实验五　CIFAR10 分类

RT-AK 教育套件实验手册

WWW.RT-THREAD.ORG

Monday 22nd November, 2021

目录

第 1 章

实验介绍和目的

1.1 实验介绍

本实验是基于 Tensorflow 训练的一个 AI 模型：Cifar10 图像分类模型。

Cifar10 图像分类模型的主要功能是：输入任意一张包含于 10 分类中的物体图片，识别出物体的类别。

该模型目的明确、任务较简单，训练数据集可使用 Tensorflow API 直接下载调用。

本实验数据集较大，一般使用 GPU 训练，但使用 CPU 同样也可以训练（训练速度较慢）。

数据集中一共有 50000 张训练图片和 10000 张测试图片，每张图片的尺寸为 32×32 。

Cifar10 是由 Hinton 的学生 Alex Krizhevsky 和 Ilya Sutskever 整理的一个用于识别普适物体的小型数据集。一共包含 10 个类别的 RGB 彩色图片：飞机（airplane ）、汽车（automobile ）、鸟类（bird ）、猫（cat ）、鹿（deer ）、狗（dog ）、蛙类（frog ）、马（horse ）、船（ship ）和卡车（truck ）。

Cifar10 的图片样例如下图所示：

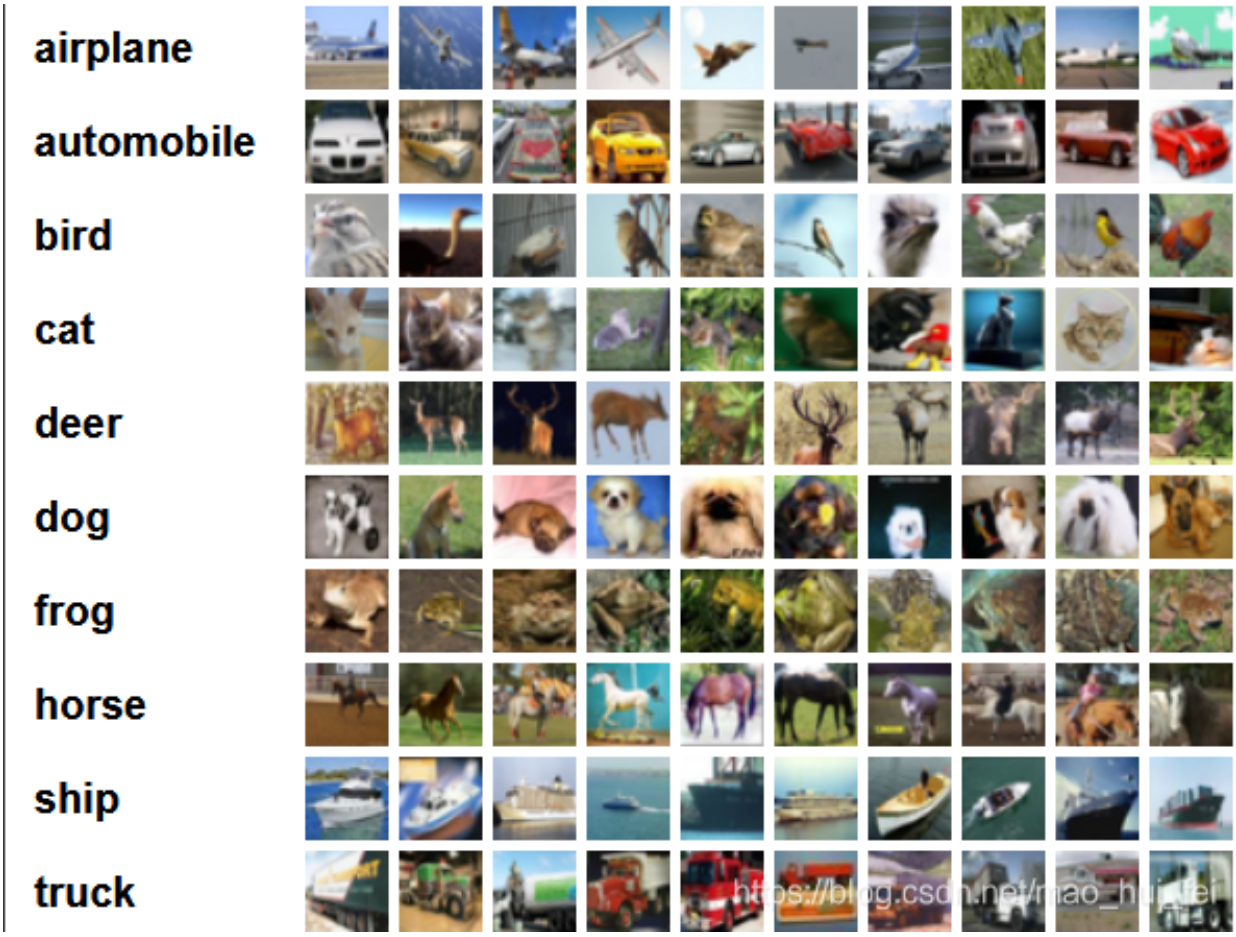

图 1.1: *cifar10 数据示例*

1.2 实验目的

1. 掌握使用 Tensorflow 框架训练 Cifar10 图像分类模型
2. 掌握使用 RT-AK 一行命令部署 AI 模型
3. 完成嵌入式 AI 开发：输入一张物体图片，成功实现一次模型推理

