
实验六 语义分割

RT-AK 教育套件实验手册

上海睿赛德电子科技有限公司 版权所有 @2021



WWW.RT-THREAD.ORG

Monday 22nd November, 2021

目录

目录	i
1 实验介绍与目的	1
1.1 实验介绍	1
1.2 实验目的	3
2 实验器材	4
3 实验步骤	5
3.1 基础概念及原理	5
3.1.1 基础概念及原理	5
3.1.2 语义分割标签图片格式说明	6
3.2 AI 模型训练	7
3.3 模型部署	15
3.4 嵌入式 AI 模型应用	17
3.4.1 代码流程	17
3.4.2 核心代码说明	17
4 编译烧录	20
4.1 编译	20
4.2 烧录	20
5 实验现象	21
6 附录	23
6.1 嵌入式 AI 开发 API 文档	23
6.2 LCD API 说明手册	24
6.3 Camera 使用	26

查找设备	26
初始化设备	26
打开和关闭设备	27
读写设备	28
数据收发回调	28

第 1 章

实验介绍与目的

1.1 实验介绍

本实验是基于 Tensorflow 训练的一个语义分割模型。语义分割模型的主要功能是：输入任意一张物体图片，对图片中物体进行像素级分类，从而将图片中的物体轮廓进行识别并分割出来。

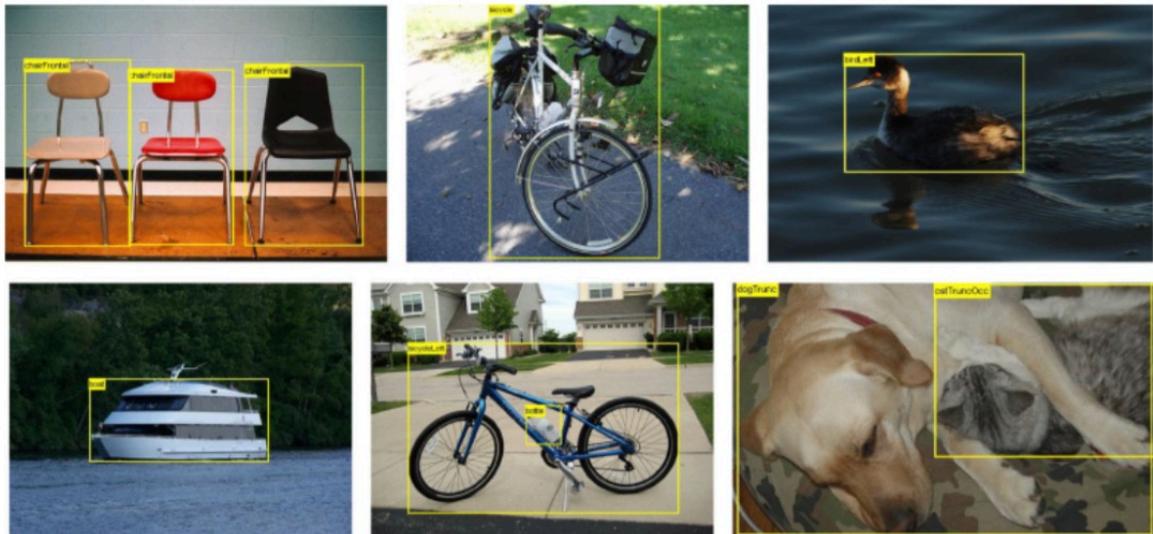
本实验模型和数据集较大，对模训练所使用的电脑有较高的性能要求（GTX1080Ti 下 200epochs 约需要 10 小时），模型部署环节建议直接使用预训练模型，提供模型搭建及训练代码供学习使用，若条件允许可尝试完整自训练过程。

数据集使用的是 **VOC2012** 图像增强版数据集，其中一共有 17125 张图片，观察 train.txt 和 val.txt 文本内容中的图片标号索引行数可知，其中有 8498 张训练图片和 1449 张测试图片，图片尺寸大小不一，但相差不大。

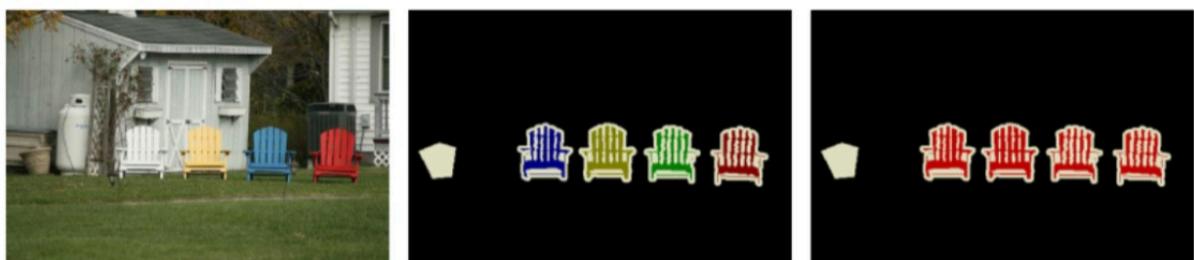
VOC2012 数据集出自 PASCAL VOC 挑战赛，PASCAL VOC 挑战赛 (The PASCAL Visual Object Classes) 是一个世界级的计算机视觉挑战赛，PASCAL 全称：Pattern Analysis, Statistical Modeling and Computational Learning，是一个由欧盟资助的网络组织。

很多优秀的计算机视觉模型比如分类，定位，检测，分割，动作识别等模型都是基于 PASCAL VOC 挑战赛及其数据集上推出的，尤其是一些目标检测模型（比如大名鼎鼎的 RCNN 系列，以及后面的 YOLO，SSD 等）。

VOC2012 的图片样例如下图所示：



(a) Classification and detection



(b) Segmentation



(c) Action classification



(d) Person layout

图 1.1: cifar10 数据

1.2 实验目的

1. 了解分割基本概念与原理
2. 掌握使用 Tensorflow 框架训练语义分割模型模型
3. 掌握使用 RT-AK 一行命令进行模型转换和部署 AI 模型
4. LCD 和摄像头的基本使用
5. 完成嵌入式 AI 开发：输入一张物体图片，完成对图片的语义分割并进行展示

