1,
                epochs=Freeze_epoch,
                initial_epoch=Init_epoch,
                # 开启多线程可以加快数据读取的速度。
                # workers=4,
                # use_multiprocessing=True,
                callbacks=[logging, checkpoint, reduce_lr, early_stopping,
                    loss_history])

for i in range(freeze_layers): model.layers[i].trainable = True

if True:
    batch_size          = 4
    Freeze_epoch        = 10
    Epoch               = 20
    learning_rate_base  = 1e-4

    gen                 = Generator(training_dataset_path,img_dim,batch_size,
        bbox_util)
    epoch_size          = gen.get_len() // batch_size
    print('Train on {} samples, with batch size {}.'.format(epoch_size, batch_size))
    if eager:
        gen         = tf.data.Dataset.from_generator(partial(gen.generate), (tf.
            float32, tf.float32, tf.float32, tf.float32))
        gen         = gen.shuffle(buffer_size=batch_size).prefetch(buffer_size=
            batch_size)

        lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
            initial_learning_rate=learning_rate_base, decay_steps=epoch_size,
                decay_rate=0.95, staircase=True
        )

        optimizer   = tf.keras.optimizers.Adam(learning_rate=lr_schedule)

        for epoch in range(Freeze_epoch,Epoch):
```

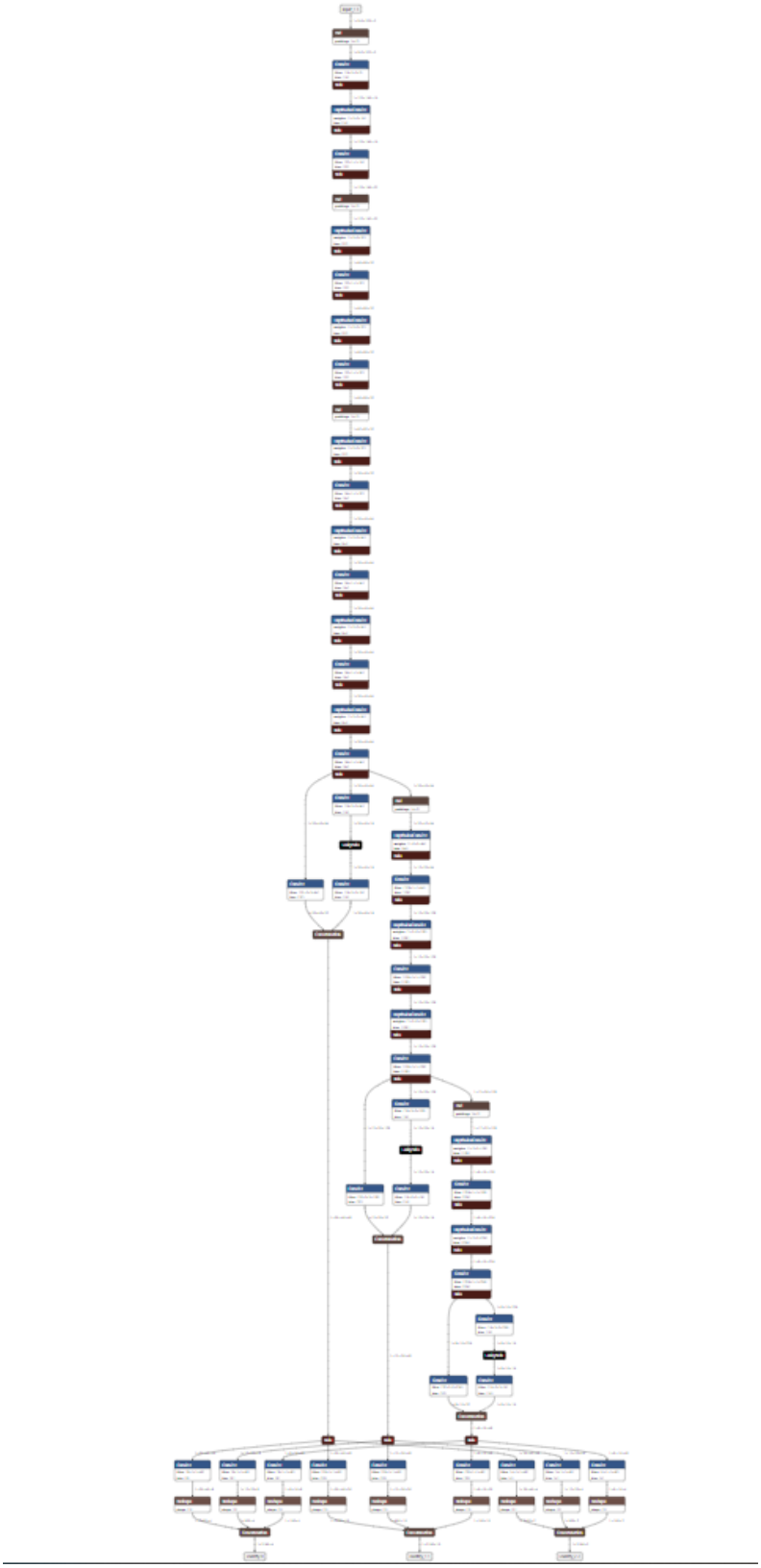
```
            fit_one_epoch(model, optimizer, epoch, epoch_size, gen, Epoch,
                get_train_step_fn())
    else:
        model.compile(loss={
                    'bbox_reg'  : box_smooth_l1(weights=cfg['loc_weight']),
                    'cls'       : conf_loss(),
                    'ldm_reg'   : ldm_smooth_l1()
                },optimizer=keras.optimizers.Adam(lr=learning_rate_base)
        )