第 2 章

实验器材

1. 上位机（电脑）
2. EgdeAI 实验板

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

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 Tensorflow 官方文档 https://tensorflow.google.cn/install/gpu?hl=zh-cn。

3.1 AI 模型训练

1. 在指定路径打开 `Jupyter notebook`, 并打开 `cifar10_train.ipynb` 模型训练文件

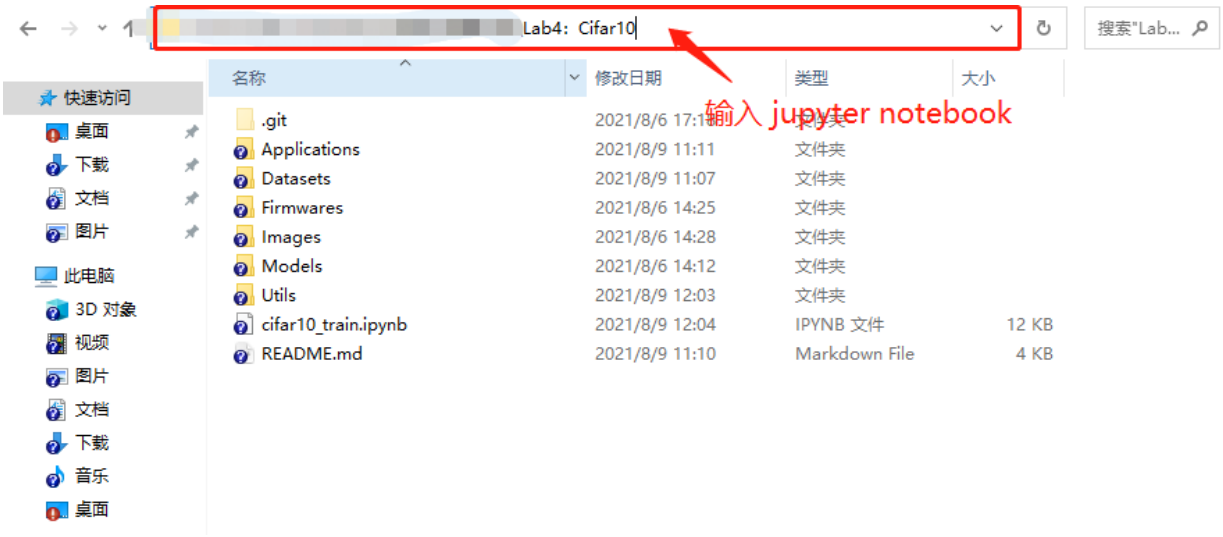

图 3.1: *AI 模型训练步骤 1*

图 3.2: *AI 模型训练文件*

以下为模型训练代码，内容摘自 cifar10_train.ipynb 文件

2. 导入库

导入所需要的 python 库，点击 Run 运行按钮，运行代码

```
import os
# 使用标号为 “0” 的 GPU 训练，若无 GPU，该行代码会产生一些 warning，此时可以将
    下行注释掉
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import backend
from tensorflow.keras import layers
import numpy as np
from matplotlib import pyplot as plt
```

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. 准备数据集

这里我们使用 Tensorflow 提供的 API 来载入 Cifar10 数据集。训练集中包含 50,000 个示例图像，测试集包含了 10,000 个示例图像，每张图像的尺寸为 32x32，为 RGB 图像，即 3 通道数。

```
# 对数据进行预处理，归一化
def preprocess_input(inputs, std=255. ,mean=0., expand_dims=None):
    inputs = tf.cast(inputs,tf.float32)
    inputs = (inputs - mean) / std
    if expand_dims is not None:
        np.expand_dims(inputs,expand_dims)
    return inputs

# 图像增强
def img_aug_fun(elem):
    elem = tf.image.random_flip_left_right(elem)  # 左右翻转
    elem = tf.image.random_brightness(elem, max_delta=0.5)  # 调亮度
```

```
    elem = tf.image.random_contrast(elem, lower=0.5, upper=1.5)  # 调对比度
    elem = preprocess_input(elem)
    return elem

# load CIFAR10 dataset, size(32,32,3)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
assert x_train.shape == (50000, 32, 32, 3) # 训练集图像 (N, H, W, C)
assert x_test.shape == (10000, 32, 32, 3)  # 测试集图像
assert y_train.shape == (50000, 1)  # 训练集标签
assert y_test.shape == (10000, 1)   # 测试集标签

x_test = preprocess_input(x_test)
x_train_ds = tf.data.Dataset.from_tensor_slices(x_train).map(img_aug_fun)
y_train_ds = tf.data.Dataset.from_tensor_slices(y_train)
x_y_train_ds = tf.data.Dataset.zip((x_train_ds,y_train_ds))
x_y_train_ds = x_y_train_ds.batch(128)  # 批处理，batch=128
```