第 2 章

实验器材

1. PC
2. EgdeAI 实验板
3. Camera 使用的为 GC0308
4. LCD 驱动 IC 为 ILI9341

第 3 章

实验步骤

注意：请先确保环境安装没有问题，环境安装可以参考实验二，实验中若提示确实相关模块，直接使用命令进行安装相应模块即可。

本实验使用 **GPU** 训练速度会快很多，但使用 **GPU** 训练不仅需要安装 **tensorflow-gpu** 版本，还需要安装对应的 **CUDA** 和 **cuDNN**。有 **Nvidia** 显卡的同学可以参考安装细节 <https://tensorflow.google.cn/install/gpu?hl=zh-cn>。

3.1 基础概念及原理

3.1.1 基础概念及原理

语义分割是对图片进行像素级别的分类，即对每个像素点进行分类。语义分割模型的输出结果为 (H, W, C) 形状，其中最后一维通道维与分类数相等，在本实验中为 21（20 类 + 背景）。每个通道代表一个类别的预测概率，在训练时与我们处理标准分类值的方法相似，我们通过 **one_hot** 编码类别标签的方法创建目标，本质上讲是要为每一个可能的类创建一个输出通道，即沿着通道维根据真实类别做 **one_hot** 编码，如下图所示意：

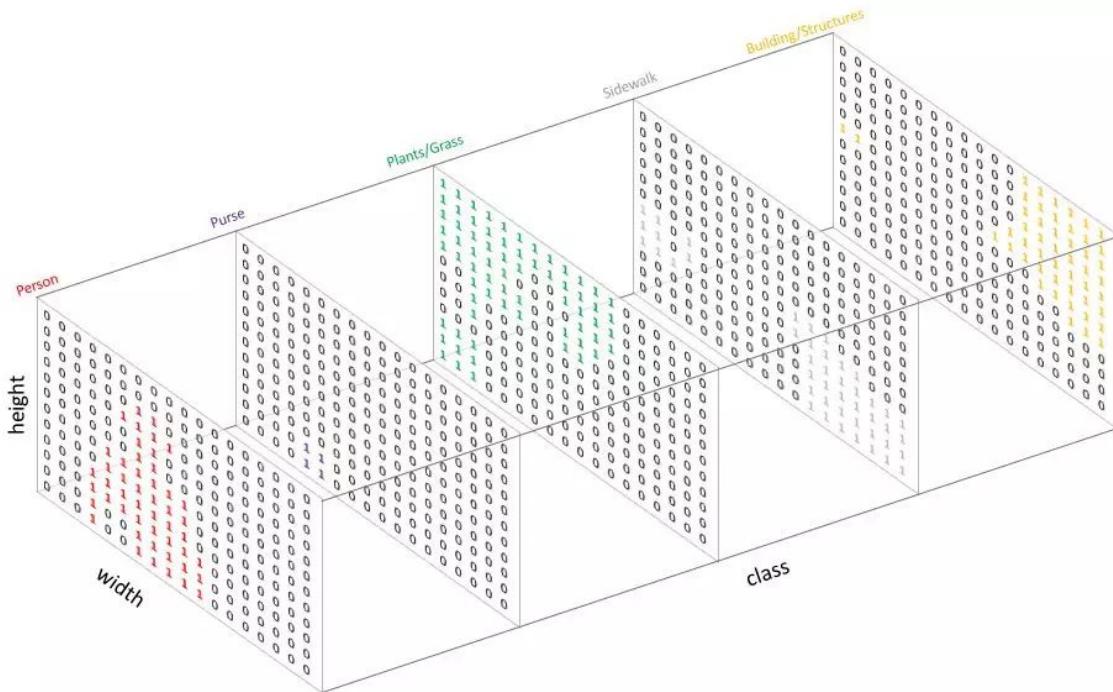


图 3.1: one_hot

模型的输出也与此类似，然后我们可以利用每一个像素位深向量的 `argmax` 函数将预测值分解为分割映射得到最终图像中像素分类结果，如下图：

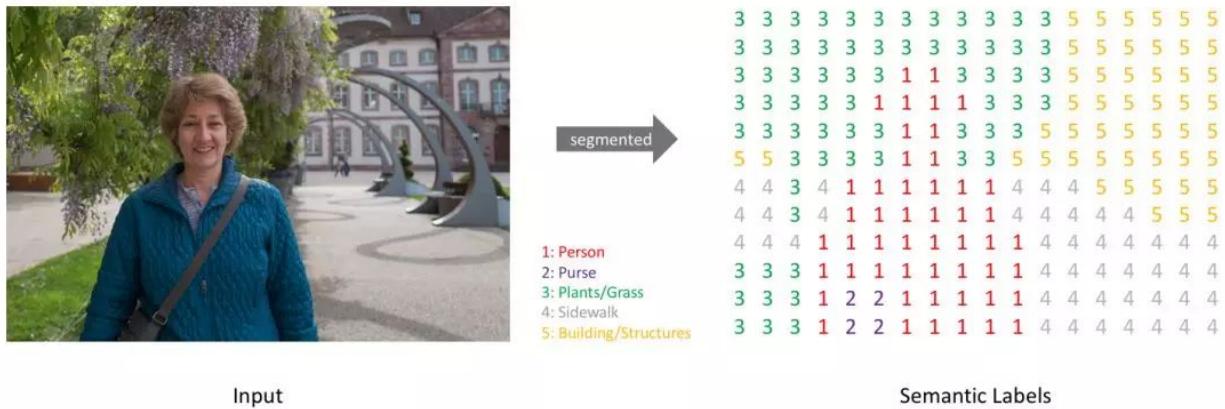


图 3.2: result

3.1.2 语义分割标签图片格式说明

语义分割数据集标签 png 格式图片，图片模式为调色板模式。图片中预先保存了 256 种 RGB 颜色，而图片每个实际像素值为 [0-255] 的索引，在显示时将其与 256 中颜色进行对应显示。而语义分割标签也即使用 [0-20] 像素值作为类别索引。

