Atlas 200I DK A2 开发者套件
23.0.RC3

AscendCL 应用开发入门

文档版本	01
发布日期	2023-11-14

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

安全声明

漏洞声明

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该政策可参考华为公司官方网站的网址：https://www.huawei.com/cn/psirt/vul-response-process。
如企业客户须获取漏洞信息，请访问：https://securitybulletin.huawei.com/enterprise/cn/security-advisory。

目 录

1 图像分类应用样例开发介绍（Python）

本章节介绍基于AscendCL接口（下文若出现ACL或acl为同一含义）如何开发一个基于ResNet-50模型的图像分类样例。

样例介绍

“图片分类应用”，即标识图片所属的分类。

图 1-1 图片分类应用

本例中使用的是Caffe框架的ResNet-50模型。可以直接使用训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以基于算法、框架构建适合的模型。

模型的输入数据与输出数据格式：

- 输入数据：RGB格式、224*224分辨率的输入图片。
- 输出数据：图片的类别标签及其对应置信度。

说明

- 置信度是指图片所属某个类别可能性。
- 类别标签和类别的对应关系与训练模型时使用的数据集有关，需要查阅对应数据集的标签及类别的对应关系。

业务模块介绍

业务模块的操作流程如**图2 业务流程图**所示。

图 1-2 业务流程图

1. 图片输入：收集待分类图片数据集做为输入数据。
2. 输入预处理：将图片读入为numpy array，调整为模型输入大小，并修改像素值范围，与训练数据保持一致。
3. 模型推理：经过模型推理后得到图片分类结果。
4. 后处理：根据预测概率排序，找出概率最高的5个预测类别。
5. 可视化或保存到文件：将预测结果显示在界面或保存到文件。

获取代码

步骤1 获取代码文件。

单击**获取链接**或使用wget命令，下载代码文件压缩包，以**root**用户登录开发者套件。

```
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/Atlas%20200I%20DK%20A2/DevKit/models/sdk_cal_samples/resnet50_acl_python_sample.zip
```

步骤2 将“resnet50_acl_python_sample.zip”压缩包上传到开发者套件，解压并进入解压后的目录。

```
unzip resnet50_acl_python_sample.zip
cd resnet50_acl_python_sample
```

代码目录结构如下所示，按照正常开发流程，需要将框架模型文件转换成昇腾AI处理器支持推理的om格式模型文件，鉴于当前是入门内容，用户可直接获取已转换好的om模型进行推理。

```
resnet50_acl_python_sample
├── data
│   ├── test.jpg                  // 测试图片
├── model
│   ├── resnet50.om                  // ResNet-50网络的om模型
├── main.py                        // 运行程序的脚本
├── imagenet-labels.json             // 图片标签文件
├── font.ttf                       // 字体文件，用于推理结果可视化
```

步骤3 准备用于推理的图片数据。

如**步骤2**所示文件结构，内置测试图片为“test.jpg”，用户也可从imagenet数据集中获取其它图片。

----结束

代码解析

开发代码过程中，在“resnet50_acl_python_sample/main.py”文件中已包含读入数据、前处理、推理、后处理等功能，串联整个应用代码逻辑，此处仅对代码进行解析。

1. 在“main.py”文件的开头有如下代码，用于导入第三方库与调用AscendCL接口推理所需文件。

   ```
   import json
   import os
   ```

```
import numpy as np  # 用于对多维数组进行计算
from PIL import Image, ImageDraw, ImageFont  # 图片处理库，用于在图片上画出推理结果

import acl  # acl推理相关接口


ACL_MEM_MALLOC_HUGE_FIRST = 0  # 内存分配策略
ACL_SUCCESS = 0  # 成功状态值
IMG_EXT = ['.jpg', '.JPG', '.png', '.PNG', '.bmp', '.BMP', '.jpeg', '.JPEG']  # 所支持的图片格式
```

2. 定义一个main函数，串联整个应用的代码逻辑。

```
def main():
    # 参数初始化
    device = 0  # 设备id
    model_path = "./model/resnet50.om"  # 模型路径
    images_path = "./data"  # 数据集路径
    label_path = "./imagenet-labels.json"  # 数据集标签
    output_path = "./output"  # 推理结果保存路径

    os.makedirs(output_path, exist_ok=True)  # 构造输出路径
    with open(label_path) as f:
        idx2label_list = json.load(f)  # 加载标签列表
    net = Net(device, model_path, idx2label_list)  # 初始化模型

    # 加载文件夹中每张图片路径
    images_list = [os.path.join(images_path, img)
                   for img in os.listdir(images_path)
                   if os.path.splitext(img)[1] in IMG_EXT]

    # 对每张图片进行预处理、推理、以及保存预测结果
    for img_path in images_list:
        print("images:{}".format(img_path))  # 输出图片路径
        img = preprocess_img(img_path)  # 预处理
        pred_dict = net.run([img])  # 推理
        save_image(img_path, pred_dict, output_path)  # 保存预测结果

    # 释放 acl 资源
    print("*****run finish******")
    net.release_resource()

if __name__ == '__main__':
    main()
```

3. 定义推理相关实现类，包含初始化acl，加载模型，创建输入输出数据类型、推理、解析模型输出、释放acl资源等功能。

   使用AscendCL接口开发应用时，必须先初始化AscendCL ，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。

```
class Net(object):
    def __init__(self, device_id, model_path, idx2label_list):
        self.device_id = device_id  # 设备id
        self.model_path = model_path  # 模型路径
        self.model_id = None  # 模型id
        self.context = None  # 用于管理资源，可详见https://www.hiascend.com/document/detail/zh/
CANNCommunityEdition/600alpha006/infacldevg/aclpythondevg/aclpythondevg_01_0065.html

        self.model_desc = None  # 模型描述信息，包括模型输入个数、输入维度、输出个数、输出维度等信
息
        self.load_input_dataset = None  # 输入数据集，aclmdlDataset类型
        self.load_output_dataset = None  # 输出数据集，aclmdlDataset类型

        self.init_resource()  # 初始化 acl 资源
        self.idx2label_list = idx2label_list  # 加载的标签列表

    def init_resource(self):
        """初始化 acl 相关资源"""
        print("init resource stage:")

        ret = acl.init()  # 初始化 acl
        check_ret("acl.init", ret)
```



```
        ret = acl.rt.set_device(self.device_id)  # 指定 device
        check_ret("acl.rt.set_device", ret)

        self.context, ret = acl.rt.create_context(self.device_id)  # 创建 context
        check_ret("acl.rt.create_context", ret)

        self.model_id, ret = acl.mdl.load_from_file(self.model_path)  # 加载模型
        check_ret("acl.mdl.load_from_file", ret)

        self.model_desc = acl.mdl.create_desc()  # 创建描述模型基本信息的数据类型
        print("init resource success")

        ret = acl.mdl.get_desc(self.model_desc, self.model_id)  # 根据模型ID获取模型基本信息
        check_ret("acl.mdl.get_desc", ret)


    def _gen_input_dataset(self, input_list):
        ''' 组织输入数据的dataset结构 '''
        input_num = acl.mdl.get_num_inputs(self.model_desc)  # 根据模型信息得到模型输入个数
        self.load_input_dataset = acl.mdl.create_dataset()  # 创建输入dataset结构
        for i in range(input_num):
            item = input_list[i]  # 获取第 i 个输入数据
            data = acl.util.bytes_to_ptr(item.tobytes())  # 获取输入数据字节流
            size = item.size * item.itemsize  # 获取输入数据字节数
            dataset_buffer = acl.create_data_buffer(data, size)  # 创建输入dataset buffer结构, 填入输入数
据
            _, ret = acl.mdl.add_dataset_buffer(self.load_input_dataset, dataset_buffer)  # 将dataset buffer
加入dataset
        print("create model input dataset success")


    def _gen_output_dataset(self):
        ''' 组织输出数据的dataset结构 '''
        output_num = acl.mdl.get_num_outputs(self.model_desc)  # 根据模型信息得到模型输出个数
        self.load_output_dataset = acl.mdl.create_dataset()  # 创建输出dataset结构
        for i in range(output_num):
            temp_buffer_size = acl.mdl.get_output_size_by_index(self.model_desc, i)  # 获取模型输出个数
            temp_buffer, ret = acl.rt.malloc(temp_buffer_size, ACL_MEM_MALLOC_HUGE_FIRST)  # 为每个
输出申请device内存
            dataset_buffer = acl.create_data_buffer(temp_buffer, temp_buffer_size)  # 创建输出的data
buffer结构,将申请的内存填入data buffer
            _, ret = acl.mdl.add_dataset_buffer(self.load_output_dataset, dataset_buffer)  # 将 data buffer
加入输出dataset
        print("create model output dataset success")


    def run(self, images):
        """ 数据集构造、模型推理、解析输出"""
        self._gen_input_dataset(images)  # 构造输入数据集
        self._gen_output_dataset()  # 构造输出数据集

        # 模型推理，推理完成后，输出会放入 self.load_output_dataset
        print('execute stage:')
        ret = acl.mdl.execute(self.model_id, self.load_input_dataset, self.load_output_dataset)
        check_ret("acl.mdl.execute", ret)

        # 解析输出
        result = []
        output_num = acl.mdl.get_num_outputs(self.model_desc)  # 根据模型信息得到模型输出个数
        for i in range(output_num):
            buffer = acl.mdl.get_dataset_buffer(self.load_output_dataset, i)  # 从输出dataset中获取buffer
            data = acl.get_data_buffer_addr(buffer)  # 获取输出数据内存地址
            size = acl.get_data_buffer_size(buffer)  # 获取输出数据字节数
            narray = acl.util.ptr_to_bytes(data, size)  # 将指针转为字节流数据

            # 根据模型输出的维度和数据类型,将字节流数据解码为numpy数组
            dims, ret = acl.mdl.get_cur_output_dims(self.model_desc, i)  # 得到当前输出的维度
            out_dim = dims['dims']  # 提取维度信息
```



```
            output_nparray = np.frombuffer(narray, dtype=np.float32).reshape(tuple(out_dim))  # 解码为
numpy数组
            result.append(output_nparray)

        pred_dict = self._print_result(result)  # 按一定格式打印输出

        # 释放模型输入输出数据集
        self._destroy_dataset()
        print('execute stage success')

        return pred_dict

    def _print_result(self, result):
        """打印预测结果"""
        vals = np.array(result).flatten()  # 将结果展开为一维
        top_k = vals.argsort()[-1:-6:-1]  # 将置信度从大到小排列，并得到top5的下标

        print("======== top5 inference results: =============")
        pred_dict = {}
        for j in top_k:
            print(f'{self.idx2label_list[j]}: {vals[j]}')  # 打印出对应类别及概率
            pred_dict[self.idx2label_list[j]] = vals[j]  # 将类别信息和概率存入 pred_dict
        return pred_dict

    def _destroy_dataset(self):
        """ 释放模型输入输出数据 """
        for dataset in [self.load_input_dataset, self.load_output_dataset]:
            if not dataset:
                continue
            number = acl.mdl.get_dataset_num_buffers(dataset)  # 获取输入buffer个数
            for i in range(number):
                data_buf = acl.mdl.get_dataset_buffer(dataset, i)  # 获取每个输入buffer
                if data_buf:
                    ret = acl.destroy_data_buffer(data_buf)  # 销毁每个输入buffer (销毁 aclDataBuffer 类型)
                    check_ret("acl.destroy_data_buffer", ret)
            ret = acl.mdl.destroy_dataset(dataset)  # 销毁输入数据 (销毁 aclmdlDataset类型的数据)
            check_ret("acl.mdl.destroy_dataset", ret)


    def release_resource(self):
        """释放 acl 相关资源"""
        print("Releasing resources stage:")
        ret = acl.mdl.unload(self.model_id)  # 卸载模型
        check_ret("acl.mdl.unload", ret)
        if self.model_desc:
            acl.mdl.destroy_desc(self.model_desc)  # 释放模型描述信息
            self.model_desc = None

        if self.context:
            ret = acl.rt.destroy_context(self.context)  # 释放 Context
            check_ret("acl.rt.destroy_context", ret)
            self.context = None

        ret = acl.rt.reset_device(self.device_id)  # 释放 device 资源
        check_ret("acl.rt.reset_device", ret)

        ret = acl.finalize()  # ACL去初始化
        check_ret("acl.finalize", ret)
        print('Resources released successfully.')

def check_ret(message, ret):
    ''' 用于检查各个返回值是否正常，若否，则抛出对应异常信息 '''
    if ret != ACL_SUCCESS:
        raise Exception("{} failed ret={}".format(message, ret))
```