在这里，我们自定义了一个 preprocess_input的函数，用于对原始数据进行归一化。预处理归一化的最大好处就是使得模型训练更稳定。接着又自定义了 img_aug_fun 函数，用于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。

数据集下载补充说明

该实验中我们使用 API: tf.keras.datasets.cifar10.load_data()，功能是载入数据集，过程如下：

- 先下载数据集的压缩包，默认保存地址为 C:/Users/Admin/.keras/datasets，(Admin 为用户名)

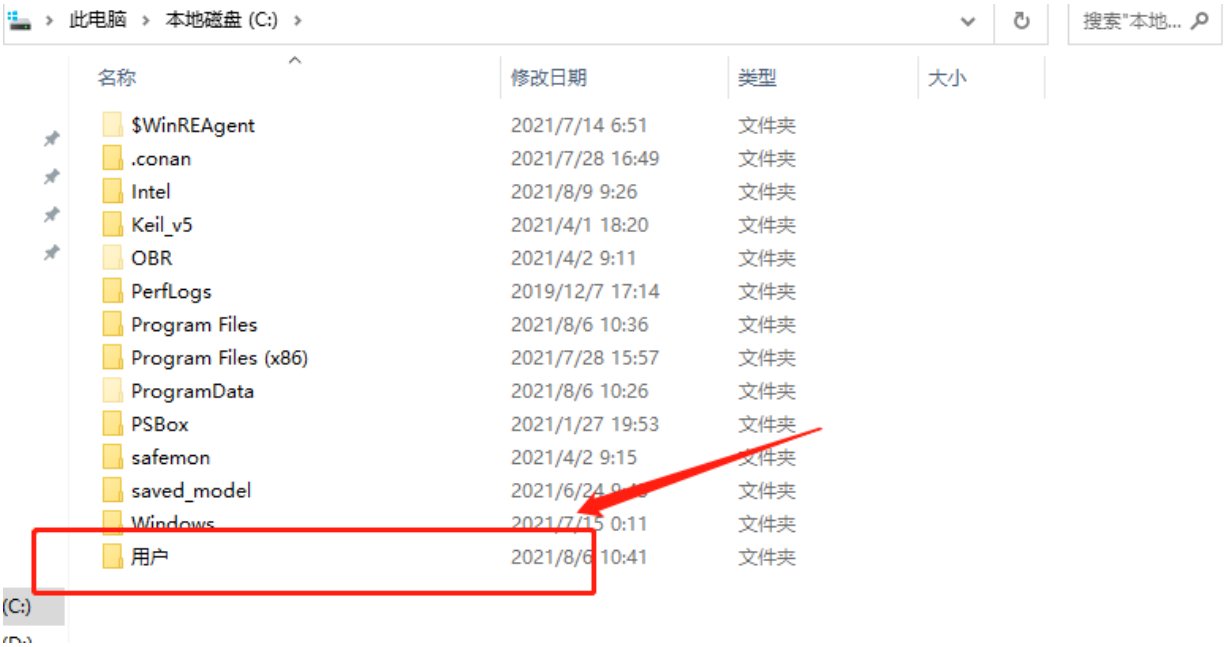