PIL 库中图片模式 “P” 为调色板模式，为 8 位彩色图像，它的每个像素用 8 个 bit 表示，其对应的彩色值是按照调色板查询出来的。

标签索引及其对应 RGB 颜色如下：

B-ground	Aero plane	Bicycle	Bird	Boat	Bottle	Bus
Car	Cat	Chair	Cow	Dining-Table	Dog	Horse
Motorbike	Person	Potted-Plant	Sheep	Sofa	Train	TV/Monitor

图 3.3: color_map

```
#RGB color map
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
[0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
[64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
[64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
[0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],[0, 64, 128]]


VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
'diningtable', 'dog', 'horse', 'motorbike', 'person',
'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

想要了解标签图片调色板信息可参考下面代码:

```
from PIL import Image
label_dir = 'imagepath' #需要修改,指定一张语义分割标签图片
label_suffix='.png' #图片后缀名
label = Image.open('%s/%s%s' % (label_dir, '2007_000032', label_suffix))
print('标签图片模式:',label.mode,'\n调色板信息:\n',np.array(label.getpalette()).reshape((256,3)))
```

语义分割模型通常都很大，训练难度也比较高。根据训练好坏，会有不同的预测准确率和表现效果。

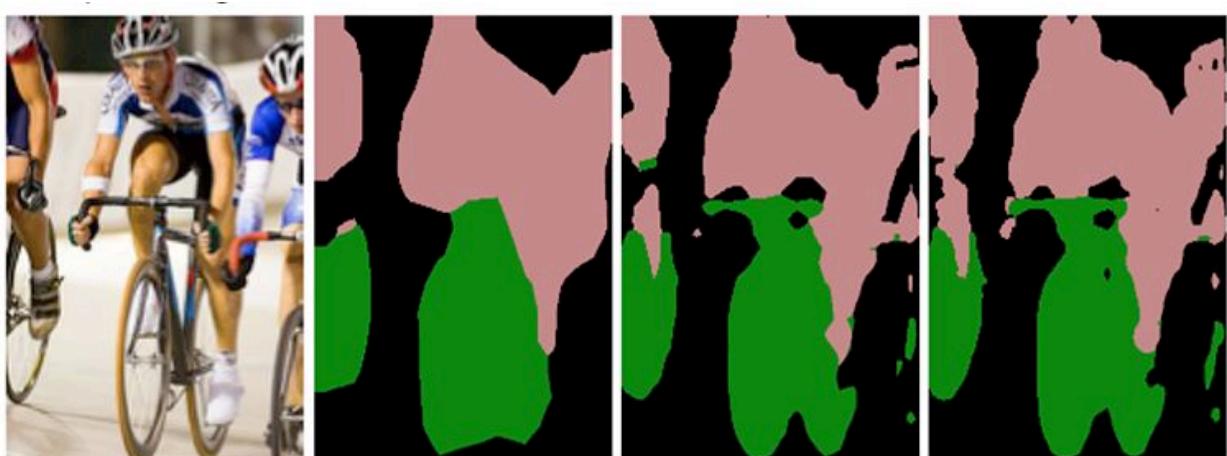


图 3.4: example

3.2 AI 模型训练

1. 实验文件夹说明

如下图所示，实验文件夹中包含模型完整训练代码，其中：

`Train_Segmentation.ipynb` 为模型训练代码，使用 `jupyter notebook` 可打开查看和运行。

`Inference_Segmentation.ipynb` 为模型推理验证代码，可以在 `jupyter notebook` 中打开，读取图片验证模型预测效果。

`tiny_mobile_seg.py` 为模型搭建代码

`segutils` 文件夹总包含训练时所需要的一些数据集处理方法代码，可自行了解学习。

		Name ↓	Last Modified	File size
□ 0	□ /			
□	□ Applications		1小时前	
□	□ Datasets		1小时前	
□	□ Firmwares		1小时前	
□	□ Images		1天前	
□	□ Models		5小时前	
□	□ segutils	数据集处理&增强	1个月前	
□	□ Utils		1小时前	
□	□ 课后拓展		39分钟前	
□	■ Inference_Segmentation.ipynb	推理验证	5小时前	128 kB
□	■ tiny_mobile_seg.py	模型搭建	1个月前	6.61 kB
□	■ Train_Segmentation.ipynb	训练代码	运行 31分钟前	55.7 kB
□	□ requirements.txt		3天前	16 B

图 3.5: files

2. 导入库

导入所需要的 `python` 库，点击 Run 运行按钮，运行代码

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
# from tensorflow.keras.preprocessing import image
import numpy as np
import csv
import json
import os
import matplotlib.pyplot as plt
import tiny_mobile_seg
from tensorflow.keras.optimizers import SGD, Adam, Nadam
from tensorflow.keras.callbacks import *
# from tensorflow.keras.objectives import *
from tensorflow.keras.metrics import binary_accuracy
from tensorflow.keras.models import load_model
import tensorflow.keras.backend as K
from segutils.loss_function import *
from segutils.metrics import *
from segutils.SegDataGenerator import *
import time

# 使用标号为“0”的 GPU 训练，若无 GPU，该行代码会产生一些 warning
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

因为官方数据集较大，这里我们使用 GPU 训练来作讲解，同样的代码也可以使用 CPU 训练，只是速度较慢。

3. 训练参数解释

在此先对训练时的一些参数设置给出解释，可结合后续代码理解其作用。在此特别提醒在实验训练的电脑中，注意设置数据集的相关路径。