4. 加载及预处理图像。

```
def preprocess_img(input_path):
    """图片预处理"""
    # 循环加载图片
    input_path = os.path.abspath(input_path)  # 得到当前图片的绝对路径
```



```
    with Image.open(input_path) as image_file:
        image_file = image_file.resize((256, 256))  # 缩放图片
        img = np.array(image_file)  # 转为numpy数组

    # 获取图片的高和宽
    height = img.shape[0]
    width = img.shape[1]

    # 对图片进行切分，取中间区域
    h_off = (height - 224) // 2
    w_off = (width - 224) // 2
    crop_img = img[h_off:height - h_off, w_off:width - w_off, :]

    # 转换bgr格式、数据类型、颜色空间、数据维度等信息
    img = crop_img[:, :, ::-1]  # 改变通道顺序，rgb to bgr
    shape = img.shape  # 暂时存储维度信息
    img = img.astype("float32")  # 转为 float32 数据类型
    img[:, :, 0] -= 104  # 常数104,117,123用于将图像转换到Caffe模型需要的颜色空间
    img[:, :, 1] -= 117
    img[:, :, 2] -= 123
    img = img.reshape([1] + list(shape))  # 扩展第一维度，适应模型输入
    img = img.transpose([0, 3, 1, 2])  # 将 (batch,height,width,channels) 转为
(batch,channels,height,width)
    return img
```

5. 将预测结果保存为图片。

```
def save_image(path, pred_dict, output_path):
    """保存预测图片"""
    font = ImageFont.truetype('font.ttf', 20)  # 指定字体和字号
    color = "#fff"  # 指定颜色
    im = Image.open(path)  # 打开图片
    im = im.resize((800, 500))  # 对图片进行缩放

    start_y = 20  # 在图片上画分类结果时的纵坐标初始值
    draw = ImageDraw.Draw(im)  # 准备画图
    for label, pred in pred_dict.items():
        draw.text(xy = (20, start_y), text=f'{label}: {pred:.2f}', font=font, fill=color)  # 将预测的类别与置信
度添加到图片
        start_y += 30  # 每一行文字往下移30个像素

    im.save(os.path.join(output_path, os.path.basename(path)))  # 保存图片到输出路径
```

运行推理

进入“resnet50_acl_python_sample”目录，执行以下命令。

```
python main.py
```

终端上屏显的结果如下，输出置信度Top5的类别名称和对应的预测概率：

```
======== top5 inference results: =============
Standard Poodle: 0.935546875
Miniature Poodle: 0.041107177734375
Toy Poodle: 0.0191192626953125
Cocker Spaniels: 0.0028858184814453125
Afghan Hound: 0.00031375885009765625
```

在“resnet50_acl_python_sample/output”目录下，保存相应的预测结果图片，如图1-3所示。

图 1-3 预测结果图

样例总结与扩展

以上代码包括以下几个步骤：

1. 初始化acl资源：在调用AscendCL相关资源时，必须先初始化AscendCL，否则可能会导致后续系统内部资源初始化出错。

2. 对图片进行前处理：在此样例中，首先利用PIL读入图片，再对图片进行裁剪，并转换数据类型、颜色空间、数据维度等操作。

3. 推理：利用AscendCL接口对图片进行推理，其中包括构建输入输出数据集结构、推理、从输出数据结构中解析出numpy数据等步骤。

4. 保存结果：取出置信度最高的前五个预测类别，保存带有分类结果的图片。

5. 资源销毁：最后记得释放相关资源，包括卸载模型、销毁输入输出数据集、释放Context、释放指定的计算设备、以及AscendCL去初始化等操作。

MindX SDK接口分类总结：

分类	接口函数	描述
AscendCL初始化相关	acl.init()	pyACL初始化函数
	acl.rt.set_device(device_id)	指定当前进程或线程中用于运算的Device，同时隐式创建默认Context
	acl.rt.create_context(device_id)	在当前进程或线程中显式创建一个Context
模型描述信息相关	acl.mdl.load_from_file(model_path)	从文件加载离线模型数据（适配昇腾AI处理器的离线模型）
	acl.mdl.create_desc()	创建aclmdlDesc类型的数据
	acl.mdl.get_desc(model_desc, model_id)	根据模型ID获取该模型的aclmdlDesc类型数据

分类	接口函数	描述
	acl.mdl.get_num_inputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输入个数
	acl.mdl.get_num_outputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输出个数
	acl.mdl.get_output_size_by_index(model_desc, i)	根据aclmdlDesc类型的数据，获取指定输出的大小，单位为Byte
	acl.mdl.get_cur_output_dims(model_desc, i)	根据模型描述信息获取指定的模型输出tensor的实际维度信息
数据集结构相关	acl.mdl.create_dataset()	创建aclmdlDataset类型的数据
	acl.create_data_buffer(data_addr, size)	创建aclDataBuffer类型的数据，该数据类型用于描述内存地址、大小等内存信息
	acl.mdl.add_dataset_buffer(dataset, date_buffer)	向aclmdlDataset中增加aclDataBuffer
	acl.mdl.get_dataset_num_buffers(dataset)	获取aclmdlDataset中aclDataBuffer的个数
	acl.mdl.get_dataset_buffer(dataset, i)	获取aclmdlDataset中的第i个aclDataBuffer
	acl.get_data_buffer_addr(buffer)	获取aclDataBuffer类型中的数据的地址对象
	acl.get_data_buffer_size(buffer)	获取aclDataBuffer类型中数据的内存大小，单位Byte
	acl.util.bytes_to_ptr(bytes_data)	将bytes对象转换成为void*数据，可以将转换好的数据传递给C函数直接使用
	acl.util.ptr_to_bytes(ptr, size)	将void*数据转换为bytes对象，可以使python代码直接访问
	acl.rt.malloc(size, policy)	申请Device上的内存
推理相关	acl.mdl.execute(model_id, input_dataset, output_dataset)	执行模型推理，直到返回推理结果
销毁资源相关	acl.destroy_data_buffer(data_buffer)	销毁aclDataBuffer类型的数据
	acl.mdl.destroy_dataset(dataset)	销毁aclmdlDataset类型的数据
	acl.mdl.unload(model_id)	系统完成模型推理后，可调用该接口卸载模型，释放资源
	acl.mdl.destroy_desc(model_desc)	销毁aclmdlDesc类型的数据

分类	接口函数	描述
	acl.rt.destroy_context(context)	销毁一个Context，释放Context的资源
	acl.rt.reset_device(device_id)	复位当前运算的Device，释放Device上的资源，包括默认Context、默认Stream以及默认Context下创建的所有Stream
	acl.finalize()	pyACL去初始化函数，用于释放进程内的pyACL相关资源

理解各个接口含义后，用户可进行灵活运用。除此外，此样例中只示范了图片推理，若需要对视频流数据进行推理，可用三种方式输入视频流数据：USB摄像头和手机摄像头。具体使用方式可参考《**摄像头拉流**》，用户只需将前处理、推理及后处理代码放入摄像头推理代码的循环中即可，注意有些细节地方需进行相应修改，下面以USB摄像头为例，其他摄像头使用方式可按相应逻辑修改。

1. 修改引入三方库部分如下，加入import cv2，用于USB视频流的读取和保存。

```
import json
import os

import numpy as np  # 用于对多维数组进行计算
from PIL import Image, ImageDraw, ImageFont  # 图片处理库, 用于在图片上画出推理结果
import cv2

import acl  # acl推理相关接口
```

2. 修改main函数如下，删除了图片读取相关代码，并加入了USB摄像头读取以及视频保存等相关代码。

```
def main():
    # 参数初始化、加载需要的路径
    device = 0  # 设备id
    model_path = "./model/resnet50.om"  # 模型路径
    label_path = "./imagenet-labels.json"  # 数据集标签
    output_path = "./output"  # 推理结果保存路径
    os.makedirs(output_path, exist_ok=True)  # 构造输出路径
    with open(label_path) as f:
        idx2label_list = json.load(f)  # 加载标签列表

    # 加载模型
    net = Net(device, model_path, idx2label_list)  # 初始化模型

    # 打开摄像头
    cap = cv2.VideoCapture(0)  # 打开摄像头

    # 获取保存视频相关变量
    fps = cap.get(cv2.CAP_PROP_FPS)
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    outfile = os.path.join(output_path, 'video_result.mp4')
    video_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    video_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    writer = cv2.VideoWriter(outfile, fourcc, fps, (video_width, video_height))

    try:
        while(cap.isOpened()):  # 在摄像头打开的情况下循环执行
            ret, frame = cap.read()  # 此处 frame 为 bgr 格式图片

            # 这里放入模型前处理、推理、后处理相关代码
            img = preprocess_img(frame)  # 前处理
            pred_dict = net.run([img])  # 推理
            img_res = save_image(frame, pred_dict)  # 保存预测结果
            writer.write(img_res)  # 将推理结果写入视频
```

```
    except KeyboardInterrupt:
        cap.release()
        writer.release()
    finally:
        cap.release()
        writer.release()

    # 释放 acl 资源
    print("*****run finish******")
    net.release_resource()
```

3. 修改前处理函数preprocess_img如下，其中原代码是利用PIL（pillow）直接读入图片，为rgb格式，而此处传入为摄像头读入的帧，为bgr格式，所以删除了PIL相关代码，并利用了img_rgb = img_bgr[:,:,::-1]这一句将摄像头帧转换为了rgb，再使用cv2.resize对图片进行了缩放，其他行的代码保持不变。

```
def preprocess_img(img_bgr):
    """图片预处理"""
    img_rgb = img_bgr[:,:,::-1]
    img = cv2.resize(img_rgb, (256, 256))

    # 获取图片的高和宽
    height = img.shape[0]
    width = img.shape[1]

    # 对图片进行切分, 取中间区域
    h_off = (height - 224) // 2
    w_off = (width - 224) // 2
    crop_img = img[h_off:height - h_off, w_off:width - w_off, :]

    # 转换bgr格式、数据类型、颜色控件、数据维度等信息
    img = crop_img[:, :, ::-1]  # 改变通道顺序, rgb to bgr
    shape = img.shape  # 暂时存储维度信息
    img = img.astype("float32")  # 转为 float32 数据类型
    img[:, :, 0] -= 104  # 常数104,117,123用于将图像转换到Caffe模型需要的颜色空间
    img[:, :, 1] -= 117
    img[:, :, 2] -= 123
    img = img.reshape([1] + list(shape))  # 扩展第一维度, 适应模型输入
    img = img.transpose([0, 3, 1, 2])  # 将 (batch,height,width,channels) 转为
(batch,channels,height,width)
    return img
```

4. 修改save_image函数如下，由于图片是利用PIL（pillow）读入原图并画出结果，但此处传入直接为摄像头帧，所以直接cv2将结果画在图片上比较方便，即使用cv2.putText函数。

```
def save_image(img_bgr, pred_dict):
    """保存预测图片"""
    start_y = 20  # 在图片上画分类结果时的纵坐标初始值
    for label, pred in pred_dict.items():
        img_bgr = cv2.putText(img_bgr, f'{label}: {pred:.2f}', (20, start_y), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (255, 255, 255), 1)  # 参数含义依次为：原图、结果文字、左上角坐标、字体、字体大小、颜色、字体粗细
        start_y += 30  # 每一行文字往下移30个像素

    return img_bgr
```


2 目标检测应用样例开发介绍（Python）

本章节介绍基于AscendCL接口如何开发一个基于Yolo模型的目标检测样例。

样例介绍

目标检测，即给定一张图片，识别出图片中的目标位置。

图 2-1 目标检测应用

本例中使用的是pytorch框架的yolov5模型。可以直接使用训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以基于算法、框架构建适合的模型。

模型的输入数据与输出数据格式：

- 输入数据：RGB格式图片，分辨率为 640×640，输入形状为（1，3，640，640），也即（batchsize，channel，height，width）。
- 输出数据：目标检测框的坐标值、置信度、类别。

  说明

  输出数组需要经过一定后处理，才能显示为正常的图片。

业务模块介绍

业务模块的操作流程如图2-2所示。

图 2-2 业务流程图

1. 图片输入：收集待检测数据集做为输入数据。
2. 输入预处理：将图片读入为数组，并缩放到模型所需大小，然后调整像素范围。
3. 模型推理：经过模型推理后得到目标检测结果。
4. 后处理：从输出中解析出检测框坐标，并在原始图像上画出检测框。
5. 可视化或保存到文件：将图片显示在界面或保存到文件。

获取代码

步骤1 获取代码文件。

单击**获取链接**或使用wget命令，下载代码文件压缩包，以**root**用户登录开发者套件。

```
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/Atlas%20200I%20DK%20A2/DevKit/models/sdk_cal_samples/yolo_acl_sample.zip
```

步骤2 将“yolo_acl_sample.zip”压缩包上传到开发者套件，解压并进入解压后的目录。

```
unzip yolo_acl_sample.zip
cd yolo_acl_sample
```

代码目录结构如下所示，按照正常开发流程，需要将框架模型文件转换成昇腾AI处理器支持推理的om格式模型文件，鉴于当前是入门内容，用户可直接获取已转换好的om模型进行推理。

```
|-- yolo_acl_sample
   |-- infer              # 推理文件夹
   |-- main.py            # 主程序
      |-- det_utils.py       # 模型相关前后处理函数，函数为通用函数，和AscnedCL接口无关联
      |-- coco_names.txt     # coco数据集所有类别名
      |-- world_cup.jpg      # 测试图片
   |-- yolov5s_bs1.om  # om模型
```

----结束

代码解析

开发代码过程中，在“yolo_acl_sample/infer/main.py”文件中已包含读入数据、前处理、推理、后处理等功能，串联整个应用代码逻辑，此处仅对代码进行解析。