图 3.3: *数据集保存至 1*

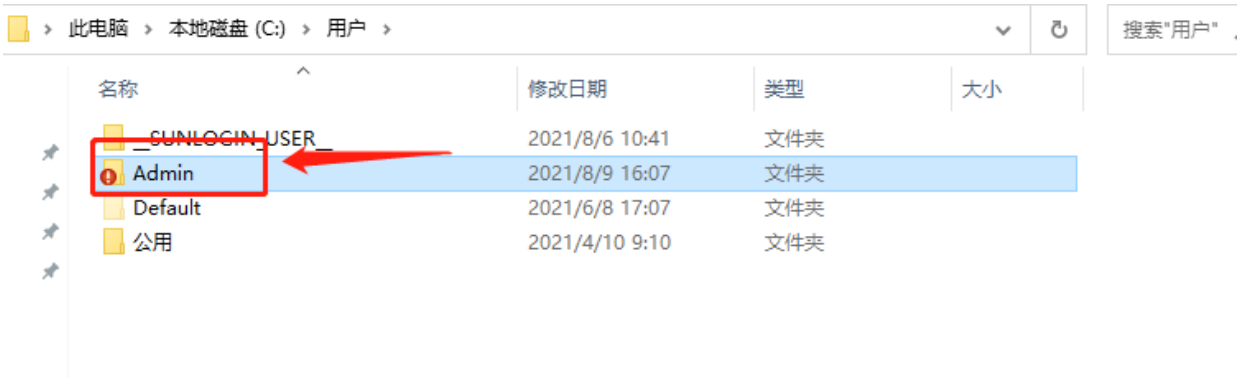

图 3.4: *数据集保存地址 2*

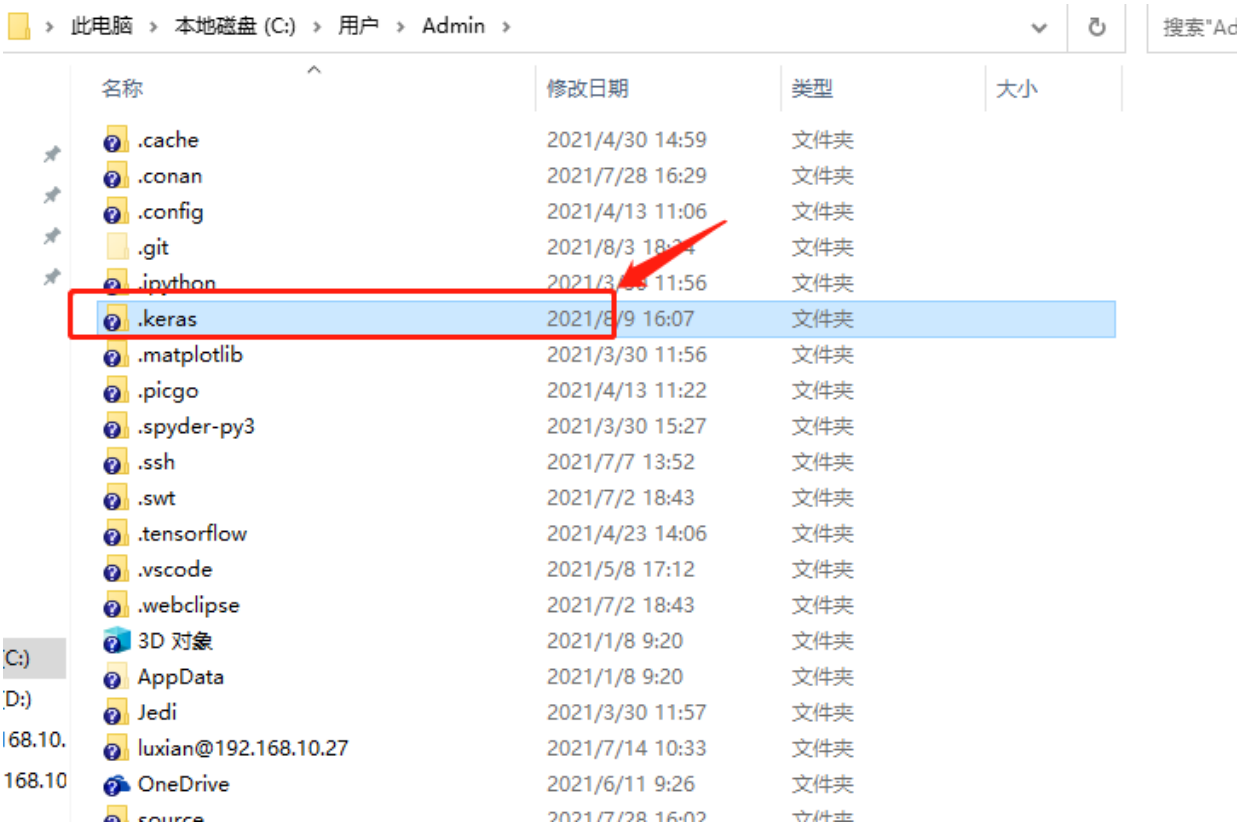

图 3.5: *数据集保存地址 3*

- 然后解压该压缩包返回训练集和测试集 (API 会自动解压)

  若训练过程中直接使用 API 下载速度过慢或出错，可以将本实验提供的`Datasets/cifar-10-batches-py.tar.gz` 文件复制到上面提到的默认保存地址中（其他不变，如下图所示），然后运行代码。**(PS: 不要解压该文件！一定要复制压缩包！)**