```
model_name = 'seg_model.h5' #keras模型名,注意.h5后缀
batch_size = 16 #训练时batch_size
epochs = 200 #训练次数
lr_base = 0.1 # 学习率初始值
resume_training = False #恢复上一次训练, 需在train()中指定checkpoint文件
target_size = (240, 320) #图片尺寸(W,H)
dataset = 'VOC2012' # 数据集
loss_fn = softmax_sparse_crossentropy_ignoring_last_label #损失函数
metrics = [sparse_accuracy_ignoring_last_label] #准确率评估函数
ignore_label = 255 #语义分割标签图片中轮廓部分,要对其进行忽略
label_cval = 255 #数据增强时,图片边界填充值
class_weight = None #训练时对不同样本计算loss时所占权重.
train_file_path = '/home/liqiwen/rtt_nas/liqiwen/VOCdatasets/VOCdevkit/VOC2012/
    train.txt' #训练集文件名索引
val_file_path = '/home/liqiwen/rtt_nas/liqiwen/VOCdatasets/VOCdevkit/VOC2012/
    ImageSets/Segmentation/val.txt' #验证集文件名索引
data_dir = '/home/liqiwen/rtt_nas/liqiwen/VOCdatasets/VOCdevkit/VOC2012/
    JPEGImages' #图片总数据集
label_dir = '/home/liqiwen/rtt_nas/liqiwen/VOCdatasets/VOCdevkit/VOC2012/
    cls_aug' #增强版语义分割标签
classes = 21 #语义分割中类别数,20类+1背景
data_suffix='.jpg' #图片后缀名
label_suffix='.png' #标签后缀名
```

4. 数据集的加载

数据集请参考 Datasets 获取文件夹下说明。数据集目录讲解:

名称	修改日期	类型	大小
Annotations	2012/5/11 23:03	文件夹	
cls_aug	2021/5/14 18:33	文件夹	
ImageSets	2012/5/11 22:55	文件夹	
JPEGImages	2021/4/9 14:45	文件夹	
SegmentationClass	2021/5/11 18:23	文件夹	
SegmentationObject	2012/5/11 23:03	文件夹	
train.txt	2021/5/14 18:07	文本文档	100 KB

图 3.6: Datasets

VOC2012

```
-- Annotations 进行detection 任务时的 标签文件,xml文件形式
-- ImageSets 存放数据集的分割文件索引, 比如train.txt,val.txt,test.txt
-- JPEGImages 存放 .jpg格式的图片文件
```

```
|-- SegmentationClass 存放 按照 class 分割的图片  
|-- SegmentationObject 存放 按照 object 分割的图片  
|-- cls_aug 存放 SegmentationClass 分割图片的增强版，训练数据集请使用此目录  
\-- train.txt 训练数据及文件名索引，对应的是cls_aug目下的文件索引，训练时请使用此  
    文件
```

下面为数据集的加载，以及设置在训练时对数据集进行随机变换，以达到扩充训练集多样性的多种。关于数据集生成及处理代码，可在 `segutils` 文件夹中查看。

```
#创建训练数据集生成器，配置数据集增强参数，对图片进行随机缩放，旋转，平移，裁剪等。  
train_datagen = SegDataGenerator(zoom_range=[0.5, 1.0], #缩放系数  
                                  zoom_maintain_shape=True, #维持  
                                  crop_mode='none', #图片裁剪  
                                  crop_size=target_size, #裁剪尺寸  
# pad_size=(505, 505),  
                                  rotation_range=0.,#旋转系数  
                                  shear_range=0, #剪切强度（以弧度逆时针方向  
                                  剪切角度）  
                                  horizontal_flip=True, #水平翻转  
                                  channel_shift_range=20., #颜色通道平移  
                                  fill_mode='constant', #填充图片边界超出部分  
                                  填充常数  
                                  label_cval=label_cval) #填充值  
  
#创建验证数据集生成器，无需数据集增强  
val_datagen = SegDataGenerator()  
#传入数据集路径  
x_y_train = train_datagen.flow_from_directory(  
                                              file_path=train_file_path,  
                                              data_dir=data_dir,  
                                              data_suffix=data_suffix,  
                                              label_dir=label_dir,  
                                              label_suffix=label_suffix,  
                                              classes=classes,  
                                              target_size=target_size,  
                                              color_mode='rgb',  
                                              batch_size=batch_size,  
                                              shuffle=True,  
                                              ignore_label=ignore_label)  
  
#传入数据集路径  
x_y_validation=val_datagen.flow_from_directory(  
                                              file_path=val_file_path,  
                                              data_dir=data_dir,  
                                              data_suffix='.jpg',  
                                              label_dir=label_dir,  
                                              label_suffix='.png',  
                                              classes=classes,  
                                              target_size=target_size,  
                                              color_mode='rgb',  
                                              batch_size=batch_size,
```

```
shuffle=False)
```

对原始数据集进行图像增强，这样可以增强模型的泛化能力。

5. 搭建神经网络模型

我们基于深度可分离卷积算子，结构使用残差结构作为骨干特征提取网络搭建整个网络模型。网络搭建代码较为庞大，在此不做详细解释，具体可参考代码文件：`tiny_mobile_seg.py`

注：API 参数具体含义参考 https://tensorflow.google.cn/api_docs/python/tf?hl=zh-cn

我们可以使用神经网络可视化工具 <https://www.electronjs.org/apps/netron> 来查看模型的网络结构，下图（a）中所示为骨干网络基本结构，使用了残差结构。（b）为模型最终输出部分，输出为形状为（1,30,40,21）：

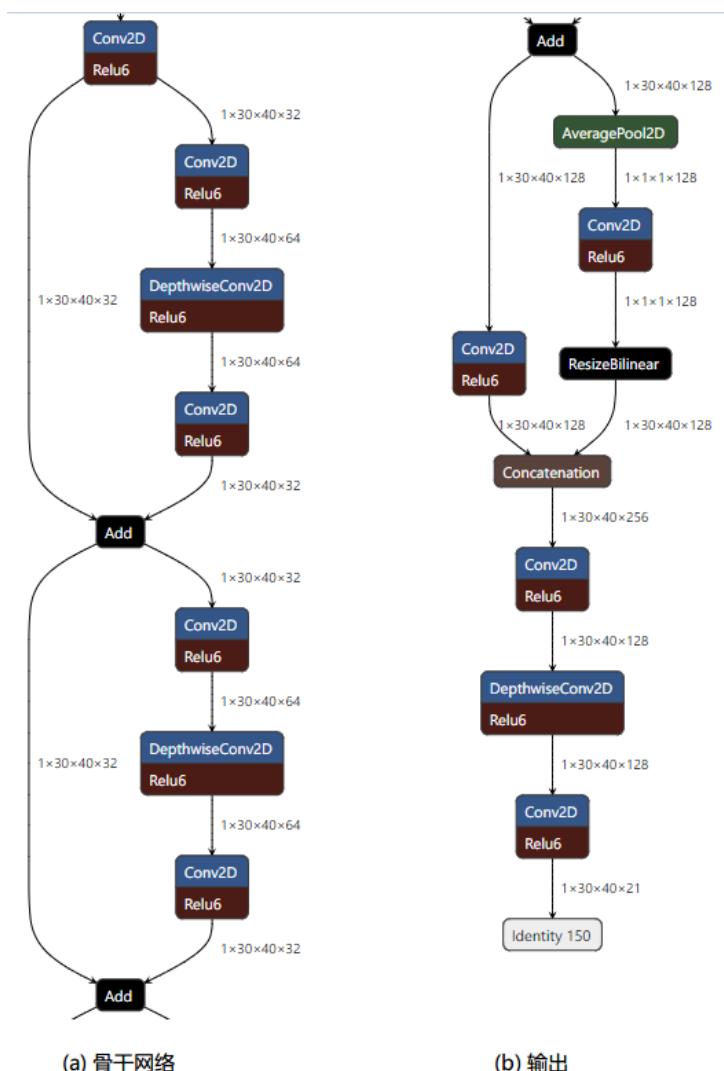


图 3.7：模型网络结构

使用 API 构建模型之后，我们可以使用 `model.summary()` 方法也可以查看模型网络结构以及各层输出尺寸、参数数量等信息：