1. 导入需要的第三方库以及调用AscendCL接口推理所需文件，定义模型相关变量，如设备ID，内存申请策略等。

```
# coding=utf-8

from abc import abstractmethod, ABC  # 用于定义抽象类

import cv2  # 图片处理三方库，用于对图片进行前后处理
import numpy as np  # 用于对多维数组进行计算
import torch  # 深度学习运算框架，此处主要用来处理数据

import acl  # AscnedCL推理相关接口

from det_utils import get_labels_from_txt, letterbox, scale_coords, nms, draw_bbox  # 模型前后处理相关函数

DEVICE_ID = 0  # 设备id
SUCCESS = 0  # 成功状态值
FAILED = 1  # 失败状态值
ACL_MEM_MALLOC_NORMAL_ONLY = 2  # 申请内存策略, 仅申请普通页

trained_model_path = '../yolov5s_bs1.om'  # 模型路径
image_path = 'world_cup.jpg'  # 测试图片路径
```

2. 资源初始化。使用AscendCL接口开发应用时，必须先初始化AscnedCL，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。

```
# acl初始化
def init_acl(device_id):
    acl.init()
    ret = acl.rt.set_device(device_id)  # 指定运算的Device
    if ret:  # 若指定出错，则抛出异常
        raise RuntimeError(ret)
    context, ret = acl.rt.create_context(device_id)  # 显式创建一个Context
    if ret:  # 若创建出错，则抛出异常
        raise RuntimeError(ret)
    print('Init ACL Successfully')
    return context

# acl 去初始化
def deinit_acl(context, device_id):
    ret = acl.rt.destroy_context(context)  # 释放 Context
    if ret:  # 若释放出错，则抛出异常
        raise RuntimeError(ret)
    ret = acl.rt.reset_device(device_id)  # 释放Device
    if ret:  # 若释放出错，则抛出异常
        raise RuntimeError(ret)
    ret = acl.finalize()  # 去初始化
    if ret:  # 若去初始化出错，则抛出异常
        raise RuntimeError(ret)
    print('Deinit ACL Successfully')
```

3. 定义模型资源相关基类，承担初始化模型资源、创建输入输出数据集、执行推理、解析输出、释放模型资源等功能，之后的Yolo模型推理可继承此类。

```
class Model(ABC):
    def __init__(self, model_path):
        print(f"load model {model_path}")
        self.model_path = model_path  # 模型路径
        self.model_id = None  # 模型 id
        self.input_dataset = None  # 输入数据结构
        self.output_dataset = None  # 输出数据结构
        self.model_desc = None  # 模型描述信息
        self._input_num = 0  # 输入数据个数
        self._output_num = 0  # 输出数据个数
        self._output_info = []  # 输出信息列表
        self._is_released = False  # 资源是否被释放
        self._init_resource()

    def _init_resource(self):
        ''' 初始化模型、输出相关资源。相关数据类型: aclmdlDesc aclDataBuffer aclmdlDataset'''
        print("Init model resource")
        # 加载模型文件
        self.model_id, ret = acl.mdl.load_from_file(self.model_path)  # 加载模型
        self.model_desc = acl.mdl.create_desc()  # 初始化模型信息对象
        ret = acl.mdl.get_desc(self.model_desc, self.model_id)  # 根据模型获取描述信息
        print("[Model] Model init resource stage success")

        # 创建模型输出 dataset 结构
        self._gen_output_dataset()  # 创建模型输出dataset结构

    def _gen_output_dataset(self):
        ''' 组织输出数据的dataset结构 '''
        ret = SUCCESS
        self._output_num = acl.mdl.get_num_outputs(self.model_desc)  # 获取模型输出个数
        self.output_dataset = acl.mdl.create_dataset()  # 创建输出dataset结构
        for i in range(self._output_num):
            temp_buffer_size = acl.mdl.get_output_size_by_index(self.model_desc, i)  # 获取模型输出个数
            temp_buffer, ret = acl.rt.malloc(temp_buffer_size, ACL_MEM_MALLOC_NORMAL_ONLY)  # 为
每个输出申请device内存
            dataset_buffer = acl.create_data_buffer(temp_buffer, temp_buffer_size)  # 创建输出的data
buffer结构,将申请的内存填入data buffer
            _, ret = acl.mdl.add_dataset_buffer(self.output_dataset, dataset_buffer)  # 将 data buffer 加入
输出dataset

        if ret == FAILED:
            self._release_dataset(self.output_dataset)  # 失败时释放dataset
        print("[Model] create model output dataset success")
```



```
    def _gen_input_dataset(self, input_list):
        ''' 组织输入数据的dataset结构 '''
        ret = SUCCESS
        self._input_num = acl.mdl.get_num_inputs(self.model_desc)  # 获取模型输入个数
        self.input_dataset = acl.mdl.create_dataset()  # 创建输入dataset结构
        for i in range(self._input_num):
            item = input_list[i]  # 获取第 i 个输入数据
            data_ptr = acl.util.bytes_to_ptr(item.tobytes())  # 获取输入数据字节流
            size = item.size * item.itemsize  # 获取输入数据字节数
            dataset_buffer = acl.create_data_buffer(data_ptr, size)  # 创建输入dataset buffer结构, 填入输入数据
            _, ret = acl.mdl.add_dataset_buffer(self.input_dataset, dataset_buffer)  # 将dataset buffer加入dataset

        if ret == FAILED:
            self._release_dataset(self.input_dataset)  # 失败时释放dataset
        print("[Model] create model input dataset success")

    def _unpack_bytes_array(self, byte_array, shape, datatype):
        ''' 将内存不同类型的数据解码为numpy数组 '''
        np_type = None

        # 获取输出数据类型对应的numpy数组类型和解码标记
        if datatype == 0:  # ACL_FLOAT
            np_type = np.float32
        elif datatype == 1:  # ACL_FLOAT16
            np_type = np.float16
        elif datatype == 3:  # ACL_INT32
            np_type = np.int32
        elif datatype == 8:  # ACL_UINT32
            np_type = np.uint32
        else:
            print("unsurpport datatype ", datatype)
            return

        # 将解码后的数据组织为numpy数组,并设置shape和类型
        return np.frombuffer(byte_array, dtype=np_type).reshape(shape)

    def _output_dataset_to_numpy(self):
        ''' 将模型输出解码为numpy数组 '''
        dataset = []
        # 遍历每个输出
        for i in range(self._output_num):
            buffer = acl.mdl.get_dataset_buffer(self.output_dataset, i)  # 从输出dataset中获取buffer
            data_ptr = acl.get_data_buffer_addr(buffer)  # 获取输出数据内存地址
            size = acl.get_data_buffer_size(buffer)  # 获取输出数据字节数
            narray = acl.util.ptr_to_bytes(data_ptr, size)  # 将指针转为字节流数据

            # 根据模型输出的shape和数据类型,将内存数据解码为numpy数组
            dims = acl.mdl.get_output_dims(self.model_desc, i)[0]["dims"]  # 获取每个输出的维度
            datatype = acl.mdl.get_output_data_type(self.model_desc, i)  # 获取每个输出的数据类型
            output_nparray = self._unpack_bytes_array(narray, tuple(dims), datatype)  # 解码为numpy数组
            dataset.append(output_nparray)
        return dataset

    def execute(self, input_list):
        '''创建输入dataset对象, 推理完成后, 将输出数据转换为numpy格式'''
        self._gen_input_dataset(input_list)  # 创建模型输入dataset结构
        ret = acl.mdl.execute(self.model_id, self.input_dataset, self.output_dataset)  # 调用离线模型的execute推理数据
        out_numpy = self._output_dataset_to_numpy()  # 将推理输出的二进制数据流解码为numpy数组, 数组的shape和类型与模型输出规格一致
        return out_numpy

    def release(self):
        ''' 释放模型相关资源 '''
        if self._is_released:
            return
```



```
        print("Model start release...")
        self._release_dataset(self.input_dataset)  # 释放输入数据结构
        self.input_dataset = None  # 将输入数据置空
        self._release_dataset(self.output_dataset)  # 释放输出数据结构
        self.output_dataset = None  # 将输出数据置空

        if self.model_id:
            ret = acl.mdl.unload(self.model_id)  # 卸载模型
        if self.model_desc:
            ret = acl.mdl.destroy_desc(self.model_desc)  # 释放模型描述信息
        self._is_released = True
        print("Model release source success")

    def _release_dataset(self, dataset):
        ''' 释放 aclmdlDataset 类型数据 '''
        if not dataset:
            return
        num = acl.mdl.get_dataset_num_buffers(dataset)  # 获取数据集包含的buffer个数
        for i in range(num):
            data_buf = acl.mdl.get_dataset_buffer(dataset, i)  # 获取buffer指针
            if data_buf:
                ret = acl.destroy_data_buffer(data_buf)  # 释放buffer
        ret = acl.mdl.destroy_dataset(dataset)  # 销毁数据集

    @abstractmethod
    def infer(self, inputs): # 保留接口, 子类必须重写
        pass
```

4. 定义yolo模型的具体推理功能，包含前处理、推理、后处理等功能。YoloV5继承自3中的Model，并重写其推理接口（infer函数），得到模型推理输出结果。

```
class YoloV5(Model):
    def __init__(self, model_path):
        super().__init__(model_path)
        self.neth = 640  # 缩放的目标高度, 也即模型的输入高度
        self.netw = 640  # 缩放的目标宽度, 也即模型的输入宽度
        self.conf_threshold = 0.1  # 置信度阈值

    def infer(self, img_bgr):

        labels_dict = get_labels_from_txt('./coco_names.txt')  # 得到类别信息，返回序号与类别对应的字典

        # 数据前处理
        img, scale_ratio, pad_size = letterbox(img_bgr, new_shape=[640, 640])  # 对图像进行缩放与填充
        img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, HWC to CHW
        img = np.ascontiguousarray(img, dtype=np.float32) / 255.0  # 转换为内存连续存储的数组

        # 模型推理, 得到模型输出
        output = self.execute([img, ])[0]

        # 后处理
        boxout = nms(torch.tensor(output), conf_thres=0.4, iou_thres=0.5)  # 利用非极大值抑制处理模型输出，conf_thres 为置信度阈值，iou_thres 为iou阈值
        pred_all = boxout[0].numpy()  # 转换为numpy数组
        scale_coords([640, 640], pred_all[:, :4], img_bgr.shape, ratio_pad=(scale_ratio, pad_size))  # 将推理结果缩放到原始图片大小
        img_dw = draw_bbox(pred_all, img_bgr, (0, 255, 0), 2, labels_dict)  # 画出检测框、类别、概率
        return img_dw
```

5. 定义一个acl初始化“main”函数。

```
if __name__ == '__main__':
    context = init_acl(DEVICE_ID)  # 初始化acl相关资源
    det_model = YoloV5(model_path=trained_model_path)  # 初始化模型

    # 读入文件并推理
    img = cv2.imread(image_path, cv2.IMREAD_COLOR)  # 读入图片
    img_res = det_model.infer(img)  # 前处理、推理、后处理, 得到最终推理图片
    cv2.imwrite('img_res.png', img_res)

    # 释放相关资源
```



```
    det_model.release()  # 释放 acl 模型相关资源, 包括输入数据、输出数据、模型等
    deinit_acl(context, 0)  # acl 去初始化
```

运行推理

进入“yolo_acl_sample/infer”目录，运行主程序“main.py”。

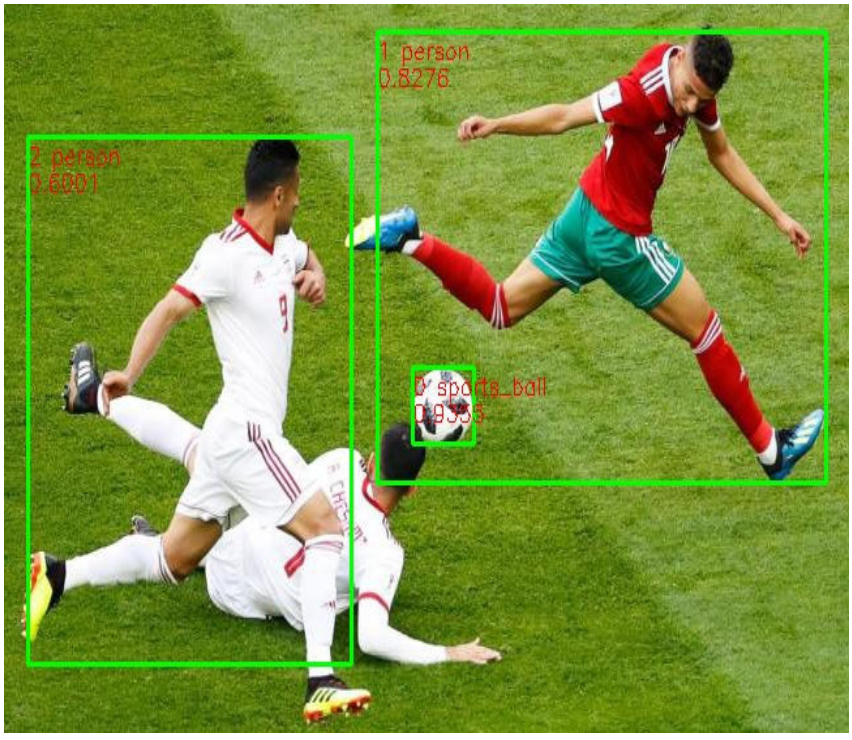
```
cd infer
python main.py
```

界面显示结果如下。

```
Init ACL Successfully
load model yolov5s_bs1.om
Init model resource
[Model] Model init resource stage success
[Model] create model output dataset success
[Model] create model input dataset success
start infer image: test.jpg
Model start release...
Model release source success
Deinit ACL Successfully
```