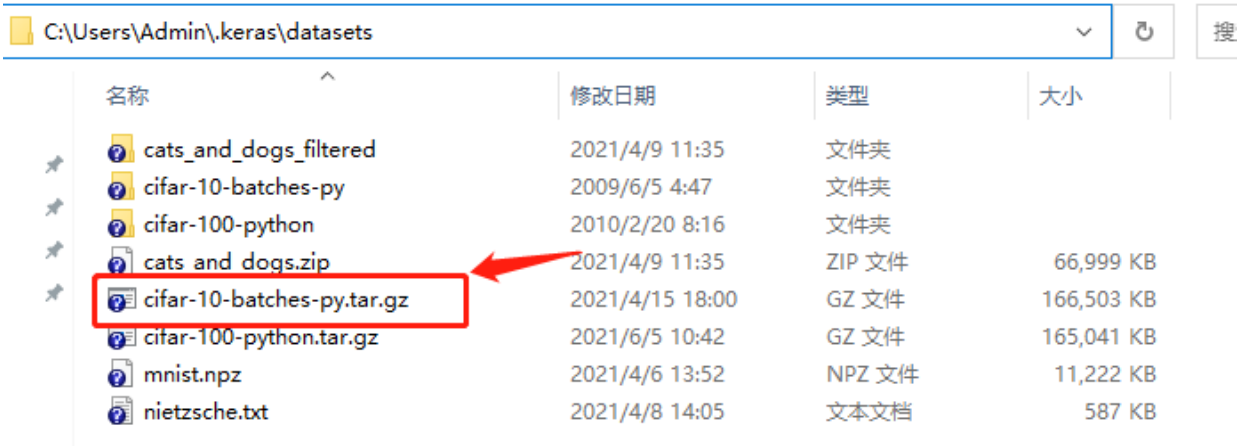

图 3.6: *数据集保存地址*

4. 搭建神经网络模型

我们基于 CNN 来搭建这个模型，每个卷积后按顺序加一个归一化操作（BatchNormalization）和激活函数（ReLU），归一化操作可以使得模型训练更加稳定、提高训练速度，而ReLU是目前最常用的激活函数，具有增加模型的非线性表达能力、求导方便等诸多好处；并且我们使用了 Dropout 方法给模型加入了正则化，以防止过拟合。在模型的最后，我们使用 **10 个神经元的全连接层**以及 Softmax 层，输出对应 10 个分类的预测概率值，用于分类预测。

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

```
# create CNN
# input_shape 为模型输入形状，本实验中为（32,32,3），dropout 为 Dropout 率
def CNNmodel(input_shape,filters=64, kernel=(3,3),size=4,dropout=0.2):
    _inputs = layers.Input(shape=input_shape)
    x = layers.Conv2D(8,(3,3),padding='same',use_bias=False,strides=(2,2), name=
        'conv_0')(_inputs)
    x = layers.BatchNormalization(axis=-1, name='conv_0_bn')(x)
    x = layers.ReLU(6., name='conv_0_relu')(x)

    x = layers.Conv2D(16,(3,3),padding='same',use_bias=False,strides=(2,2), name
        ='conv_1')(_inputs)
    x = layers.BatchNormalization(axis=-1, name='conv_1_bn')(x)
    x = layers.ReLU(6., name='conv_1_relu')(x)
    # 构建相同的 卷积+归一化+relu 层
    for block_id in range(2,size+2):
        x = layers.Conv2D(filters,kernel,padding='same',use_bias=False,strides
            =(1,1), name='conv_%d'%block_id)(x)
        x = layers.BatchNormalization(axis=-1, name='conv_%d_bn'%block_id)(x)
        x = layers.ReLU(6., name='conv_%d_relu'%block_id)(x)

    x = layers.GlobalAveragePooling2D()(x) # 全局平均池化
    x = layers.Dropout(dropout, name='dropout')(x)
    x = layers.Dense(10)(x)  # 10个神经元的全连接层
    x = layers.Softmax()(x)  # Softmax 层，输出10个分类的概率值
    return keras.Model(inputs=_inputs,outputs=x)
```