        model.fit(gen,
                steps_per_epoch=epoch_size,
                verbose=1,
                epochs=Epoch,
                initial_epoch=Freeze_epoch,
                # 开启多线程可以加快数据读取的速度。
                # workers=4,
                # use_multiprocessing=True,
                callbacks=[logging, checkpoint, reduce_lr, early_stopping,
                    loss_history])
```

3. 模型信息：

- tflite 模型文件大小为 1481KB，适合部署至 K210 来执行嵌入式推理工作
- 模型输入：图像，[height, width, channel] –> 320x240x3，uint8
- 模型三个输出

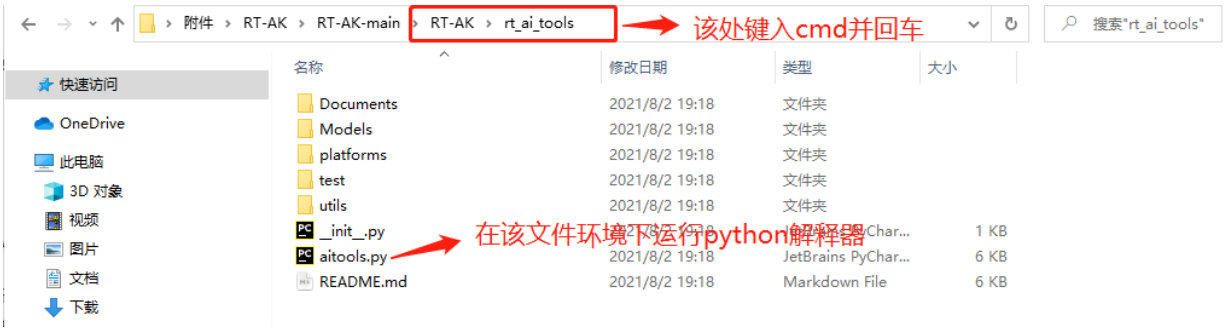
    1. 输出 1，面部坐标: [1, 3160, 4]
    2. 输出 2，人脸特征点坐标: [1, 3160, 10]
    3. 输出 3，面部/非面部分类: [1, 3160, 2]

使用 Netron 查看网络模型结构：具体可以参考文件夹 Models 中的模型文件



## 3.2 模型部署

在 RT-AK/rt_ai_tools 路径下打开 Windows 命令行终端（确保目标文件夹下面包含aitools.py文件）




在 Windows 输入以下命令 (如何获取量化数据请查阅附录文档)：

```
# 量化为 uint8，使用 KPU 加速，量化数据集为图片
$ python aitools.py --project=<your_project_path> --model=<your_model_path> --
    model_name=facelandmark --platform=k210 --dataset=<your_val_dataset>
```

--model_name=facelandmark为了匹配实验中的示例代码，不要随意更改。下面是自己构建工程时使用的示例，真实实验过程中需要注意自己文件所在的文件夹