在“yolo_acl_sample/infer”目录下，保存相应的推理结果：img_res.png。

图 2-3 推理结果图



样例总结与扩展

以上代码包括以下几个步骤：

1. 初始化acl资源：在调用AscendCL相关资源时，必须先初始化AscendCL，否则可能会导致后续系统内部资源初始化出错。此样例中，包括指定计算设备、创建context等操作，再初始化了模型类，初始化时，进行了模型的加载以及输出数据集的创建。

2. 推理：读入图片，调用model.infer进行推理，其中包含数据的前处理、输入数据集结构的创建、推理、将推理结果转换为numpy、并进行后处理等操作，得到最终带有检测框的图片结果，最后将结果保存到图片。

3. 资源销毁：最后记得释放相关资源，包括卸载模型、销毁输入输出数据集、释放Context、释放指定的计算设备、以及AscendCL去初始化等操作。

AscendCL接口分类总结：

分类	接口函数	描述
AscendCL初始化相关	acl.init()	pyACL初始化函数
	acl.rt.set_device(device_id)	指定当前进程或线程中用于运算的Device，同时隐式创建默认Context
	acl.rt.create_context(device_id)	在当前进程或线程中显式创建一个Context
模型描述信息相关	acl.mdl.load_from_file(model_path)	从文件加载离线模型数据（适配昇腾AI处理器的离线模型）
	acl.mdl.create_desc()	创建aclmdlDesc类型的数据
	acl.mdl.get_desc(model_desc, model_id)	根据模型ID获取该模型的aclmdlDesc类型数据
	acl.mdl.get_num_inputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输入个数
	acl.mdl.get_num_outputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输出个数
	acl.mdl.get_output_size_by_index(model_desc, i)	根据aclmdlDesc类型的数据，获取指定输出的大小，单位为Byte
	acl.mdl.get_output_dims(model_desc, i)	根据模型描述信息获取指定的模型输出tensor的维度信息
	acl.mdl.get_output_data_type(model_desc, i)	根据模型描述信息获取模型中指定输出的数据类型
数据集结构相关	acl.mdl.create_dataset()	创建aclmdlDataset类型的数据
	acl.create_data_buffer(data_addr, size)	创建aclDataBuffer类型的数据，该数据类型用于描述内存地址、大小等内存信息
	acl.mdl.add_dataset_buffer(dataset, date_buffer)	向aclmdlDataset中增加aclDataBuffer
	acl.mdl.get_dataset_num_buffers(dataset)	获取aclmdlDataset中aclDataBuffer的个数
	acl.mdl.get_dataset_buffer(dataset, i)	获取aclmdlDataset中的第i个aclDataBuffer
	acl.get_data_buffer_addr(buffer)	获取aclDataBuffer类型中的数据的地址对象

分类	接口函数	描述
	acl.get_data_buffer_size(buffer)	获取aclDataBuffer类型中数据的内存大小，单位Byte
	acl.util.bytes_to_ptr(bytes_data)	将bytes对象转换成为void*数据，可以将转换好的数据传递给C函数直接使用
	acl.util.ptr_to_bytes(ptr, size)	将void*数据转换为bytes对象，可以使python代码直接访问
	acl.rt.malloc(size, policy)	申请Device上的内存
推理相关	acl.mdl.execute(model_id, input_dataset, output_dataset)	执行模型推理，直到返回推理结果
销毁资源相关	acl.destroy_data_buffer(data_buffer)	销毁aclDataBuffer类型的数据
	acl.mdl.destroy_dataset(dataset)	销毁aclmdlDataset类型的数据
	acl.mdl.unload(model_id)	系统完成模型推理后，可调用该接口卸载模型，释放资源
	acl.mdl.destroy_desc(model_desc)	销毁aclmdlDesc类型的数据
	acl.rt.destroy_context(context)	销毁一个Context，释放Context的资源
	acl.rt.reset_device(device_id)	复位当前运算的Device，释放Device上的资源，包括默认Context、默认Stream以及默认Context下创建的所有Stream
	acl.finalize()	pyACL去初始化函数，用于释放进程内的pyACL相关资源

理解各个接口含义后，用户可进行灵活运用。除此外，此样例中只示范了图片推理，若需要对视频流数据进行推理，可用三种方式输入视频流数据：USB摄像头和手机摄像头。具体使用方式可参考《**摄像头拉流**》，用户只需将前处理、推理及后处理代码放入摄像头推理代码的循环中即可，注意有些细节地方需进行相应修改，具体逻辑可参照图像分类应用中的样例总结与扩展。

3 图像分割应用样例开发介绍（Python）

本章节介绍基于AscendCL接口如何开发一个基于Unet++模型的图像分割样例。

样例介绍

图像分割应用，即将图片各个部分进行识别划分。

图 3-1 图片分割应用

本例中使用的是**开源数据集**训练的Unet++图片分割模型（参见**ModelZoo-PyTorch-Unet++**）。可以直接使用训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以基于算法、框架构建适合的模型。

模型的输入数据与输出数据格式：

- 输入数据：RGB格式图片，分辨率为 96*96，输入形状为（1，3，96，96），也即（batchsize，channel，height，width）。
- 输出数据：分割结果灰度图，分辨率也为 96*96，输入形状为（1，1，96，96），也即（batchsize，channel，height，width）。

> **说明**
>
> 输出数组需要经过一定后处理，才能显示为正常的图片。

业务模块介绍

业务模块的操作流程如**图3-2**所示。

图 3-2 业务流程图

1. 图片输入：收集待分割的数据集做为输入数据。
2. 输入预处理：将图片或视频读入为数组，并缩放到模型所需大小，然后调整像素范围。
3. 模型推理：经过模型推理后得到图片分割结果。
4. 后处理：调整模型输出值的范围，将其缩放到原始图片大小，并在原始图像上画出分割结果。
5. 可视化或保存到文件：将分割后的图片显示在界面或保存到文件。

获取代码

步骤1 获取代码文件。

单击**获取链接**或使用wget命令，下载代码文件压缩包，以**root**用户登录开发者套件。

```
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/Atlas%20200I%20DK%20A2/DevKit/models/sdk_cal_samples/Unetplusplus_acl_sample.zip
```

步骤2 将“Unetplusplus_acl_sample.zip”压缩包上传到开发者套件，解压并进入解压后的目录。

```
unzip Unetplusplus_acl_sample.zip
cd Unetplusplus_acl_sample
```

代码目录结构如下所示，按照正常开发流程，需要将框架模型文件转换成昇腾AI处理器支持推理的om格式模型文件，鉴于当前是入门内容，用户可直接获取已转换好的om模型进行推理。

```
|-- Unetplusplus_acl_sample
   |-- unetplusplus.om          # om模型
   |-- main.py                  # 主程序
   |-- img.png                  # 测试图片
```

测试图片来自数据集 **2018 Data Science Bowl | Kaggle** ，用户可自行更改测试图片。

----结束

代码解析

开发代码过程中，在“Unetplusplus_acl_sample/main.py”文件中已包含读入数据、前处理、推理、后处理等功能，串联整个应用代码逻辑，此处仅对代码进行解析。

1. 在“main.py”文件的开头有如下代码，导入需要的第三方库以及调用AscendCL接口推理所需文件。

```
# 引用库
#!/usr/bin/python
# -*- coding: utf-8 -*-

import cv2  # 图片处理三方库，用于对图片进行前后处理
import numpy as np  # 用于对多维数组进行计算
from albumentations.augmentations import transforms  # 数据增强库，用于对图片进行变换
```

```
import acl  # AscnedCL推理所需库文件
import constants as const  # 其中包含acl相关状态值，用于判断推理时程序状态是否正常
```

2. 定义模型相关变量，如图片路径、模型路径、类别数量等，并进行acl的资源初始化，使用AscendCL接口开发应用时，必须先初始化AscendCL，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。

```
# 初始化变量
pic_input = 'img.png'  # 单张图片
model_path = "unetplusplus.om"  # 模型路径
num_class = 1  # 类别数量, 需要根据模型结构、任务类别进行改变; 此处只分割出细胞, 即为一分类
device_id = 0  # 指定运算的Device

# acl初始化
print("init resource stage:")
ret = acl.init()
ret = acl.rt.set_device(device_id)  # 指定运算的Device
context, ret = acl.rt.create_context(device_id)  # 显式创建一个Context
print("Init resource success")
```

3. 模型加载。使用acl接口加载om模型。

```
# 加载模型
model_id, ret = acl.mdl.load_from_file(model_path)  # 加载离线模型文件, 返回标识模型的ID
model_desc = acl.mdl.create_desc()  # 初始化模型描述信息, 包括模型输入个数、输入维度、输出个数、输出维度等信息
ret = acl.mdl.get_desc(model_desc, model_id)  # 根据加载成功的模型的ID, 获取该模型的描述信息
print("Init model resource success")
```

4. 准备输入数据集。先使用opencv读入图片，得到三维数组，再进行相应的图片大小缩放、像素值缩放等处理，并将其转化为模型推理所需要的数据集格式。

```
# 前处理
img_bgr = cv2.imread(pic_input)  # 读入图片
img = cv2.resize(img_bgr, (96, 96))  # 将原图缩放到 96*96 大小
img= transforms.Normalize().apply(img)  # 将像素值标准化（减去均值除以方差）
img = img.astype('float32') / 255  # 将像素值缩放到 0~1 范围内
img = img.transpose(2, 0, 1)  # 将形状转换为 channel first (3, 96, 96)

# 准备输入数据集
input_buffer = []  # 初始化输入buffer列表
input_list = [img, ]  # 初始化输入数据列表
input_num = acl.mdl.get_num_inputs(model_desc)  # 得到模型输入个数
for i in range(input_num):  # 根据模型输入个数, 初始化输入 buffer
    item = {"addr": None, "size": 0}  # 每个输入的地址、所占字节数
    input_buffer.append(item)  # 加入buffer列表

input_dataset = acl.mdl.create_dataset()  # 创建输入数据
for i in range(input_num):
    input_data = input_list[i]  # 得到每个输入数据

    # 得到每个输入数据流的指针(input_ptr)和所占字节数(size)
    size = input_data.size * input_data.itemsize  # 得到所占字节数
    bytes_data=input_data.tobytes()  # 将每个输入数据转换为字节流
    input_ptr=acl.util.bytes_to_ptr(bytes_data)  # 得到输入数据指针

    model_size = acl.mdl.get_input_size_by_index(model_desc, i)  # 从模型信息中得到输入所占字节数
    if size != model_size:  # 判断所分配的内存是否和模型的输入大小相符
        print(" Input[%d] size: %d not equal om size: %d" % (i, size, model_size) + ", may cause
inference result error, please check model input")

    dataset_buffer = acl.create_data_buffer(input_ptr, size)  # 为每个输入创建 buffer
    _, ret = acl.mdl.add_dataset_buffer(input_dataset, dataset_buffer)  # 将每个 buffer 添加到输入数据中
print("Create model input dataset success")
```

5. 准备输出数据集。构造输出数据集数据结构，为其分配内存，为之后的推理步骤做准备。

```
# 准备输出数据集
output_size = acl.mdl.get_num_outputs(model_desc)  # 得到模型输出个数
output_dataset = acl.mdl.create_dataset()  # 创建输出数据
for i in range(output_size):
    size = acl.mdl.get_output_size_by_index(model_desc, i)  # 得到每个输出所占内存大小
```



```
    buf, ret = acl.rt.malloc(size, const.ACL_MEM_MALLOC_NORMAL_ONLY)  # 为输出分配内存
    dataset_buffer = acl.create_data_buffer(buf, size)  # 为每个输出创建 buffer
    _, ret = acl.mdl.add_dataset_buffer(output_dataset, dataset_buffer)  # 将每个 buffer 添加到输出数据
中
    if ret:  # 若分配出现错误, 则释放内存
        acl.rt.free(buf)
        acl.destroy_data_buffer(dataset_buffer)
print("Create model output dataset success")
```

6. 使用acl接口进行模型推理，得到模型输出结果。

```
# 模型推理, 得到的输出将写入 output_dataset 中
ret = acl.mdl.execute(model_id, input_dataset, output_dataset)
if ret != const.ACL_SUCCESS:  # 判断推理是否出错
    print("Execute model failed for acl.mdl.execute error ", ret)
```