我们可以使用神经网络可视化工具 Netron (下载地址：https://www.electronjs.org/apps/netron) 来查看模型的网络结构：

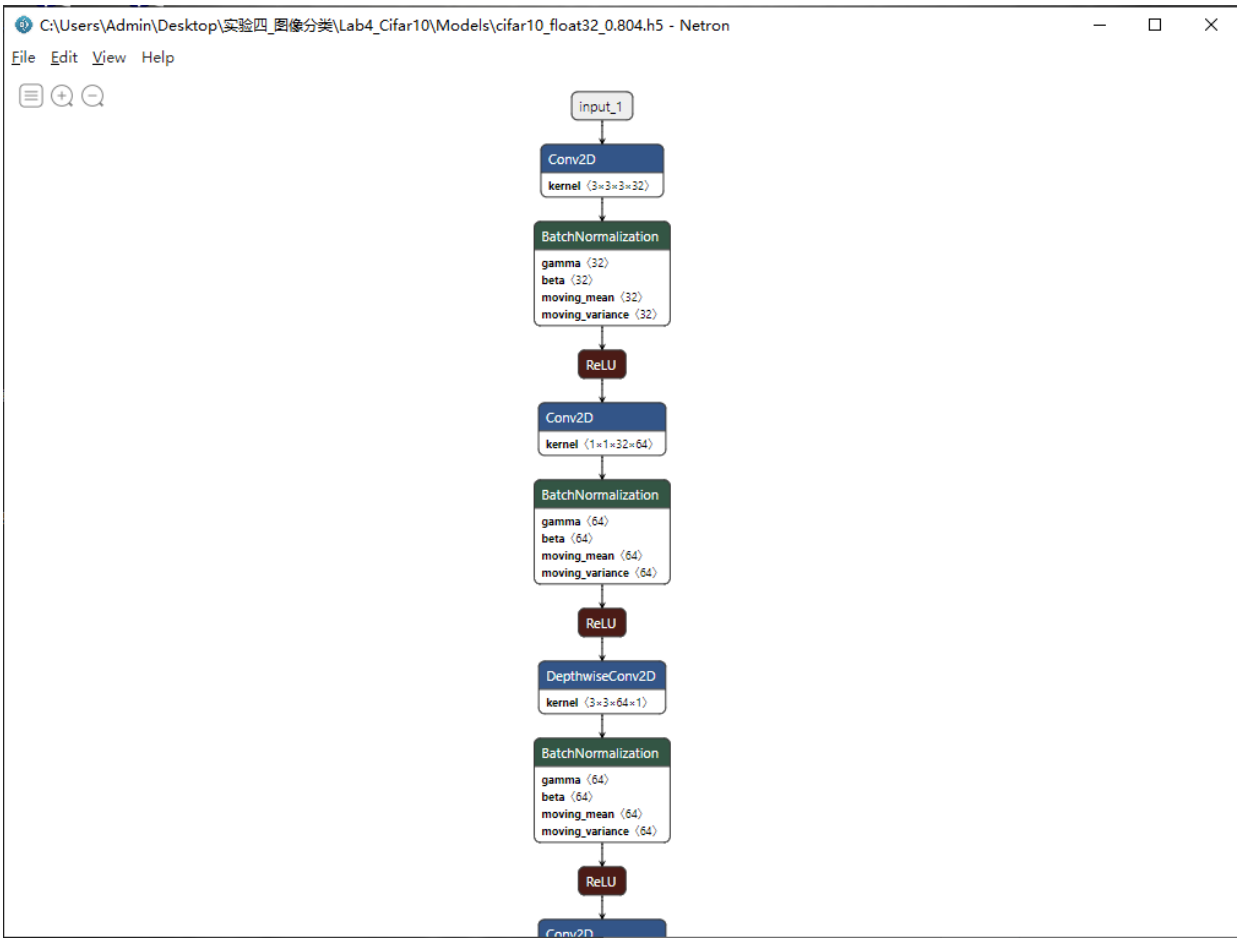

图 3.7: *模型网络结构*

使用 API 构建模型之后，我们也可以使用 `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 = CNNmodel(input_shape=(32,32,3),filters=64, kernel=(3,3),size=9) # 实例化模型
model.summary()
```

```
Model: "model_1"
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 32, 32, 3)]	0
conv_1 (Conv2D)	(None, 16, 16, 16)	432
conv_1_bn (BatchNormalizatio	(None, 16, 16, 16)	64
conv_1_relu (ReLU)	(None, 16, 16, 16)	0
conv_2 (Conv2D)	(None, 16, 16, 64)	9216
conv_2_bn (BatchNormalizatio	(None, 16, 16, 64)	256
conv_2_relu (ReLU)	(None, 16, 16, 64)	0
conv_3 (Conv2D)	(None, 16, 16, 64)	36864
conv_3_bn (BatchNormalizatio	(None, 16, 16, 64)	256
conv_3_relu (ReLU)	(None, 16, 16, 64)	0
conv_4 (Conv2D)	(None, 16, 16, 64)	36864
conv_4_bn (BatchNormalizatio	(None, 16, 16, 64)	256
conv_4_relu (ReLU)	(None, 16, 16, 64)	0
conv_5 (Conv2D)	(None, 16, 16, 64)	36864
conv_5_bn (BatchNormalizatio	(None, 16, 16, 64)	256
conv_5_relu (ReLU)	(None, 16, 16, 64)	0
conv_6 (Conv2D)	(None, 16, 16, 64)	36864
conv_6_bn (BatchNormalizatio	(None, 16, 16, 64)	256
conv_6_relu (ReLU)	(None, 16, 16, 64)	0
conv_7 (Conv2D)	(None, 16, 16, 64)	36864

图 3.8: *模型网络结构信息*

5. 模型训练

```
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='accuracy', factor=0.5,
    patience=4, min_lr=0.0001,verbose=1)  # 当 accuracy 不再提升时，减小学习率
earlystop = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=8,
    verbose=1)  # 当测试集准确率不再提高，停止训练
```

```
model.compile(optimizer='SGD',loss='sparse_categorical_crossentropy',metrics=['
    accuracy']) # 指定优化器、损失函数和评价指标
history = model.fit(x_y_train_ds,validation_data=(x_test,y_test),callbacks=[
    reduce_lr,earlystop],verbose=2,epochs=500) # 根据参数训练模型
```

参数说明：

- `keras.callbacks.ReduceLROnPlateau`：当 monitor 在 patience 个 epoch 中没出现明显变化，则会由因子 factor 来降低学习率，最小降低到 min_lr 。
  - monitor：监控指标
  - factor：new_learning_rate = old_learning_rate * factor
  - patience: 监控指标没有明显提高的训练次数
  - min_lr：学习率的下界
  - verbose：是否打印出信息，1 为打印出信息
- `keras.callbacks.EarlyStopping`: 当 monitor 在 patience 个 epoch 中没出现明显变化，则会停止训练。
  - monitor：监控指标
  - patience: 监控指标没有明显提高的训练次数
  - verbose：是否打印出信息，1 为打印出信息
- `model.compile`: 指定优化器、损失函数和评价指标
  - 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`: 根据参数训练模型
  - 第一个参数 x_y_train_ds 为训练集
  - validation_data：测试集
  - callbacks：训练过程中的回调
  - epochs：训练次数
  - 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。