```
# 示例(量化模型，图片数据集)
$ python aitools.py --project="xxx\lab7-face_landmark\xxx" --model="xxx\lab7-
    face_landmark\Models\face_landmark.tflite" --model_name=facelandmark --platform=
    k210 --dataset="xxx\lab7-face_landmark\Datasets\quantize_data"
```

其中，--project 是目标工程路径，--model 是目标模型路径，--model_name 是转化模型的名字，--platform 是指定插件支持的目标平台为 k210，--dataset 是模型量化所需要用到的数据集。

更多详细的参数信息请看文档: RT-AK\rt_ai_tools\platforms\plugin_k210\README.md。

当部署成功之后，目标工程文件会多出以下几个文件：

| 文件 | 描述 |
| --- | --- |
| rt_ai_lib/ | RT-AK Libs，模型推理库 |
| applications/facelandmark_kmodel.c | kmodel 的十六进制储存 |
| applications/rt_ai_facelandmark_model.c | 与目标平台相关的信息 |
| applications/rt_ai_facelandmark_model.h | 模型相关信息 |

同时，在 RT-AK\rt_ai_tools\platforms\plugin_k210 路径下会生成两个中间文件

| 文件 | 描述 |
| --- | --- |
| facelandmark.kmodel | k210 所支持的模型格式 |
| convert_report.txt | tflite 模型转成 kmodel 格式的缓存信息 |

| 文件 | 描述 |
| --- | --- |

如果不想生成上述两文件，可以在模型部署的时候在命令行参数末尾加上：`--clear`，则在部署过程中即可删除相关中间文件

注意：

1、RT-AK 部署成功后不会产生应用代码，比如模型推理代码，需要在嵌入式项目工程中手工编写相关代码，详见 “3.3 嵌入式 AI 模型应用”

2、在应用开发过程中，请遵守 RT-Thread 的编程规范以及 API 使用标准

## 3.3 嵌入式 AI 模型应用

使用 RT-AK 将训练好的 tflite 模型成功部署到工程之后，我们就可以开始着手编写应用层代码来使用该模型。本节的所有代码详见文件夹 `Applications`内部的文件

### 3.3.1 代码流程

**系统内部初始化：**

- kpu 时钟初始化
- 相机初始化
- LCD 屏幕初始化

**RT-AK Lib 模型加载并运行：**

- 注册模型（代码自动注册，无需修改）
- 查找注册模型
- 初始化模型，挂载模型信息，准备运行环境
- 运行（推理）模型
- 获取输出结果
- 模型后处理

**人脸特征点分类业务逻辑层：**

- 输出三组模型推理结果，并进行分析。

### 3.3.2 核心代码说明

```
void _thread_facelandmark(void *params)
{
    rt_kprintf("Hello, world\n");                   //打印hello world
    volatile uint32_t g_ai_done_flag;               //AI推理完成标志位
```

```
    static uint8_t *kpu_image;
    static uint8_t *display_image;
    static facelandmark_region_layer_t rl;
    static facelandmark_box_info_t boxes;
    static float variances[2]= {0.1, 0.2};
    static float *pred_box, *pred_landm, *pred_clses;
    static size_t pred_box_size, pred_landm_size, pred_clses_size;
    rt_ubase_t mb_value = 0;
    rt_ai_t face_landmark = NULL;

    /* LCD init */
    rt_kprintf("LCD init\n");
    lcd_init();
    lcd_clear(BLACK);

    /* read一帧图像,显示与AI图像会按顺序连续排放在buffer中 */
    display_image = rt_malloc((240 * 320 * 2)+(240 * 320 * 3));
    kpu_image = display_image + (240 * 320 * 2);                    //接着显示之后存放
        ,注意地址偏移
    rt_device_t camera_dev = rt_device_find(CAMERA);                //查找摄像头设备,
        CAMERA="ov2640" OR "ov5640" OR "gc0308"
    if(!camera_dev) {
        rt_kprintf("find camera err!\n");
        return -1;
    };
    rt_device_init(camera_dev);                          //初始化摄像头
    rt_device_open(camera_dev,RT_DEVICE_OFLAG_RDWR); //打开摄像头,读写模式
    rt_device_set_rx_indicate(camera_dev,camera_cb); //设置read回调函数