7. 对模型输出进行解析，再进行后处理，保存推理后的图片。由于模型推理后的数据以buffer的形式存储在output_dataset中，需要将其提取出，并转换为利于处理的numpy数组，再进行后处理。后处理步骤即将像素值变换到 0~1范围内后，再将其画在原图上，得到最终可以用于显示的的分割结果。

```
# 解析 output_dataset, 得到模型输出列表
model_output = [] # 模型输出列表
for i in range(output_size):
    buf = acl.mdl.get_dataset_buffer(output_dataset, i)  # 获取每个输出buffer
    data_addr = acl.get_data_buffer_addr(buf)  # 获取输出buffer的地址
    size = int(acl.get_data_buffer_size(buf))  # 获取输出buffer的字节数
    byte_data = acl.util.ptr_to_bytes(data_addr, size)  # 将指针转为字节流数据
    dims = tuple(acl.mdl.get_output_dims(model_desc, i)[0]["dims"])  # 从模型信息中得到每个输出的维
度信息
    output_data = np.frombuffer(byte_data, dtype=np.float32).reshape(dims)  # 将 output_data 以流的
形式读入转化成 ndarray 对象
    model_output.append(output_data) # 添加到模型输出列表

# 后处理
model_out_msk = model_output[0]  # 取出模型推理结果, 推理结果形状为 (1, 1, 96, 96),即（batchsize,
num_class, height, width）
model_out_msk = sigmoid(model_out_msk[0][0])  # 将模型输出变换到 0~1 范围内
img_to_save = plot_mask(img_bgr, model_out_msk)  # 将处理后的输出画在原图上, 并返回

# 保存图片到文件
cv2.imwrite('result.png', img_to_save)
```

8. 释放输入、输出、模型等资源。

```
# 释放输入资源, 包括数据结构和内存
num = acl.mdl.get_dataset_num_buffers(input_dataset)  # 获取输入个数
for i in range(num):
    data_buf = acl.mdl.get_dataset_buffer(input_dataset, i)  # 获取每个输入buffer
    if data_buf:
        ret = acl.destroy_data_buffer(data_buf)  # 销毁每个输入buffer (销毁 aclDataBuffer 类型)
ret = acl.mdl.destroy_dataset(input_dataset)  # 销毁输入数据 (销毁 aclmdlDataset类型的数据)

# 释放输出资源, 包括数据结构和内存
num = acl.mdl.get_dataset_num_buffers(output_dataset)  # 获取输出个数
for i in range(num):
    data_buf = acl.mdl.get_dataset_buffer(output_dataset, i)  # 获取每个输出buffer
    if data_buf:
        ret = acl.destroy_data_buffer(data_buf)  # 销毁每个输出buffer (销毁 aclDataBuffer 类型)
ret = acl.mdl.destroy_dataset(output_dataset)  # 销毁输出数据 (销毁 aclmdlDataset类型的数据)

# 卸载模型
if model_id:
    ret = acl.mdl.unload(model_id)

# 释放模型描述信息
if model_desc:
    ret = acl.mdl.destroy_desc(model_desc)

# 释放 Context
if context:
    acl.rt.destroy_context(context)
```



```
# 释放Device
acl.rt.reset_device(device_id)
acl.finalize()
print("Release acl resource success")
```

9. 主体函数定义。

   其中sigmoid函数用于将矩阵的每个元素变换到0~1范围内，plot_mask用于将得到的分割结果画在原图上。

   以下代码定义了这两个函数功能。

```
def sigmoid(x):
    y = 1.0 / (1 + np.exp(-x))  # 对矩阵的每个元素执行 1/(1+e^(-x))
    return y


def plot_mask(img, msk):
    """ 将推理得到的 mask 覆盖到原图上 """
    msk = msk + 0.5  # 将像素值范围变换到 0.5~1.5, 有利于下面转为二值图
    msk = cv2.resize(msk, (img.shape[1], img.shape[0]))  # 将 mask 缩放到原图大小
    msk = np.array(msk, np.uint8)  # 转为二值图, 只包含 0 和 1

    # 从 mask 中找到轮廓线, 其中第二个参数为轮廓检测的模式, 第三个参数为轮廓的近似方法
    # cv2.RETR_EXTERNAL 表示只检测外轮廓，cv2.CHAIN_APPROX_SIMPLE 表示压缩水平方向、
    # 垂直方向、对角线方向的元素, 只保留该方向的终点坐标, 例如一个矩形轮廓只需要4个点来保存轮廓信
息
    # contours 为返回的轮廓（list）
    contours, _ = cv2.findContours(msk, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # 在原图上画出轮廓, 其中 img 为原图, contours 为检测到的轮廓列表
    # 第三个参数表示绘制 contours 中的哪条轮廓, -1 表示绘制所有轮廓
    # 第四个参数表示颜色,（0, 0, 255）表示红色, 第五个参数表示轮廓线的宽度
    cv2.drawContours(img, contours, -1, (0, 0, 255), 1)

    # 将轮廓线以内（即分割区域）覆盖上一层红色
    img[..., 2] = np.where(msk == 1, 255, img[..., 2])

    return img
```

运行推理

进入“Unetplusplus_acl_sample”目录，运行主程序main.py 。

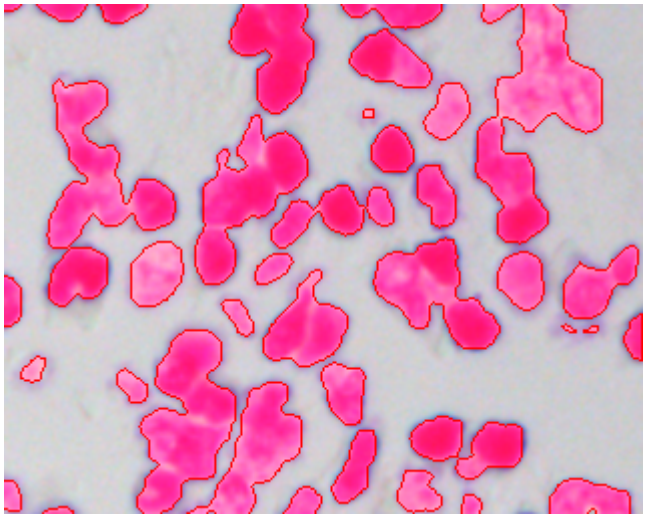
```
python main.py
```

界面显示结果如下。

```
init resource stage:
Init resource success
Init model resource success
Create model input dataset success
Create model output dataset success
Release acl resource success
```

推理完成后，在当前文件夹下生成“result.png”文件，如**图3-3**所示。

图 3-3 推理结果图



样例总结与扩展

以上代码包括以下几个步骤：

从AscendCL初始化、指定计算设备，到加载模型、准备测试数据，再到执行推理、处理推理的结果数据，最后到释放各类资源，包括卸载模型、释放内存、销毁各种数据类型等，最后释放指定的计算设备、进行AscendCL去初始化

1. 初始化acl资源：此样例中，包括指定计算设备、创建context、创建stream等操作，再进行了模型的加载。

2. 前处理：包括缩放、标准化、数据类型转换以及维度转换等操作。

3. 推理：包括准备输入数据集结构、准备输出数据集结构、并利用acl.mdl.execute()接口进行推理等，并将结果解析为numpy array类型。

4. 后处理：包括sigmoid变换以及将输出画在原图上，并保存图片。

5. 资源销毁：最后记得释放相关资源，包括卸载模型、销毁输入输出数据集、释放Context、释放指定的计算设备、以及AscendCL去初始化等操作。

AscendCL接口分类总结：

分类	接口函数	描述
AscendCL初始化相关	acl.init()	pyACL初始化函数
	acl.rt.set_device(device_id)	指定当前进程或线程中用于运算的Device，同时隐式创建默认Context
	acl.rt.create_context(device_id)	在当前进程或线程中显式创建一个Context
模型描述信息相关	acl.mdl.load_from_file(model_path)	从文件加载离线模型数据（适配昇腾AI处理器的离线模型）
	acl.mdl.create_desc()	创建aclmdlDesc类型的数据

分类	接口函数	描述
	acl.mdl.get_desc(model_desc, model_id)	根据模型ID获取该模型的aclmdlDesc类型数据
	acl.mdl.get_num_inputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输入个数
	acl.mdl.get_num_outputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输出个数
	acl.mdl.get_output_size_by_index(model_desc, i)	根据aclmdlDesc类型的数据，获取指定输出的大小，单位为Byte
	acl.mdl.get_output_dims(model_desc, i)	根据模型描述信息获取指定的模型输出tensor的维度信息
数据集结构相关	acl.mdl.create_dataset()	创建aclmdlDataset类型的数据
	acl.create_data_buffer(data_addr, size)	创建aclDataBuffer类型的数据，该数据类型用于描述内存地址、大小等内存信息
	acl.mdl.add_dataset_buffer(dataset, date_buffer)	向aclmdlDataset中增加aclDataBuffer
	acl.mdl.get_dataset_num_buffers(dataset)	获取aclmdlDataset中aclDataBuffer的个数
	acl.mdl.get_dataset_buffer(dataset, i)	获取aclmdlDataset中的第i个aclDataBuffer
	acl.get_data_buffer_addr(buffer)	获取aclDataBuffer类型中的数据的地址对象
	acl.get_data_buffer_size(buffer)	获取aclDataBuffer类型中数据的内存大小，单位Byte
	acl.util.bytes_to_ptr(bytes_data)	将bytes对象转换成为void*数据，可以将转换好的数据传递给C函数直接使用
	acl.util.ptr_to_bytes(ptr, size)	将void*数据转换为bytes对象，可以使python代码直接访问
	acl.rt.malloc(size, policy)	申请Device上的内存
	acl.rt.free(data_buffer)	释放通过acl.rt.malloc申请的内存
推理相关	acl.mdl.execute(model_id, input_dataset, output_dataset)	执行模型推理，直到返回推理结果
销毁资源相关	acl.destroy_data_buffer(data_buffer)	销毁aclDataBuffer类型的数据
	acl.mdl.destroy_dataset(dataset)	销毁aclmdlDataset类型的数据

分类	接口函数	描述
	acl.mdl.unload(model_id)	系统完成模型推理后，可调用该接口卸载模型，释放资源
	acl.mdl.destroy_desc(model_desc)	销毁aclmdlDesc类型的数据
	acl.rt.destroy_context(context)	销毁一个Context，释放Context的资源
	acl.rt.reset_device(device_id)	复位当前运算的Device，释放Device上的资源，包括默认Context、默认Stream以及默认Context下创建的所有Stream
	acl.finalize()	pyACL去初始化函数，用于释放进程内的pyACL相关资源

理解各个接口含义后，用户可进行灵活运用。除此外，此样例中只示范了图片推理，若需要对视频流数据进行推理，可用三种方式输入视频流数据：USB摄像头、手机摄像头。具体使用方式可参考《**摄像头拉流**》，用户只需将前处理、推理及后处理代码放入摄像头推理代码的循环中即可，注意有些细节地方需进行相应修改，具体逻辑可参照图像分类应用中的样例总结与扩展。

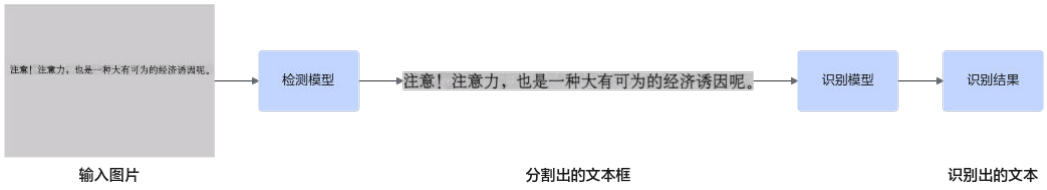
4 文本识别应用样例开发介绍（Python）

在本章节介绍基于AscendCL接口（Python语言接口）如何开发一个简单的文本检测样例。

样例介绍

文本识别应用，即标识图片中文本框的位置并识别文本框内的内容。

图 4-1 文本识别样例



本例中使用的是Mindspore框架的CTPN模型与ONNX框架的SVTR模型，其中CTPN模型为文本检测模型，能够检测出文本所在区域；SVTR模型为文字识别模型，可识别文字内容。用户可以直接使用训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以基于算法、框架构建适合的模型。

CTPN模型的基本介绍如下。

- 输入数据：BGR格式、任意分辨率的输入图片。
- 输出数据：图片中文本框的4个顶点的坐标。

SVTR模型的基本介绍如下。

- 输入数据：BGR格式、任意分辨率的输入图片。
- 输出数据：图片中字符的内容。

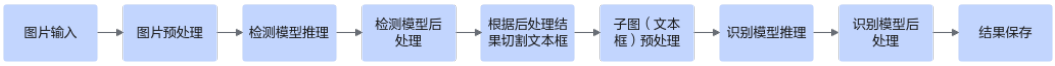
  说明

  图片中文本框的4个顶点的坐标为缩放后的图片中的坐标，需要映射回原图的坐标。

业务模块介绍

从功能模块介绍以及业务流程说明方面来介绍一下本案例相关的业务模块。

图 4-2 案例流程图



1. 图片输入：收集待识别的文本数据集做为输入数据。
2. 图片预处理：图片通过OpenCV解码，再将图片缩放至CTPN模型要求的输入大小，并把图片进行归一化。
3. 检测模型推理：CTPN模型识别图片中的文本框。
4. 模型后处理：处理推理结果，获取所有文本框的位置以及对应的置信度。
5. 切割文本框：将原图片根据文本框的位置切割成仅包含文本框的子图。
6. 子图预处理：将图片缩放至SVTR模型要求的输入大小，并把图片进行归一化。
7. 识别模型推理：SVTR模型识别图片中的文本。
8. 识别模型后处理：处理推理结果，根据字典查询结果拼接出文本框中的内容。