  其他参数说明详见 Tensorflow API 文档 https://tensorflow.google.cn/api_docs/python/tf/keras

最终训练过程如下图所示：

```
Epoch 50/500
391/391 - 14s - loss: 0.5735 - accuracy: 0.8049 - val_loss: 0.5938 - val_accuracy: 0.7953
Epoch 51/500
391/391 - 14s - loss: 0.5700 - accuracy: 0.8064 - val_loss: 0.7173 - val_accuracy: 0.7630
Epoch 52/500
391/391 - 13s - loss: 0.5681 - accuracy: 0.8055 - val_loss: 0.6121 - val_accuracy: 0.7933
Epoch 53/500
391/391 - 14s - loss: 0.5566 - accuracy: 0.8088 - val_loss: 0.6420 - val_accuracy: 0.7791
Epoch 54/500
391/391 - 14s - loss: 0.5563 - accuracy: 0.8103 - val_loss: 0.6321 - val_accuracy: 0.7854
Epoch 55/500
391/391 - 14s - loss: 0.5431 - accuracy: 0.8134 - val_loss: 0.8218 - val_accuracy: 0.7380
Epoch 56/500
391/391 - 13s - loss: 0.5426 - accuracy: 0.8141 - val_loss: 0.6992 - val_accuracy: 0.7706
Epoch 57/500
391/391 - 13s - loss: 0.5331 - accuracy: 0.8179 - val_loss: 0.8830 - val_accuracy: 0.7286
Epoch 58/500
391/391 - 14s - loss: 0.5372 - accuracy: 0.8156 - val_loss: 0.6566 - val_accuracy: 0.7785
Epoch 00058: early stopping
```

图 3.9: *模型训练*

查看每次训练后模型在验证集上的精度变化：

```
plt.plot(history.history['val_accuracy'],label='val_acc')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Acc')
plt.show()
```

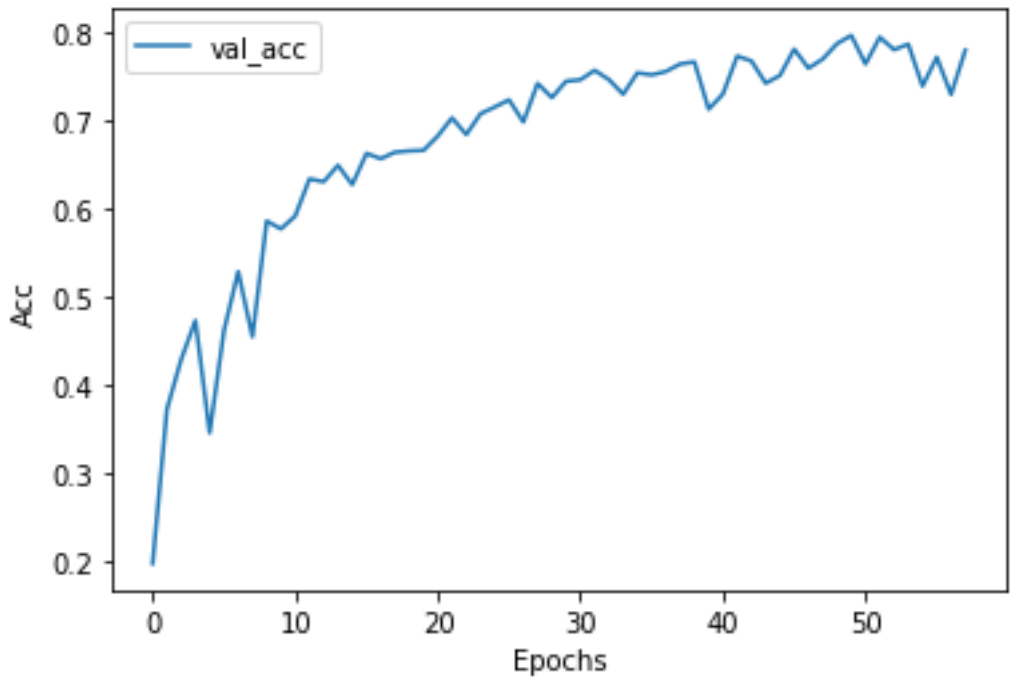

图 3.10: *模型训练测试集准确率变化*

6. 模型文件保存（.h5 文件）

这里保存的模型文件包含**模型权重**和**网络结构**

```
# 这里将最后一个 epoch 的测试集准确率加入到模型文件名中
keras_file = './Models/Cifar10_CNN_%.3f'%history.history['val_accuracy'][-1]+'.
    h5'
model.save(keras_file) # 保存文件名中尾缀为 .h5 则默认保存格式为 h5 模型
```

7. 模型文件转成 **RT-AK** 部署所支持的格式（**.tflite** 文件）

```
# keras 模型转 tflite，后者模型会更小一点，算子支持更多
model = tf.keras.models.load_model(keras_file)   # keras_file 即 h5 模型文件路径
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
tflite_file = "./Models/cifar10.tflite"  # tflite 模型保存路径
with open(tflite_file, 'wb') as f:   # 以二进制类型将模型写进tflite文件中
    f.write(tflite_model)
```