    /* init face detect model */
    face_landmark = rt_ai_find(RT_AI_FACELANDMARK_MODEL_NAME);                    //查
        找AI模型
    RT_ASSERT(face_landmark);
    if (rt_ai_init(face_landmark,(rt_ai_buffer_t *)kpu_image) != 0)
    {
        rt_kprintf("\nmodel init error\n");
        while (1) {};
    }
    ai_log("rt_ai_init complete..\n");
    facelandmark_region_layer_init(&rl, anchor, 3160, 4, 5, 1, 320, 240, 0.7, 0.4,
        variances);
    facelandmark_boxes_info_init(&rl, &boxes, 200);
    while (1){
        /* 清除标志 */
        g_dvp_finish_flag = 0;
        /* 采集图像显示&AI由display_image地址开始连续存放 */
        rt_device_read(camera_dev,0,display_image,0);
        while (g_dvp_finish_flag == 0) {};  // 等待采集中断 使能
```

```
        /* run face detect */
        g_ai_done_flag= 0;
        if(rt_ai_run(face_landmark,ai_done,&g_ai_done_flag) != 0){
            ai_log("rtak run fail!\n");
            while (1){}  ;
        }
        while(!g_ai_done_flag){};                          //等待推理结束
        pred_box = (float*)rt_ai_output(face_landmark,  0);       //调用相关后处理函
            数，获得bbox信息
        pred_landm = (float*)rt_ai_output(face_landmark, 1);      //调用相关后处理函
            数，获得人脸关键点坐标
        pred_clses = (float*)rt_ai_output(face_landmark, 2);      //调用相关后处理函
            数，获得分类结果
        rl.bbox_input= pred_box;
        rl.landm_input= pred_landm;
        rl.clses_input= pred_clses;
        facelandmark_region_layer_run(&rl, &boxes);
        lcd_draw_picture(0, 0, 320, 240, (uint32_t *)display_image);   //显示图片到
            LCD