```
model.summary()
#out[]:
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[None, 240, 320, 3]	0	
conv1_2 (Conv2D)	(None, 120, 160, 8)	216	input_1[0][0]
conv1_bn_2 (BatchNormalization)	(None, 120, 160, 8)	32	conv1_2[0][0]
conv1_relu_2 (ReLU)	(None, 120, 160, 8)	0	conv1_bn_2[0][0]
conv_dw_3 (DepthwiseConv2D)	(None, 120, 160, 8)	72	conv1_relu_2[0][0]
conv_dw_3_bn (BatchNormalizatio)	(None, 120, 160, 8)	32	conv_dw_3[0][0]
conv_dw_3_relu (ReLU)	(None, 120, 160, 8)	0	conv_dw_3_bn[0][0]
conv_pw_3 (Conv2D)	(None, 120, 160, 8)	64	conv_dw_3_relu[0][0]
conv_pw_3_bn (BatchNormalizatio)	(None, 120, 160, 8)	32	conv_pw_3[0][0]
conv_pw_3_relu (ReLU)	(None, 120, 160, 8)	0	conv_pw_3_bn[0][0]
conv_dw_4 (DepthwiseConv2D)	(None, 120, 160, 8)	72	conv_pw_3_relu[0][0]
conv_dw_4_bn (BatchNormalizatio)	(None, 120, 160, 8)	32	conv_dw_4[0][0]
conv_dw_4_relu (ReLU)	(None, 120, 160, 8)	0	conv_dw_4_bn[0][0]
conv_pw_4 (Conv2D)	(None, 120, 160, 8)	64	conv_dw_4_relu[0][0]
conv_pw_4_bn (BatchNormalizatio)	(None, 120, 160, 8)	32	conv_pw_4[0][0]
conv_pw_4_relu (ReLU)	(None, 120, 160, 8)	0	conv_pw_4_bn[0][0]
add (Add)	(None, 120, 160, 8)	0	conv_pw_4_relu[0][0] conv_pw_3_relu[0][0]
conv_pad_5 (ZeroPadding2D)	(None, 121, 161, 8)	0	add[0][0]
•			
•			
•			
average_pooling2d (AveragePooli	(None, 1, 1, 128)	0	add_10[0][0]
conv1_26 (Conv2D)	(None, 1, 1, 128)	16384	average_pooling2d[0][0]
conv1_27 (Conv2D)	(None, 30, 40, 128)	16384	add_10[0][0]
conv1_bn_26 (BatchNormalizatio)	(None, 1, 1, 128)	512	conv1_26[0][0]
conv1_bn_27 (BatchNormalizatio)	(None, 30, 40, 128)	512	conv1_27[0][0]
conv1_relu_26 (ReLU)	(None, 1, 1, 128)	0	conv1_bn_26[0][0]
conv1_relu_27 (ReLU)	(None, 30, 40, 128)	0	conv1_bn_27[0][0]
tf.image.resize (TFOpLambda)	(None, 30, 40, 128)	0	conv1_relu_26[0][0]
concatenate (Concatenate)	(None, 30, 40, 256)	0	conv1_relu_27[0][0] tf.image.resize[0][0]
conv1_28 (Conv2D)	(None, 30, 40, 128)	32768	concatenate[0][0]
conv1_bn_28 (BatchNormalizatio)	(None, 30, 40, 128)	512	conv1_28[0][0]
conv1_relu_28 (ReLU)	(None, 30, 40, 128)	0	conv1_bn_28[0][0]
conv_dw_29 (DepthwiseConv2D)	(None, 30, 40, 128)	1152	conv1_relu_28[0][0]
conv_dw_29_bn (BatchNormalizati)	(None, 30, 40, 128)	512	conv_dw_29[0][0]
conv_dw_29_relu (ReLU)	(None, 30, 40, 128)	0	conv_dw_29_bn[0][0]
conv_pw_29 (Conv2D)	(None, 30, 40, 21)	2688	conv_dw_29_relu[0][0]
conv_pw_29_bn (BatchNormalizati)	(None, 30, 40, 21)	84	conv_pw_29[0][0]
conv_pw_29_relu (ReLU)	(None, 30, 40, 21)	0	conv_pw_29_bn[0][0]
tf.image.resize_1 (TFOpLambda)	(None, 60, 80, 21)	0	conv_pw_29_relu[0][0]
tf.image.resize_2 (TFOpLambda)	(None, 240, 320, 21)	0	tf.image.resize_1[0][0]
Total params: 139,908 Trainable params: 135,162 Non-trainable params: 4,746			