获取代码

步骤1 获取代码文件。

单击**获取链接**或使用wget命令，下载代码文件压缩包，以**root**用户登录开发者套件。

```
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/Atlas%20200I%20DK%20A2/DevKit/models/sdk_cal_samples/ocr_acl_sample.zip
```

步骤2 将“ocr_acl_sample.zip”压缩包上传到开发者套件，解压并进入解压后的目录。

```
unzip ocr_acl_sample.zip
cd ocr_acl_sample
```

代码目录结构如下所示，按照正常开发流程，需要将框架模型文件转换成昇腾AI处理器支持推理的om格式模型文件，鉴于当前是入门内容，用户可直接获取已转换好的om模型进行推理。

```
ocr_acl_sample
├── datas #推理图片目录
│   └── test.png
├── models #om模型放置的目录
│   ├── ctpn.om #检测模型(CTPN)的om文件
│   └── svtr.om #识别模型（SVTR）的om文件
└── src
│   └── utils.py #主程序以及模型前后处理相关函数
├── main.py # 推理主程序
├── ppocr_keys_v1.txt # 识别模型（SVTR）所需标签文件
```

----结束

代码解析

开发代码过程中，在“ocr_acl_sample/main.py”文件中已包含读入数据、前处理、推理、后处理等功能，串联整个应用代码逻辑，此处仅对代码进行解析。

1. 在“main.py”文件的开头有如下代码，用于导入第三方库与调用AscendCL接口推理所需文件。

```
import json
import math
import os
from argparse import ArgumentParser
from abc import abstractmethod, ABC
```

```
import cv2  # 图片处理三方库，用于对图片进行前后处理
import numpy as np  # 用于对多维数组进行计算
from PIL import Image, ImageDraw  # 图像处理库，此处用于读入图片并画出结果

import acl

from src.utils import get_images_from_path, img_read, detect


SUCCESS = 0  # 成功状态值
FAILED = 1  # 失败状态值
ACL_MEM_MALLOC_NORMAL_ONLY = 2  # 申请内存策略, 仅申请普通页
```

2. 获取输入的参数。

```
def parse_args():
    parser = ArgumentParser()
    parser.add_argument('--image_path', type=str, required=True)  # 获取推理图片/文件夹的路径
    parser.add_argument('--det_model_path', type=str, required=True)  # 获取检测模型（CTPN）的路径
    parser.add_argument('--rec_model_path', type=str, required=True)  # 获取识别模型（SVTR）的路径
    parser.add_argument('--rec_model_dict', type=str, required=True)  # 获取识别模型字典的路径
    parser.add_argument('--device_id', type=int, required=False, default=0)  # 获取推理使用的NPU的编
号
    return parser.parse_args()
```

3. 资源初始化。

   使用AscendCL接口开发应用时，必须先初始化AscendCL ，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。有初始化就要去初始化，在确定完成了AscendCL的所有调用之后，或者进程退出之前，需调用接口实现AscendCL去初始化。以下两个函数实现了acl资源的初始化和去初始化。

```
def init_acl(device_id):
    acl.init()  # 初始化 acl
    ret = acl.rt.set_device(device_id)  # 初始化 NPU
    if ret:
        raise RuntimeError(ret)
    context, ret = acl.rt.create_context(device_id) # 创建context
    if ret:
        raise RuntimeError(ret)
    print('Init ACL Successfully')
    return context


def deinit_acl(context, device_id):
    ret = acl.rt.destroy_context(context)  # 销毁context
    if ret:
        raise RuntimeError(ret)

    ret = acl.rt.reset_device(device_id)  # 重置 NPU
    if ret:
        raise RuntimeError(ret)
    ret = acl.finalize()  # 检查acl相关资源是否全部释放
    if ret:
        raise RuntimeError(ret)
    print('Deinit ACL Successfully')
```

4. 定义模型资源相关基类，承担初始化模型资源、创建输入输出数据集、执行推理、解析输出、释放模型资源等功能。之后的CTPN与SVTR模型推理可继承此类。

```
class Model(ABC):
    def __init__(self, model_path):
        print(f"load model {model_path}")
        self.model_path = model_path  # 模型路径
        self.model_id = None  # 模型 id
        self.input_dataset = None  # 输入数据结构
        self.output_dataset = None  # 输出数据结构
        self.model_desc = None  # 模型描述信息
        self._input_num = 0  # 输入数据个数
        self._output_num = 0  # 输出数据个数
        self._output_info = []  # 输出信息列表
```

```
        self._is_released = False  # 资源是否被释放
        self.model_height = None  # 模型输入图片高度
        self.model_width = None  # 模型输入图片宽度
        self.model_channel = None  # 模型输入图片通道数
        self._init_resource()

    def _init_resource(self):
        ''' 初始化模型、输入、输出相关资源 '''
        print("Init model resource")
        # 加载模型文件
        self.model_id, ret = acl.mdl.load_from_file(self.model_path)  # 加载模型
        self.model_desc = acl.mdl.create_desc()  # 初始化模型信息对象
        ret = acl.mdl.get_desc(self.model_desc, self.model_id)  # 根据模型获取描述信息
        print("[Model] Model init resource stage success")

        # 创建模型输出 dataset 结构
        self._output_num = acl.mdl.get_num_outputs(self.model_desc)  # 获取模型输出个数
        self._gen_output_dataset()  # 创建模型输出dataset结构
        for i in range(self._output_num):
            dims = acl.mdl.get_output_dims(self.model_desc, i)[0]["dims"]  # 获取每个输出的维度
            datatype = acl.mdl.get_output_data_type(self.model_desc, i)  # 获取每个输出的数据类型
            self._output_info.append({"shape": tuple(dims), "type": datatype})  # 将维度和数据类型加入输出信息列表

        # 获取模型输入个数, 为创建输入dataset结构做准备
        self._input_num = acl.mdl.get_num_inputs(self.model_desc)

        # 获取模型输入的shape，为预处理做准备
        dims, ret = acl.mdl.get_input_dims_v2(self.model_desc, 0)
        if ret:
            raise RuntimeError(ret)
        dims = dims['dims']

        # 获取模型输入图片的通道数、高、宽
        self.model_channel = dims[1]
        self.model_height = dims[2]
        self.model_width = dims[3]

    def _gen_output_dataset(self):
        ''' 组织输出数据的dataset结构 '''
        ret = SUCCESS
        self.output_dataset = acl.mdl.create_dataset()  # 创建输出dataset结构
        for i in range(self._output_num):
            temp_buffer_size = acl.mdl.get_output_size_by_index(self.model_desc, i)  # 获取模型输出个数
            temp_buffer, ret = acl.rt.malloc(temp_buffer_size, ACL_MEM_MALLOC_NORMAL_ONLY)  # 为每个输出申请device内存
            dataset_buffer = acl.create_data_buffer(temp_buffer, temp_buffer_size)  # 创建输出的data buffer结构,将申请的内存填入data buffer
            _, ret = acl.mdl.add_dataset_buffer(self.output_dataset, dataset_buffer)  # 将 data buffer 加入输出dataset

        if ret == FAILED:
            self._release_dataset(self.output_dataset)  # 失败时释放dataset
        print("[Model] create model output dataset success")

    def _gen_input_dataset(self, input_list):
        ''' 组织输入数据的dataset结构 '''
        ret = SUCCESS
        self.input_dataset = acl.mdl.create_dataset()  # 创建输入dataset结构
        for i in range(self._input_num):
            item = input_list[i]  # 获取第 i 个输入数据
            data = acl.util.bytes_to_ptr(item.tobytes())  # 获取输入数据字节流
            size = item.size * item.itemsize  # 获取输入数据字节数
            dataset_buffer = acl.create_data_buffer(data, size)  # 创建输入dataset buffer结构, 填入输入数据
            _, ret = acl.mdl.add_dataset_buffer(self.input_dataset, dataset_buffer)  # 将dataset buffer加入dataset

        if ret == FAILED:
```

```
            self._release_dataset(self.input_dataset)  # 失败时释放dataset
        print("[Model] create model input dataset success")

    def _unpack_bytes_array(self, byte_array, shape, datatype):
        ''' 将内存不同类型的数据解码为numpy数组 '''
        np_type = None

        # 获取输出数据类型对应的numpy数组类型和解码标记
        if datatype == 0:  # ACL_FLOAT
            np_type = np.float32
        elif datatype == 1:  # ACL_FLOAT16
            np_type = np.float16
        elif datatype == 3:  # ACL_INT32
            np_type = np.int32
        elif datatype == 8:  # ACL_UINT32
            np_type = np.uint32
        elif datatype == 12:
            np_type = np.bool8
        else:
            print("unsurpport datatype ", datatype)
            return

        # 将解码后的数据组织为numpy数组,并设置shape和类型
        return np.frombuffer(byte_array, dtype=np_type).reshape(shape)

    def _output_dataset_to_numpy(self):
        ''' 将模型输出解码为numpy数组 '''
        dataset = []
        # 遍历每个输出
        for i in range(self._output_num):
            buffer = acl.mdl.get_dataset_buffer(self.output_dataset, i)  # 从输出dataset中获取buffer
            data = acl.get_data_buffer_addr(buffer)  # 获取输出数据内存地址
            size = acl.get_data_buffer_size(buffer)  # 获取输出数据字节数
            narray = acl.util.ptr_to_bytes(data, size)  # 将指针转为字节流数据

            # 根据模型输出的shape和数据类型,将内存数据解码为numpy数组
            output_nparray = self._unpack_bytes_array(narray, self._output_info[i]["shape"],
self._output_info[i]["type"])
            dataset.append(output_nparray)
        return dataset

    def execute(self, input_list):
        '''创建输入dataset对象, 推理完成后, 将输出数据转换为numpy格式'''
        self._gen_input_dataset(input_list)  # 创建模型输入dataset结构
        ret = acl.mdl.execute(self.model_id, self.input_dataset, self.output_dataset)  # 调用离线模型的
execute推理数据
        out_numpy = self._output_dataset_to_numpy()  # 将推理输出的二进制数据流解码为numpy数组,
数组的shape和类型与模型输出规格一致
        return out_numpy

    def release(self):
        ''' 释放模型相关资源 '''
        if self._is_released:
            return

        print("Model start release...")
        self._release_dataset(self.input_dataset)  # 释放输入数据结构
        self.input_dataset = None  # 将输入数据置空
        self._release_dataset(self.output_dataset)  # 释放输出数据结构
        self.output_dataset = None  # 将输出数据置空

        if self.model_id:
            ret = acl.mdl.unload(self.model_id)  # 卸载模型
        if self.model_desc:
            ret = acl.mdl.destroy_desc(self.model_desc)  # 释放模型描述信息
        self._is_released = True
        print("Model release source success")

    def _release_dataset(self, dataset):
```



```
        ''' 释放 aclmdlDataset 类型数据 '''
        if not dataset:
            return
        num = acl.mdl.get_dataset_num_buffers(dataset)  # 获取数据集包含的buffer个数
        for i in range(num):
            data_buf = acl.mdl.get_dataset_buffer(dataset, i)  # 获取buffer指针
            if data_buf:
                ret = acl.destroy_data_buffer(data_buf)  # 释放buffer
        ret = acl.mdl.destroy_dataset(dataset)  # 销毁数据集

    @abstractmethod
    def infer(self, inputs): # 保留接口, 子类必须重写
        pass
```

5. 定义CTPN模型的具体推理功能，包含前处理、推理、后处理等功能，得到模型推理输出结果。CTPN继承自 src 文件夹下 model.py 中的Model基类，并重写了其infer功能。此处的Model基类包含了初始化模型资源、创建输入输出数据集、执行推理、解析输出、释放模型资源等功能，具体可参见步骤4。另外，此基类的定义也与目标检测应用样例中的步骤6相似（有少许不同），开发者可根据需要改变此基类的定义。

```
class CTPN(Model):
    def __init__(self, model_path):
        super().__init__(model_path)
        self.mean = np.array([123.675, 116.28, 103.53]).reshape((1, 1, 3)).astype(np.float32)  # 定义均值
        self.std = np.array([58.395, 57.12, 57.375]).reshape((1, 1, 3)).astype(np.float32)  # 定义标准差

    def infer(self, inputs):
        # 前处理
        dst_img = cv2.resize(inputs, (int(self.model_width), int(self.model_height))).astype(np.float32)  #
缩放图片至模型要求的shape，并将数据类型转为fp32
        dst_img -= self.mean  # 标准化
        dst_img /= self.std
        dst_img = dst_img.transpose((2, 0, 1))  # 更改图片格式从 HWC 到 CHW
        dst_img = np.expand_dims(dst_img, axis=0) # 更改图片格式从 CHW to NCHW
        dst_img = np.ascontiguousarray(dst_img).astype(np.float32)  # 转换到连续内存

        # 推理
        output = self.execute([dst_img, ])

        # 后处理
        proposal = output[0]  # 获取 proposals
        proposal_mask = output[1]  # 获取proposal的mask
        all_box_tmp = proposal
        all_mask_tmp = np.expand_dims(proposal_mask, axis=1)  # 将mask扩充一维
        using_boxes_mask = all_box_tmp * all_mask_tmp  # proposal * mask 获取到真正的proposal
        textsegs = using_boxes_mask[:, 0:4].astype(np.float32)  # 区分出分割信息与置信度信息
        scores = using_boxes_mask[:, 4].astype(np.float32)
        bboxes = detect(textsegs, scores[:, np.newaxis], (self.model_height, self.model_width))  # 根据分
割信息与置信度信息计算出文本框
        return bboxes
```