        /* run key point detect */
        facelandmark_region_layer_draw_boxes(&boxes, facelandmark_drawboxes);
        facelandmark_boxes_info_reset(&boxes);
    }
}
```

模型输入输出说明：

- k210 指定的图片数据输入格式是 CHW，chanel: C，heigh: H，width: W; 模型的输入是 3*240*320，C=3，H=240，W=320，图像数据类型为 uint8

# 第 5 章

# 编译和运行

## 4.1 编译

参考 lab2-env 教程中 Studio 使用方法。新建->RT-Thread项目->基于开发板->K210-RT-DRACO 输入工程目录和工程名，新建基于开发板的模板工程。将实验代码复制到 application 文件中替换原文件代码。点击 编译。会在你的工程根目录下生成一个 rtthread.bin 文件，然后参考下面的 烧录方法。其中 rtthread.bin 需要烧写到设备中进行运行。

## 4.2 烧录

连接好串口，点击 Studio 中的下载图标进行下载，详细可参考lab-env2

**或者**

使用 K-Flash 工具进行烧写 bin 文件。

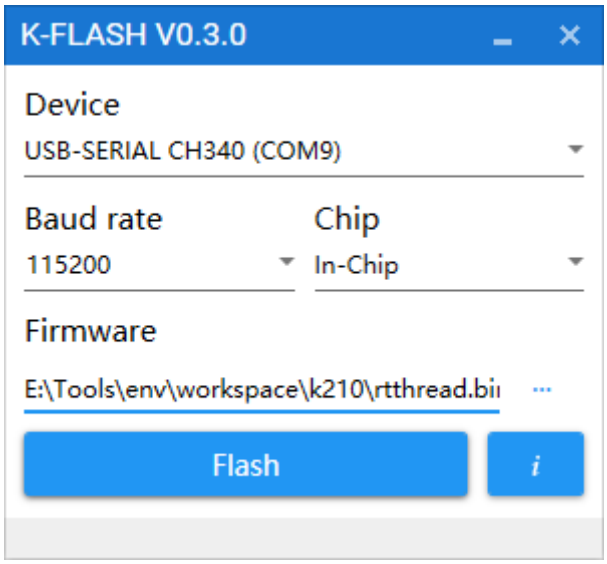



图 5.1: *K-Flash*

# 第 6 章

# 实验现象

如果编译 & 烧写无误，K-Flash 会自动打开 Windows 终端，自动连接实验板，系统启动后，我们的程序会自动运行。

本实验中，经过嵌入式工程后处理给出推理结果，原图像和人脸关键点检测图片的对比结果如下所示：

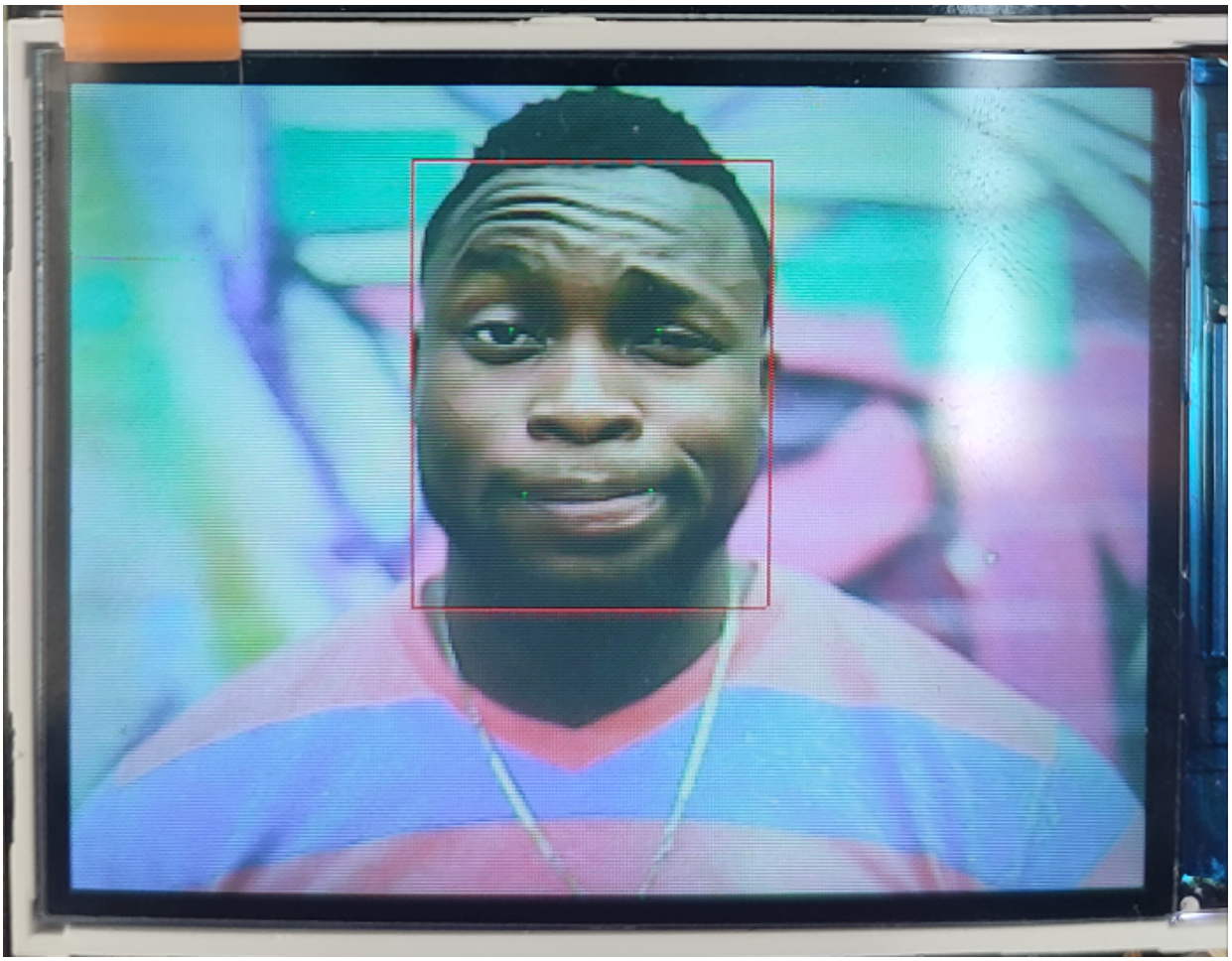


**图 6.1:** *识别结果*

# 第 7 章

# 附件

## 6.1 嵌入式 AI 开发 API 文档