图 3.8: 模型网络结构信息

6. 模型训练

```
#学习率调整方案
def lr_scheduler(epoch, lr):
    if epoch > 0.9 * epochs:
        lr = 0.0001
    elif epoch > 0.5 * epochs:
        lr = 0.001
    elif epoch > 0.25 * epochs:
        lr = 0.01
    elif epoch > 0.1 * epochs:
        lr = 0.05
    else:
        lr = 0.1
    print('learning_rate: %f' % lr)
    return lr
scheduler = LearningRateScheduler(lr_scheduler)

#模型保存路径
current_dir = os.getcwd()
save_path = os.path.join(current_dir, 'Models')
if os.path.exists(save_path) is False:
    os.mkdir(save_path)

#训练时回调函数
callbacks = [scheduler]

#训练时checkpoint保存路径
checkpoint_path = os.path.join(save_path, 'checkpoint')
if os.path.exists(checkpoint_path) is False:
    os.mkdir(checkpoint_path)
checkpoint = ModelCheckpoint(filepath=os.path.join(checkpoint_path,
    'checkpoint_weights{epoch:03d}-{val_loss:.2f}.h5'), monitor='val_loss',
    save_weights_only=True).{epoch:d}
callbacks.append(checkpoint)

#随机梯度下降优化器
optimizer = SGD(learning_rate=lr_base, momentum=0.9)

#创建模型和编译模型
model = tiny_mobile_seg.Tiny_MobileNet()
model.compile(loss=loss_fn,
              optimizer=optimizer,
              metrics=metrics)

#获取数据集数量
def get_file_len(file_path):
    fp = open(file_path)
    lines = fp.readlines()
    fp.close()
    return len(lines)

#每个epoch迭代次数 = 数据集数量 / batch_size
steps_per_epoch = int(np.ceil(get_file_len(train_file_path) / float(batch_size)))
)
```

```
#启动训练
history = model.fit(
    x_y_train,
    steps_per_epoch=steps_per_epoch,
    epochs=epochs,
    callbacks=callbacks,
    workers=4,
    validation_data=x_y_validation,
    class_weight=class_weight)
```

参数说明:

- `LearningRateScheduler` : `tf.keras.callbacks.LearningRateScheduler(schedule, verbose=0)` 在每个 epoch 的开始，这个回调函数从提供的 `schedule` 函数中获取更新的学习率值，以及当前 `epoch` 和当前学习率，并将更新的学习率应用于优化器。
 - `schedule`: 一个函数，以 `epoch` 索引 (整数，从 0 开始索引) 和当前学习率 (浮点数) 作为输入，并返回一个新的学习率作为输出 (浮点数)。
 - `verbose`: 0: 不打印信息, 1: 打印信息。
- `model.compile`: 编译模型
 - `optimizer`: 优化器。本实验中使用的是 SGD — 随机梯度下降，即在一批次中随机取一个样本的梯度作为整个批次的训练梯度
 - `loss`: 损失函数。本实验中使用的是 `spare_categorical_crossentropy`—稀疏分类交叉熵，这是分类任务中最常用的是损失函数
 - `metrics`: 评价指标。本实验中使用的是 `accuracy`，即正确率 = 预测正确样本数 / 总样本数
- `model.fit`: 修理模型 (反向求导更新)
 - 第一个参数 `x_y_train` 为训练集
 - `validation_data`: 测试集
 - `steps_per_epoch`: 整数或 `None`。在声明一个轮次完成并开始下一个轮次之前的总步数 (样品批次)。使用 TensorFlow 数据张量等输入张量进行训练时，默认值 `None` 等于数据集中样本的数量除以 `batch` 的大小，如果无法确定，则为 1。
 - `callbacks`: 训练过程中的回调
 - `epochs`: 训练次数
 - `workers`: 整数。仅用于数据集生成器启动多线程加载数据集时最大线程数。如果未指定，`workers` 将默认为 1。
 - `class_weight`: 可选的字典，用来映射类索引 (整数) 到权重 (浮点) 值，用于加权损失函数 (仅在训练期间)。这可能有助于告诉模型「更多关注」来自代表性不足的类的样本。
 - 该方法返回值为 `history`, `history.history` 中包含了训练中每个 `epoch` 的各项指标值。

关于函数详细说明详见 https://tensorflow.google.cn/api_docs/python/tf/keras/Model?hl=zh-cn

```
# 查看每次训练后模型在验证集上的精度变化:
plt.plot(Acc[0],label='val_acc')
plt.legend()
plt.xlabel('epochs')
```

```
plt.ylabel('Acc')
plt.show()
```