6. 与上一步骤相似，继承自 src 文件夹下 model.py 中的Model基类，定义SVTR模型推理功能。

```
class SVTR(Model):
    def __init__(self, model_path, dict_path):
        super().__init__(model_path)
        self.labels = ['']
        with open(dict_path, 'r') as f:  # 逐行读入识别模型的字典文件，加入标签列表
            labels = f.readlines()
            for char in labels:
                self.labels.append(char.strip())
        self.labels.append(' ')
        self.scale = np.float32(1 / 255)  # 用于缩放图片像素值域
        self.mean = 0.5  # 设定均值与方差
        self.std = 0.5

    def infer(self,inputs):
        # 前处理：缩放图片至模型输入要求的shape
```



```
        h, w, _ = inputs.shape
        ratio = w / h
        if math.ceil(ratio * self.model_height) > self.model_width:
            resize_w = self.model_width
        else:
            resize_w = math.ceil(ratio * self.model_height)
        img = cv2.resize(inputs, (resize_w, self.model_height))

        _, w, _ = img.shape
        padding_w = self.model_width - w  # 计算图片需要padding的长度
        img = cv2.copyMakeBorder(img, 0, 0, 0, padding_w, cv2.BORDER_CONSTANT,
value=0.).astype(np.float32)  # 使用opencv进行padding图片，并将图片转为 fp32格式
        img *= self.scale
        img -= self.mean  # 标准化
        img /= self.std
        dst_img = img.transpose((2, 0, 1))  # 更改图片格式从 HWC 到 CHW
        dst_img = np.expand_dims(dst_img, axis=0)  # 更改图片格式从 CHW to NCHW
        dst_img = np.ascontiguousarray(dst_img).astype(np.float32)  # 转换到连续内存

        # 推理
        output = self.execute([dst_img, ])

        # 后处理
        output = np.argmax(output[0], axis=2).reshape(-1)  # 将输出结果进行argmax处理，并reshape
        ans = []  # 初始化字符列表
        last_char = ''
        for i, char in enumerate(output):  # 逐个解析输出，并加入字符列表
            if char and self.labels[char] != last_char:
                ans.append(self.labels[char])
            last_char = self.labels[char]
        return ''.join(ans)
```

7. 定义一个acl初始化与运行main函数。

```
if __name__ == '__main__':
    args = parse_args()  # 读取输入参数
    context, stream = init_acl(args.device_id)  # 初始化acl相关资源
    image_lst = get_images_from_path(args.image_path)  # 获取输入图片路径中的所有图片
    det_model = CTPN(model_path=args.det_model_path)  # 创建CTPN模型对象
    rec_model = SVTR(model_path=args.rec_model_path, dict_path=args.rec_model_dict)  # 创建SVTR模
型对象
    infer_res = []  # 初始化推理结果列表
    if not os.path.exists('infer_result'):  # 创建保存推理结果的目录
        os.makedirs('infer_result')

    # 对每一张图片进行推理
    for image in image_lst:
        ans = {}  # 初始化结果字典

        img_src = img_read(image)  # 使用opencv对图片及进行解码
        basename = os.path.basename(image)  # 获取图片名称（带后缀）
        print(f'start infer image: {basename}')
        name, ext = os.path.splitext(basename)  # 获取图片名称与后缀
        image_h, image_w = img_src.shape[:2] # 获取图片的宽高
        bboxes = det_model.infer(img_src)  # 利用检测模型推理，得到结果

        im = Image.open(image)  # 打开图片
        draw = ImageDraw.Draw(im)
        ans['image_name'] = basename
        ans['result'] = []  # 初始化每张图片的推理结果
        # 遍历检测模型的结果
        for bbox in bboxes:
            bbox_detail = {}
            # 将文本框的坐标映射回原图的坐标
            x1 = int(bbox[0] / det_model.model_width * image_w)
            y1 = int(bbox[1] / det_model.model_height * image_h)
            x2 = int(bbox[2] / det_model.model_width * image_w)
            y2 = int(bbox[3] / det_model.model_height * image_h)
            draw.line([(x1, y1), (x1, y2), (x2, y2), (x2, y1), (x1, y1)], fill='red', width=2)  # 在原图上画出检
测框
            bbox_detail['bbox'] = [x1, y1, x1, y2, x2, y2, x2, y1]
```



```
            crop_img = img_src[y1:y2, x1:x2]  # 切割子图

            bbox_detail['text'] = rec_model.infer(crop_img)  # 利用识别模型推理，得到结果
            print('文字结果：',bbox_detail['text'])
            ans['result'].append(bbox_detail)  # 保存推理结果

        infer_res.append(ans)  # 将本张图片推理结果加入 infer_res
        im.save(os.path.join('infer_result', name + '_res' + ext))  # 保存推理结果图片

    det_model.release()  # 释放模型资源
    rec_model.release()
    deinit_acl(context, stream, args.device_id)  # 释放acl资源

    with open(os.path.join('infer_result', 'infer_result.json'), 'w') as f:  # 保存推理结果至json文件
        json.dump(infer_res, f, indent=4)
```

运行推理

在“ocr_acl_sample”目录下，执行以下命令。

```
python ./main.py --det_model_path=./models/ctpn.om --rec_model_path=./models/svtr.om --
rec_model_dict=./ppocr_keys_v1.txt --image_path=./datas
```

说明

- 如果执行脚本报错ModuleNotFoundError: No module named 'PIL'，则表示缺少Pillow库，请使用**pip3 install Pillow --user**命令安装Pillow库。
- 如果执行脚本报错ModuleNotFoundError: No module named 'cv2'，则表示缺少OpenCV库，请使用**pip3 install opencv-python --user**命令安装OpenCV库。
- 如果Python导入OpenCV模块报错“ImportError: libGL.so.1: cannot open shared object file: No such file or directory”，可参考**Python导入opencv模块报错，提示：ImportError: libGL.so.1: cannot open shared object file: No such file or directory**解决。

终端上屏显的结果如下, 推理结果保存在当前路径下的“infer_result”目录下：

```
Init ACL Successfully
load model ./models/ctpn.om
Init model resource
[Model] Model init resource stage success
[Model] create model output dataset success
load model ./models/svtr.om
Init model resource
[Model] Model init resource stage success
[Model] create model output dataset success
start infer image: test.png
[Model] create model input dataset success
[Model] create model input dataset success
文字结果： 开启开发者之旅
[Model] create model input dataset success
文字结果： 从入门到进阶，开启异腾开发者成长之旅
Model start release...
Model release source success
Model start release...
Model release source success
Deinit ACL Successfully
```

推理完成后，在当前infer_result文件夹下生成结果图片test_res.png：

样例总结与扩展

以上代码包括以下几个步骤：

1. 初始化acl资源：此样例中，包括指定计算设备、创建context、创建stream等操作。再初始化了两个模型类（CTPN、SVTR），初始化时，进行了模型的加载以及输出数据集的创建

2. 推理：读入图片，调用model.infer进行推理，其中包含数据的前处理、输入数据集结构的创建、推理、将推理结果转换为numpy、并进行后处理等操作，得到最终文本识别后的图片，保存在infer_result文件夹中。

3. 资源销毁：最后记得释放相关资源，包括卸载模型、销毁输入输出数据集、释放Context、释放指定的计算设备、以及AscendCL去初始化等操作。

AscendCL接口分类总结：

分类	接口函数	描述
AscendCL初始化相关	acl.init()	pyACL初始化函数
	acl.rt.set_device(device_id)	指定当前进程或线程中用于运算的Device，同时隐式创建默认Context
	acl.rt.create_context(device_id)	在当前进程或线程中显式创建一个Context
模型描述信息相关	acl.mdl.load_from_file(model_path)	从文件加载离线模型数据（适配昇腾AI处理器的离线模型）
	acl.mdl.create_desc()	创建aclmdlDesc类型的数据
	acl.mdl.get_desc(model_desc, model_id)	根据模型ID获取该模型的aclmdlDesc类型数据
	acl.mdl.get_num_inputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输入个数
	acl.mdl.get_num_outputs(model_desc)	根据aclmdlDesc类型的数据，获取模型的输出个数
	acl.mdl.get_output_size_by_index(model_desc, i)	根据aclmdlDesc类型的数据，获取指定输出的大小，单位为Byte
	acl.mdl.get_output_dims(model_desc, i)	根据模型描述信息获取指定的模型输出tensor的维度信息
	acl.mdl.get_output_data_type(model_desc, i)	根据模型描述信息获取模型中指定输出的数据类型
数据集结构相关	acl.mdl.create_dataset()	创建aclmdlDataset类型的数据
	acl.create_data_buffer(data_addr, size)	创建aclDataBuffer类型的数据，该数据类型用于描述内存地址、大小等内存信息
	acl.mdl.add_dataset_buffer(dataset, date_buffer)	向aclmdlDataset中增加aclDataBuffer

分类	接口函数	描述
	acl.mdl.get_dataset_num_buffers(dataset)	获取aclmdlDataset中aclDataBuffer的个数
	acl.mdl.get_dataset_buffer(dataset, i)	获取aclmdlDataset中的第i个aclDataBuffer
	acl.get_data_buffer_addr(buffer)	获取aclDataBuffer类型中的数据的地址对象
	acl.get_data_buffer_size(buffer)	获取aclDataBuffer类型中数据的内存大小，单位Byte
	acl.util.bytes_to_ptr(bytes_data)	将bytes对象转换成为void*数据，可以将转换好的数据传递给C函数直接使用
	acl.util.ptr_to_bytes(ptr, size)	将void*数据转换为bytes对象，可以使python代码直接访问
	acl.rt.malloc(size, policy)	申请Device上的内存
推理相关	acl.mdl.execute(model_id, input_dataset, output_dataset)	执行模型推理，直到返回推理结果
销毁资源相关	acl.destroy_data_buffer(data_buffer)	销毁aclDataBuffer类型的数据
	acl.mdl.destroy_dataset(dataset)	销毁aclmdlDataset类型的数据
	acl.mdl.unload(model_id)	系统完成模型推理后，可调用该接口卸载模型，释放资源
	acl.mdl.destroy_desc(model_desc)	销毁aclmdlDesc类型的数据
	acl.rt.destroy_context(context)	销毁一个Context，释放Context的资源
	acl.rt.reset_device(device_id)	复位当前运算的Device，释放Device上的资源，包括默认Context、默认Stream以及默认Context下创建的所有Stream
	acl.finalize()	pyACL去初始化函数，用于释放进程内的pyACL相关资源

理解各个接口含义后，用户可进行灵活运用。除此外，此样例中只示范了图片推理，若需要对视频流数据进行推理，可用三种方式输入视频流数据：USB摄像头、手机摄像头。具体使用方式可参考《**摄像头拉流**》，用户只需将前处理、推理及后处理代码放入摄像头推理代码的循环中即可，注意有些细节地方需进行相应修改，具体逻辑可参照图像分类应用中的样例总结与扩展。

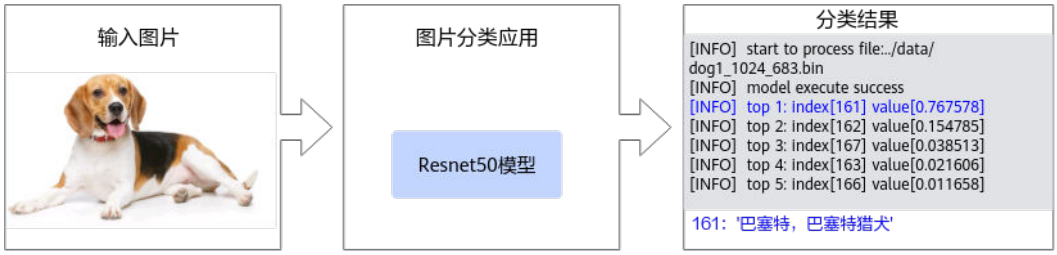
5 图像分类应用样例开发介绍（C++）

在本章节介绍基于AscendCL接口如何开发一个基于ResNet-50模型的图像分类样例。

样例介绍

“图片分类应用”，即标识图片所属的分类。

图 5-1 图片分类应用



本例中使用的是Caffe框架的ResNet-50模型。用户可以直接使用训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以基于算法、框架构建适合的模型。

ResNet-50模型的基本介绍如下：

- 输入数据：RGB格式、224*224分辨率的输入图片。
- 输出数据：图片的类别标签及其对应置信度。

说明

- 置信度是指图片所属某个类别可能性。
- 类别标签和类别的对应关系与训练模型时使用的数据集有关，需要查阅对应数据集的标签及类别的对应关系。

获取代码

步骤1 获取代码文件。

单击**获取链接**或使用wget命令，下载代码文件压缩包，以**root**用户登录开发者套件。

```
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/Atlas%20200I%20DK%20A2/DevKit/models/sdk_cal_samples/resnet50_acl_cplusplus.zip
```