```
rt_ai_t rt_ai_find(const char *name);
```

| Paramaters | Description |
| --- | --- |
| name | 注册的模型名 |
| **Return** | – |
| rt_ai_t | 已注册模型句柄 |
| NULL | 未发现模型 |

**描述**: 查找已注册模型

```
rt_err_t rt_ai_init(rt_ai_t ai, rt_aibuffer_t* work_buf);
```

| Paramaters | Description |
| --- | --- |
| ai | rt_ai_t 句柄 |
| work_buf | 运行时计算所用内存 |
| | |
| **Return** | – |
| 0 | 初始化成功 |
| 非 0 | 初始化失败 |

**描述**: 初始化模型句柄, 挂载模型信息, 准备运行环境.

```
rt_err_t rt_ai_run(rt_ai_t ai, void (*callback)(void * arg), void *arg);
```

| Paramaters | Description |
| --- | --- |
| ai | rt_ai_t 模型句柄 |
| callback | 运行完成回调函数 |
| arg | 运行完成回调函数参数 |
| **Return** | – |
| 0 | 成功 |
| 非 0 | 失败 |

**描述**: 模型推理计算

```
rt_aibuffer_t rt_ai_output(rt_ai_t aihandle,rt_uint32_t index);
```

| Paramaters | Description |
| --- | --- |
| ai | rt_ai_t 模型句柄 |
| index | 结果索引 |
| **Return** | – |
| NOT NULL | 结果存放地址 |
| NULL | 获取结果失败 |

**描述**: 获取模型运行的结果, 结果获取后.

`rt_ai_libs/readme.md` 文件中有详细说明

## 6.2 LCD API 说明手册

```
/**
 * @fn void lcd_init(void);
 * @brief LCD初始化
*/
void lcd_init(void);
/**
 * @fn void lcd_clear(uint16_t color);
 * @brief 清屏
 * @param color 清屏时屏幕填充色
*/
void lcd_clear(uint16_t color);
/**
 * @fn void lcd_set_direction(lcd_dir_t dir);
 * @brief 设置LCD显示方向
 * @param dir 显示方向参数
*/
```

```
void lcd_set_direction(lcd_dir_t dir);
/**
 * @fn void lcd_set_area(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2);
 * @brief 设置LCD显示区域
 * @param x1 左上角横坐标
 * @param y1 左上角纵坐标
 * @param x2 右下角横坐标
 * @param y2 右下角纵坐标
*/
void lcd_set_area(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2);
/**
 * @fn void lcd_draw_point(uint16_t x, uint16_t y, uint16_t color);
 * @brief 画点
 * @param x 横坐标
 * @param y 纵坐标
 * @param color 颜色
*/
void lcd_draw_point(uint16_t x, uint16_t y, uint16_t color);

/**
 * @fn void lcd_draw_string(uint16_t x, uint16_t y, char *str, uint16_t color);
 * @brief 显示字符串
 * @param x 显示位置横坐标
 * @param y 显示位置纵坐标
 * @param str 字符串
 * @param color 字符串颜色
*/
void lcd_draw_string(uint16_t x, uint16_t y, char *str, uint16_t color);
/**
 * @fn void lcd_draw_picture(uint16_t x1, uint16_t y1, uint16_t width, uint16_t
     height, uint32_t *ptr);
 * @brief 显示图片
 * @param x1 左上角横坐标
 * @param y1 左上角纵坐标
 * @param width 图片宽
 * @param height 图片高
 * @param ptr 图片地址
*/
void lcd_draw_picture(uint16_t x1, uint16_t y1, uint16_t width, uint16_t height,
    uint32_t *ptr);
/**
 * @fn void lcd_draw_rectangle(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2,
     uint16_t width, uint16_t color);
 * @brief 画矩形
 * @param x1 左上角横坐标
 * @param y1 左上角纵坐标
 * @param x2 右下角横坐标
 * @param y2 右下角纵坐标
 * @param width 线条宽度(当前无此功能)
```