7. 模型文件转成 RT-AK 部署所支持的格式 (.tflite 文件)

```
#保存为tflite文件
h5model = tiny_mobile_seg.Tiny_MobileNet(with_resize=False) #创建模型结构
h5model.load_weights('Models/seg_model.h5', by_name=True) #加载权重
conveter = tf.lite.TFLiteConverter.from_keras_model(h5model) #创建tflite转换器
tflite_model = conveter.convert() #转换
#保存
with open('seg_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

3.3 模型部署

在获取要部署的模型前，先运行Inference_Segmentation.ipynb代码，此为模型推理验证代码，可以在jupyter notebook中打开，读取图片验证模型预测效果。推理完成后会在Models文件夹下生成deeplabv3_noresize.h5模型文件，该模型文件删掉了原模型最后2层resize层，这样模型的最终输出会变小（原始输出数据量非常大，不利于在嵌入式端部署）。

在RT-AK/rt_ai_tools路径下打开Windows终端

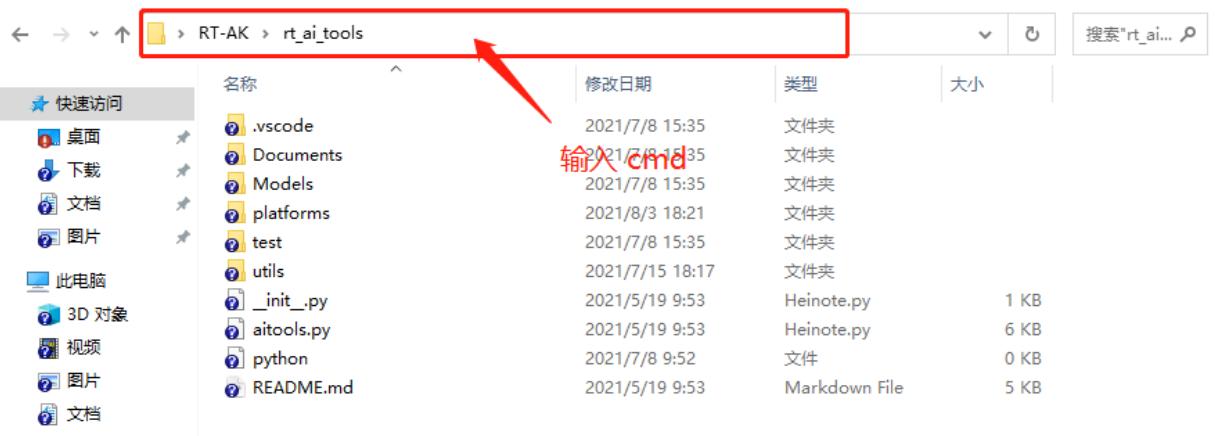


图 3.9: RT-AK 部署

关于量化和转换

--input_std 和 --input_mean 用于指定量化校准集的预处理方法。工具会先将你的图片转换为值域是 [0,1] 布局是 NCHW 的张量，之后会使用 $y = (x - \text{mean}) / \text{std}$ 公式对数据进行归一化。这里有一张参数的参考表。

输入值域	-input-std	-input-mean
[0,1] (默认)	1	0

输入值域	-input-std	-input-mean
[-1,1]	0.5	0.5
[0,255]	0.0039216	0

由于训练时，输入到模型中的图片是经过预处理的，并且图片像素值被缩放到 **[-120,130]** 之间，所以在进行量化时，也要输入相同的值域，注意参数中的 **--input_std=0.0039216 --input_mean=0.47**。量化数据集可直接从训练集中选出 **200-500** 张图片，复制到一个单独的文件夹中作为量化数据集。

```
# 量化为 uint8，使用 KPU 加速，量化数据集为图片
$ python aitoools.py --project=<your_project_path> --model=<your_model_path> --
  model_name=segmentation --platform=k210 --dataset=<your_val_dataset> --input_std
  =0.0039216 --input_mean=0.47
```

其中，**--project** 是你的目标工程路径，**--model** 是你的模型路径，**--model_name** 是转化的模型文件名，**--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，一般 200 张左右。

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几个文件：

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

同时，在 [RT-AK\rt_ai_tools\platforms\plugin_k210](#) 路径下会生成两个文件

文件	描述
segmentation.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

如果不想生成上述两个文件，可以在模型部署的时候命令行参数末尾加上： **--clear**

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，需要手工编写，详见下节“**3.4 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

3.4 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 Lab6_Segmentation\Applications。

3.4.1 代码流程

本实验中的应用层代码按照以下一个流程进行：

1. 系统内部初始化：

- 1.1 开启 KPU 时钟 `sysctl_clock_enable(SYSCTL_CLOCK_AI)`
- 1.2 lcd 初始化 `lcd_init()`
- 1.3 摄像头初始化 `rt_device_init()`

2. RT-AK Lib 模型加载并运行：

- 2.1 注册模型（部署过程中自动注册，无需修改）
- 2.2 找到模型
- 2.3 初始化模型，挂载模型信息，准备运行环境
- 2.4 运行（推理）模型
- 2.5 获取输出结果

3. 语义分割输出处理逻辑：

- 沿输出结果通道维找出最大值的索引
- 根据索引为其设置相应的颜色，得到分割图片

3.4.2 核心代码说明

核心代码及注释详见 Applications\demo.c，代码量较大在此不做粘贴。

```
/* 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

/* enable global interrupt，使能全局中断*/
sysctl_enable_irq();
```

```
/* system start */
printf("system start\n");
//定义RT-AI模型句柄
rt_ai_t deeplabv3 = NULL;
//查找模型
deeplabv3 = rt_ai_find(RT_AI_SEGMENTATION_MODEL_NAME);
if(!deeplabv3){
    rt_kprintf("rtak find model err!\n");
    return ;
}
//初始化句柄，传入图片地址
if(rt_ai_init(deeplabv3,(rt_ai_buffer_t*)kpu_image) != 0){
    ai_log("rtak init fail!\n");
    return ;
}

while (1){
    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
    g_ai_done_flag= 0;
    /* start to calculate */
    if(rt_ai_run(deeplabv3,ai_done,&g_ai_done_flag) != 0){
        ai_log("rtak run fail!\n");
        while (1){} ;
    }
    while(!g_ai_done_flag); //等待
    //获取计算结果
    float *output;
    size_t output_size;
    output = rt_ai_output(deeplabv3,0);
    //获得语义分割类别索引
    _argmax(output,label,30,40,21);
    for(int i=0;i<40*30;i++){
        //获得该位置像素类别索引
        rt_uint32_t idx = label[i];
        //设置相应类别颜色
        label[i]= palette[idx];
    }
    //LCD显示
    lcd_draw_picture(0, 0, 320, 240, display_image);
    lcd_draw_picture(0, 0, 40, 30, label);
}
```

模型输出结果处理说明：