步骤2 将“resnet50_acl_cplusplus.zip”压缩包上传到开发者套件，解压并进入解压后的目录。

```
unzip resnet50_acl_cplusplus.zip
cd resnet50_acl_cplusplus
```

代码目录结构如下所示，按照正常开发流程，需要将框架模型文件转换成昇腾AI处理器支持推理的om格式模型文件，鉴于当前是入门内容，用户可直接获取已转换好的om模型进行推理。

```
cd resnet50_acl_cplusplus
├── data
│   ├── dog1_1024_683.jpg           // 测试图片
├── model
│   ├── resnet50.om            // om模型文件
├── script
│   ├── transferPic.py              // 将测试图片预处理为符合模型要求的图片，包括将*.jpg转换为*.bin，同时将图片从1024*683的分辨率缩放为224*224
├── src
│   ├── CMakeLists.txt              // cmake编译脚本
│   ├── main.cpp                   // 主函数，图片分类功能的实现文件
├── sample_build.sh                 // 编译代码的脚本
├── sample_run.sh                   // 运行应用的脚本
```

----结束

代码解析

开发代码过程中，在“resnet50_acl_cplusplus/src/main.cpp”文件中已包含读入数据、前处理、推理、后处理等功能，串联整个应用代码逻辑，此处仅对代码进行解析。

1. 在“main.cpp”文件的开头有如下代码，用于导入第三方库与调用AscendCL接口推理所需文件。

   ```
   #include "acl/acl.h"
   #include <iostream>
   #include <fstream>
   #include <cstring>
   #include <map>

   using namespace std;
   ```

2. 资源初始化。

   使用AscendCL接口开发应用时，必须先初始化AscendCL ，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。

   在初始化时，还支持跟推理相关的可配置项（例如，性能相关的采集信息配置），以json格式的配置文件传入AscendCL初始化接口。如果当前的默认配置已满足需求（例如，默认不开启性能相关的采集信息配置），无需修改，可向AscendCL初始化接口中传入nullptr。

   ```
   int32_t deviceId = 0;//指定运行模型的昇腾AI处理器ID，0表示指定0号处理器
   void InitResource()
   {
       aclError ret = aclInit(nullptr);//调用AscendCL初始化接口
       ret = aclrtSetDevice(deviceId); //调用指定运行模型的昇腾AI处理器ID的接口
   }
   ```

3. 模型加载。

   ```
   uint32_t modelId;
   void LoadModel(const char* modelPath)
   {
       aclError ret = aclmdlLoadFromFile(modelPath, &modelId);//调用加载模型文件接口，modelId为模型ID的指针，通过接口加载模型后，会为模型ID赋值
   }
   ```


4. 将测试图片读入内存，供推理使用。

```
size_t pictureDataSize = 0;
void *pictureData;

//申请内存，使用C/C++标准库的函数将测试图片读入内存
void LoadPicture(const char* picturePath)
{
    string fileName = picturePath;
    ifstream binFile(fileName, ifstream::binary);
    binFile.seekg(0, binFile.end);
    pictureDataSize = binFile.tellg();
    binFile.seekg(0, binFile.beg);
    aclError ret = aclrtMallocHost(&pictureData, pictureDataSize);
    binFile.read((char*)pictureData, pictureDataSize);
    binFile.close();
}
```

5. 执行推理。

   在调用AscendCL接口进行模型推理时，模型推理有输入、输出数据，输入、输出数据需要按照AscendCL规定的数据类型存放。相关数据类型如下：

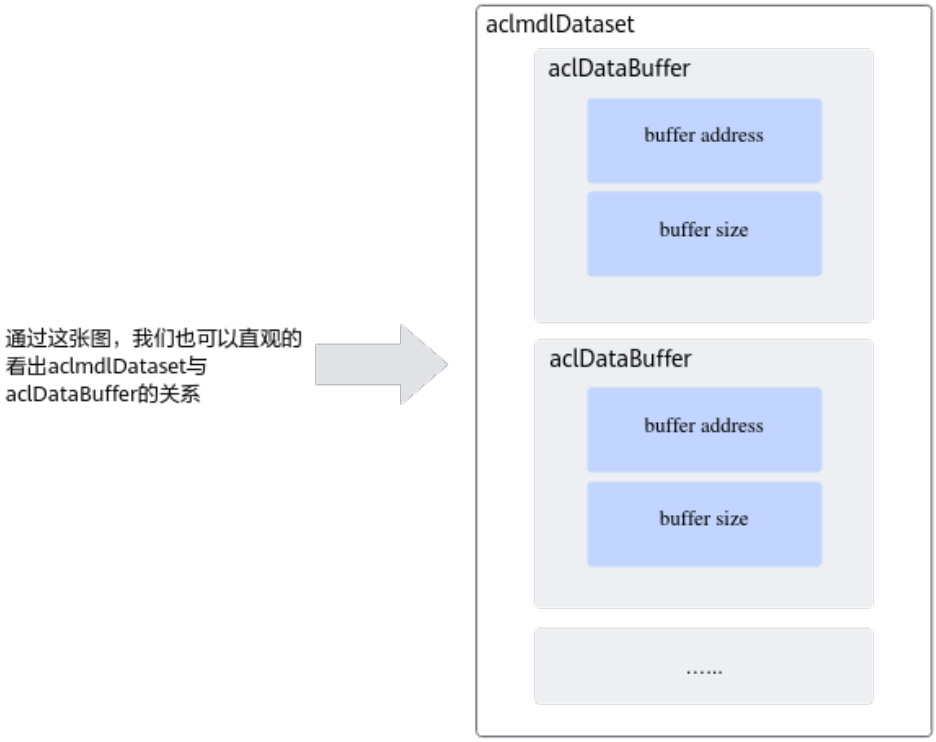
   - 使用aclmdlDesc类型的数据描述模型基本信息（例如输入/输出的个数、名称、数据类型、Format、维度信息等）。

     模型加载成功后，用户可根据模型的ID，调用该数据类型下的操作接口获取该模型的描述信息，进而从模型的描述信息中获取模型输入/输出的个数、内存大小、维度信息、Format、数据类型等信息。
   - 使用aclDataBuffer类型的数据来描述每个输入/输出的内存地址、内存大小。

     调用aclDataBuffer类型下的操作接口获取内存地址、内存大小等，便于向内存中存放输入数据、获取输出数据。
   - 使用aclmdlDataset类型的数据描述模型的输入/输出数据。

     模型可能存在多个输入、多个输出，调用aclmdlDataset类型的操作接口添加多个aclDataBuffer类型的数据。

     图 5-2 aclmdlDataset 与 aclDataBuffer 的关系




```
aclmdlDataset *inputDataSet;
aclDataBuffer *inputDataBuffer;
aclmdlDataset *outputDataSet;
aclDataBuffer *outputDataBuffer;
aclmdlDesc *modelDesc;
size_t outputDataSize = 0;
void *outputData;

// 准备模型推理的输入数据结构
void CreateModelInput()
{
    // 创建aclmdlDataset类型的数据，描述模型推理的输入
    inputDataSet = aclmdlCreateDataset();
    inputDataBuffer = aclCreateDataBuffer(pictureData, pictureDataSize);
    aclError ret = aclmdlAddDatasetBuffer(inputDataSet, inputDataBuffer);
}

// 准备模型推理的输出数据结构
void CreateModelOutput()
{
    // 创建模型描述信息
    modelDesc =  aclmdlCreateDesc();
    aclError ret = aclmdlGetDesc(modelDesc, modelId);
    // 创建aclmdlDataset类型的数据，描述模型推理的输出
    outputDataSet = aclmdlCreateDataset();
    // 获取模型输出数据需占用的内存大小，单位为Byte
    outputDataSize = aclmdlGetOutputSizeByIndex(modelDesc, 0);
    // 申请输出内存
    ret = aclrtMalloc(&outputData, outputDataSize, ACL_MEM_MALLOC_HUGE_FIRST);
    outputDataBuffer = aclCreateDataBuffer(outputData, outputDataSize);
    ret = aclmdlAddDatasetBuffer(outputDataSet, outputDataBuffer);
}

// 执行模型
void Inference()
{
    CreateModelInput();
    CreateModelOutput();

    float start;
    float finish;
    aclError ret = aclmdlExecute(modelId, inputDataSet, outputDataSet);
}
```

6. 处理模型推理的结果数据，在屏幕上显示图片的top5置信度的类别编号。

```
void PrintResult()
{
    // 将内存中的数据转换为float类型
    float* outFloatData = reinterpret_cast<float *>(outputData);

    // 屏显测试图片的top5置信度的类别编号
    map<float, unsigned int, greater<float>> resultMap;
    for (unsigned int j = 0; j < outputDataSize / sizeof(float);++j)
    {
        resultMap[*outFloatData] = j;
        outFloatData++;
    }

    int cnt = 0;
    for (auto it = resultMap.begin();it != resultMap.end();++it)
    {
        if(++cnt > 5)
        {
            break;
        }
        printf("top %d: index[%d] value[%lf] \n", cnt, it->second, it->first);
    }
}
```

7. 卸载模型，并释放模型描述信息。

推理结束，需及时释放模型描述信息、卸载模型。

```
void UnloadModel()
{
    // 释放模型描述信息
    aclmdlDestroyDesc(modelDesc);
    // 卸载模型
    aclmdlUnload(modelId);
}
```

8. 释放内存、销毁推理相关的数据类型。

```
void UnloadPicture()
{
    aclError ret = aclrtFreeHost(pictureData);
    pictureData = nullptr;
    aclDestroyDataBuffer(inputDataBuffer);
    inputDataBuffer = nullptr;
    aclmdlDestroyDataset(inputDataSet);
    inputDataSet = nullptr;

    ret = aclrtFree(outputData);
    outputData = nullptr;
    aclDestroyDataBuffer(outputDataBuffer);
    outputDataBuffer = nullptr;
    aclmdlDestroyDataset(outputDataSet);
    outputDataSet = nullptr;
}
```

9. 释放资源。

在确定完成了AscendCL的所有调用之后，或者进程退出之前，需调用如下接口实现计算设备释放，AscendCL去初始化。

```
void DestroyResource()
{
    aclError ret = aclrtResetDevice(deviceId);
    aclFinalize();
}
```

10. 定义一个运行main函数。

```
int main()
{
    // 1.定义一个资源初始化的函数，用于AscendCL初始化、运行管理资源申请（指定计算设备）
    InitResource();

    // 2.定义一个模型加载的函数，加载图片分类的模型，用于后续推理使用
    const char *modelPath = "../model/resnet50.om"; //定义模型文件路径，om模型文件路径请参见获取代码。
    LoadModel(modelPath);//定义模型加载函数

    // 3.定义一个读图片数据的函数，将测试图片数据读入内存，并传输到Device侧，用于后续推理使用
    const char *picturePath = "../data/dog1_1024_683.bin";
    LoadPicture(picturePath);

    // 4.定义一个推理的函数，用于执行推理
    Inference();

    // 5.定义一个推理结果数据处理的函数，用于在终端上屏显测试图片的top5置信度的类别编号
    PrintResult();

    // 6.定义一个模型卸载的函数，卸载图片分类的模型
    UnloadModel();

    // 7.定义一个函数，用于释放内存、销毁推理相关的数据类型，防止内存泄露
    UnloadPicture();

    // 8.定义一个资源去初始化的函数，用于AscendCL去初始化、运行管理资源释放（释放计算设备）
    DestroyResource();
}
```


运行推理

步骤1 编译代码。

进入“resnet50_acl_cplusplus”目录下，执行以下命令。

下为设置环境变量的示例，<SAMPLE_DIR>表示样例所在的目录。

```
export APP_SOURCE_PATH=<SAMPLE_DIR>/resnet50_acl_cplusplus
export DDK_PATH=/usr/local/Ascend/ascend-toolkit/latest
export NPU_HOST_LIB=/usr/local/Ascend/ascend-toolkit/latest/aarch64-linux/devlib
chmod +x sample_build.sh
./sample_build.sh
```

说明

- 如果复制多行命令到MobaXterm执行，可能会出现是否确认执行的提示，单击"OK"即可；
- 如果执行脚本报错“ModuleNotFoundError: No module named 'PIL'”，则表示缺少Pillow库，请使用**pip3 install Pillow --user**命令安装Pillow库。

步骤2 执行以下命令。

```
chmod +x sample_run.sh
./sample_run.sh
```

终端上屏显的结果如下，index表示类别标识、value表示该分类的最大置信度。

```
top 1: index[161] value[0.764648]
top 2: index[162] value[0.156616]
top 3: index[167] value[0.038971]
top 4: index[163] value[0.021698]
top 5: index[166] value[0.011887]
```

说明

类别标签和类别的对应关系与训练模型时使用的数据集有关，本样例使用的模型是基于imagenet数据集进行训练的，用户可以在互联网上查阅对应数据集的标签及类别的对应关系。

当前屏显信息中的类别标识与类别的对应关系如下：

"161": ["basset", "basset hound"]

"162": ["beagle"]

"163": ["bloodhound", "sleuthhound"]

"166": ["Walker hound", "Walker foxhound"]

"167": ["English foxhound"]

----结束

6 更多代码样例

更多的开发者套件代码样例请访问**昇腾社区应用样例**。