```
 * @param color 线条颜色
*/
```

## 6.3 Camera 使用

Camera 已对接 RT-Thread 设备驱动框架，可通过 RT-Thread Device 标准接口进行调用。RT-Thread 设备框架文档：https://www.rt-thread.org/document/site/#/rt-thread-version/rt-thread-standard/README

下面将使用到的 API 进行简要说明，引用自 RTT 官方文档：

### 查找设备

应用程序根据设备名称获取设备句柄，进而可以操作设备。查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

| 参数 | 描述 |
| --- | --- |
| name | 设备名称 |
| 返回 | —— |
| 设备句柄 | 查找到对应设备将返回相应的设备句柄 |
| RT_NULL | 没有找到相应的设备对象 |

### 初始化设备

获得设备句柄后，应用程序可使用如下函数对设备进行初始化操作：

```
rt_err_t rt_device_init(rt_device_t dev);
```

| 参数 | 描述 |
| --- | --- |
| dev | 设备句柄 |
| 返回 | —— |
| RT_EOK | 设备初始化成功 |
| 错误码 | 设备初始化失败 |

[!NOTE] 注：当一个设备已经初始化成功后，调用这个接口将不再重复做初始化 0。

### 打开和关闭设备

通过设备句柄，应用程序可以打开和关闭设备，打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

```
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
```

| 参数 | 描述 |
| --- | --- |
| dev | 设备句柄 |
| oflags | 设备打开模式标志 |
| 返回 | —— |
| RT_EOK | 设备打开成功 |
| -RT_EBUSY | 如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数，此设备将不允许重复打开 |
| 其他错误码 | 设备打开失败 |

oflags 支持以下的参数：

```
#define RT_DEVICE_OFLAG_CLOSE 0x000   /* 设备已经关闭（内部使用）*/
#define RT_DEVICE_OFLAG_RDONLY 0x001  /* 以只读方式打开设备 */
#define RT_DEVICE_OFLAG_WRONLY 0x002  /* 以只写方式打开设备 */
#define RT_DEVICE_OFLAG_RDWR 0x003    /* 以读写方式打开设备 */
#define RT_DEVICE_OFLAG_OPEN 0x008    /* 设备已经打开（内部使用）*/
#define RT_DEVICE_FLAG_STREAM 0x040   /* 设备以流模式打开 */
#define RT_DEVICE_FLAG_INT_RX 0x100   /* 设备以中断接收模式打开 */
#define RT_DEVICE_FLAG_DMA_RX 0x200   /* 设备以 DMA 接收模式打开 */
#define RT_DEVICE_FLAG_INT_TX 0x400   /* 设备以中断发送模式打开 */
#define RT_DEVICE_FLAG_DMA_TX 0x800   /* 设备以 DMA 发送模式打开 */
```

应用程序打开设备完成读写等操作后，如果不需要再对设备进行操作则可以关闭设备，通过如下函数完成：

```
rt_err_t rt_device_close(rt_device_t dev);
```

| 参数 | 描述 |
| --- | --- |
| dev | 设备句柄 |
| 返回 | —— |
| RT_EOK | 关闭设备成功 |
| -RT_ERROR | 设备已经完全关闭，不能重复关闭设备 |
| 其他错误码 | 关闭设备失败 |

### 读写设备

应用程序从设备中读取数据可以通过如下函数完成（摄像头设备中pos 和size参数无作用)：

```
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos,void* buffer, rt_size_t size)
    ;
```

| 参数 | 描述 |
| --- | --- |
| dev | 设备句柄 |
| pos | 读取数据偏移量 |
| buffer | 内存缓冲区指针，读取的数据将会被保存在缓冲区中 |
| size | 读取数据的大小 |
| 返回 | —— |
| 读到数据的实际大小 | 如果是字符设备，返回大小以字节为单位，如果是块设备，返回的大小以块为单位 |
| 0 | 需要读取当前线程的 errno 来判断错误状态 |

### 数据收发回调

当硬件设备收到数据时，可以通过如下函数回调另一个函数来设置数据接收指示，通知上层应用线程有数据到达：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t
    dev,rt_size_t size));
```

| 参数 | 描述 |
| --- | --- |
| dev | 设备句柄 |
| rx_ind | 回调函数指针 |
| 返回 | —— |
| RT_EOK | 设置成功 |