```
/**
```

```
* @breif argmax函数,沿通道维选择值最大的通道索引
* @param in 张量输入
* @param out 结果输出
* @param height 输入张量的高
* @param width 输入张量的宽
* @param ch 输入张量的通道数
*/
static void _argmax(float *in, unsigned short *out, rt_uint32_t height, rt_uint32_t
width, rt_uint32_t ch){
    float store = 0.0;
    rt_uint16_t h, w, c = 0;
    for (h = 0; h < height; h++) {
        for (w = 0; w < width; w++) {
            store = 0.0;
            for (c = 0; c < ch; c++) {
                if (in[height * width * c + h * width + w] > store) {
                    out[h * width + w] = c;
                    store = in[height * width * c + h * width + w];
                }
            }
        }
    }
}
```

第 4 章

编译烧录

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。

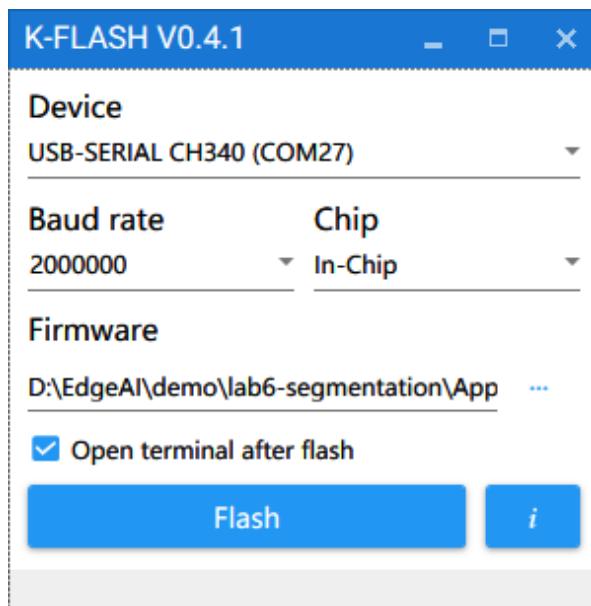
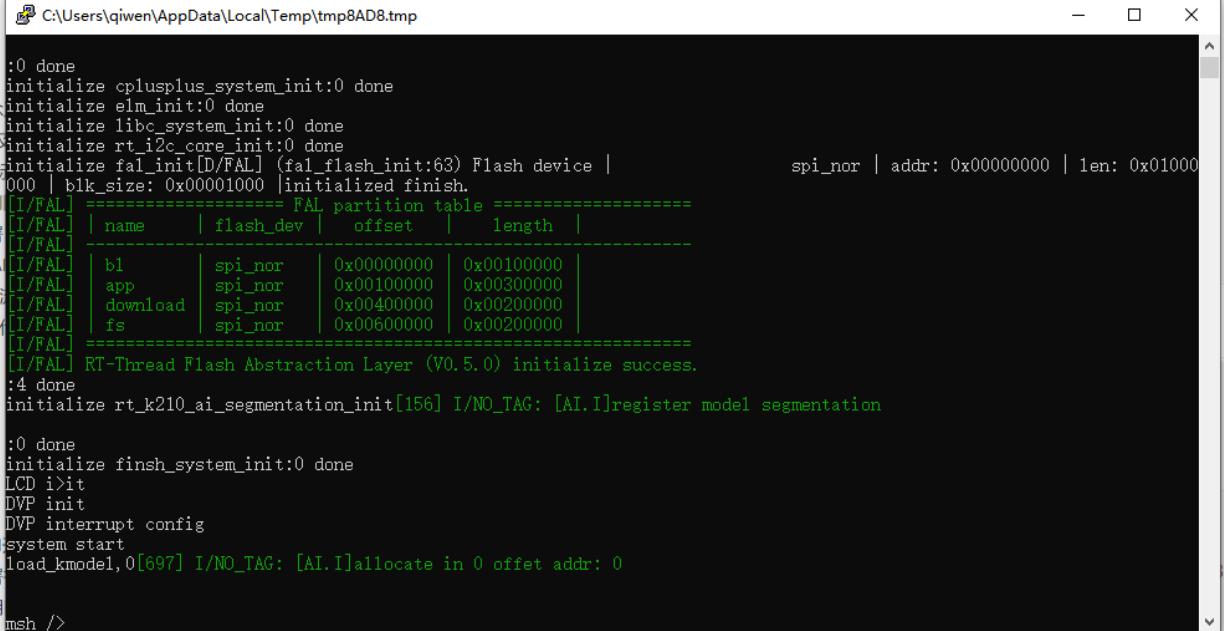


图 4.1: K-Flash

第 5 章

实验现象

如果编译 & 烧写无误，K-Flash 会自动打开 Windows 串口终端，自动连接实验板，系统启动后，我们的程序会自动运行，会在终端串口显示如下命令行。在 LCD 左上角会显示分割缩略图。



The screenshot shows a terminal window titled 'C:\Users\qiwen\AppData\Local\Temp\tmp8AD8.tmp'. The window displays the output of a K-Flash burn process. The output includes initialization logs for Cplusplus_system_init, Elm_init, Libc_system_init, and RT_I2C_core_init. It then shows the initialization of the Fal_flash_init module, which identifies a Flash device with a size of 0x00001000. A table is printed showing the FAL partition table with four entries: b1, app, download, and fs, all mapped to the spi_nor flash device at offsets 0x00000000, 0x00100000, 0x00300000, and 0x00400000 respectively. The length for each entry is 0x00200000. Following the table, a message indicates that the RT-Thread Flash Abstraction Layer (V0.5.0) has initialized successfully. The process then moves on to initialize rt_k210_ai_segmentation_init, which allocates memory for AI models. Finally, the system starts and loads the kmodel0[697] into memory at offset 0.

```
:0 done
initialize cplusplus_system_init:0 done
initialize elm_init:0 done
initialize libc_system_init:0 done
initialize rt_i2c_core_init:0 done
initialize fal_init[D/FAL] (fal_flash_init:63) Flash device |
| 000 | blk_size: 0x00001000 | initialized finish.
[I/FAL] ====== FAL partition table ======
[I/FAL] | name      | flash_dev | offset   | length   |
[I/FAL]
[I/FAL] | b1        | spi_nor   | 0x00000000 | 0x00100000 |
[I/FAL] | app       | spi_nor   | 0x00100000 | 0x00300000 |
[I/FAL] | download  | spi_nor   | 0x00400000 | 0x00200000 |
[I/FAL] | fs        | spi_nor   | 0x00600000 | 0x00200000 |
[I/FAL] ======
[I/FAL] RT-Thread Flash Abstraction Layer (V0.5.0) initialize success.
:4 done
initialize rt_k210_ai_segmentation_init[156] I/NO_TAG: [AI.I]register model segmentation

:0 done
initialize finsh_system_init:0 done
LCD i>it
DVP init
DVP interrupt config
System start
load_kmodel0[697] I/NO_TAG: [AI.I]allocate in 0 offset addr: 0

msh />
```

图 5.1: 烧录结果



图 5.2: *display*

第 6 章

附录

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);
```

Paramters	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);
```

Paramters	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