3.2 模型信息

已训练好的模型文件在文件夹 `Models`中，后缀 h5 的为 keras 模型文件，后缀 tflite 的为 tflite 模型文件，**后者更适合部署**。模型训练时，所有数据都是 float32 类型，关于模型其他信息，我们需要说明以下几点：

- 本实验训练的 cifar10 模型结构简单，总参数量为 307578，tflite 模型文件大小为 346 KB，适合部署至 K210
- h5 模型文件转为 tflite 时，会将 `Conv2D`、`Batchnormalization` 和 `ReLU` 三个算子融合在一起，优化了模型结构
- 模型输入格式：NHWC，类型为 float32
- 模型输出格式：N*10，类型为 float32

3.3 模型部署

在 `RT-AK/rt_ai_tools` 路径下打开 Windows 终端

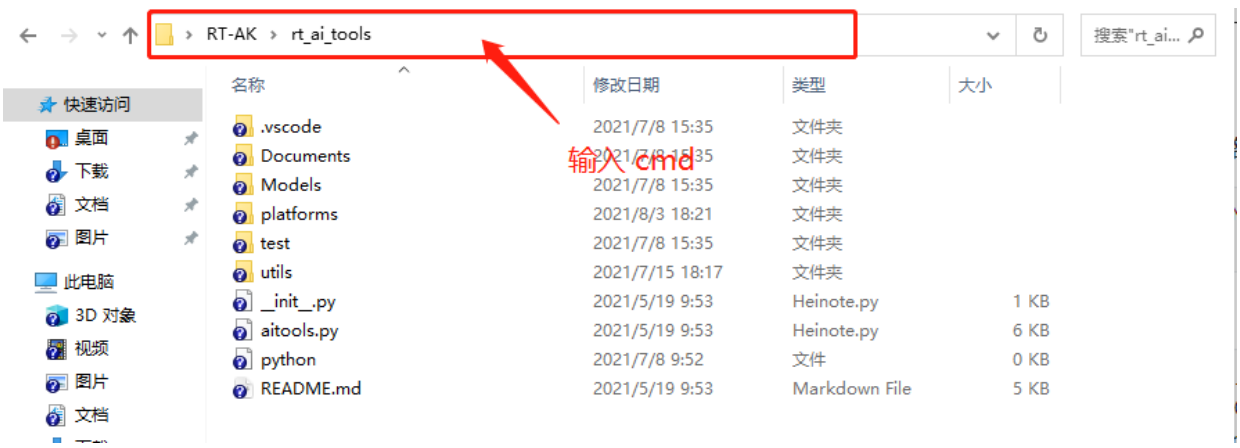

图 3.11: *RT-AK 部署*

在终端输入以下命令：

```
# 量化为 uint8，使用 KPU 加速，量化数据集为图片
$ python aitools.py --project=<your_project_path> --model=<your_model_path> --
    model_name=cifar10_mb --platform=k210 --dataset=<your_quant_dataset>
```

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/cifar10_mb_kmodel.c	kmodel 的十六进制储存
applications/rt_ai_cifar10_mb_model.c	与目标平台相关的信息
applications/rt_ai_cifar10_mb_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
cifar10_mb.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`Lab5_Cifar10\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 系统时钟初始化 `sysctl_clock_enable(SYSCTL_CLOCK_AI)`

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. **Cifar10 业务逻辑层：**

   3.1 找出输出最大值的索引

3.4.2 核心代码说明

核心代码详见 `Lab5_Cifar10\Applications\main.c`

```
/* Set CPU clock */
sysctl_clock_enable(SYSCTL_CLOCK_AI);  // 使能系统时钟（1.1 系统时钟初始化）
...

// 2.1 注册模型的代码在 rt_ai_cifar10_mb_model.c 文件下的第31行，代码自动执行
// 模型的相关信息在 rt_ai_cifar10_mb_model.h 文件

/* AI modol inference */
mymodel = rt_ai_find(MY_MODEL_NAME);  // 2.2 找到模型rt_ai_cifar10_mb_model.h 文件中
    有模型相关信息声明，命名格式 RT_AI_<model_name>_MODEL_NAME

if (rt_ai_init(mymodel,(rt_ai_buffer_t *)input_chw_data) != 0)  // 2.3 初始化模型
...
if(rt_ai_run(mymodel, ai_done, NULL) != 0)    // 2.4 模型推理一次
...

output = (float *)rt_ai_output(mymodel,0);  // 2.5 获取模型输出结果

/* 3.1 对模型输出结果进行处理，本实验是cifar10，输出结果为10个概率值，选出其中最大概
    率即可 */
for (int i=0;i<MY_MODEL_OUT_1_SIZE;i++){
    if(output[i]>output[pred])
        pred = i ;
}

rt_kprintf("The prediction is : %s\n", label[pred]); // 在终端打印出预测结果
...
```

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。

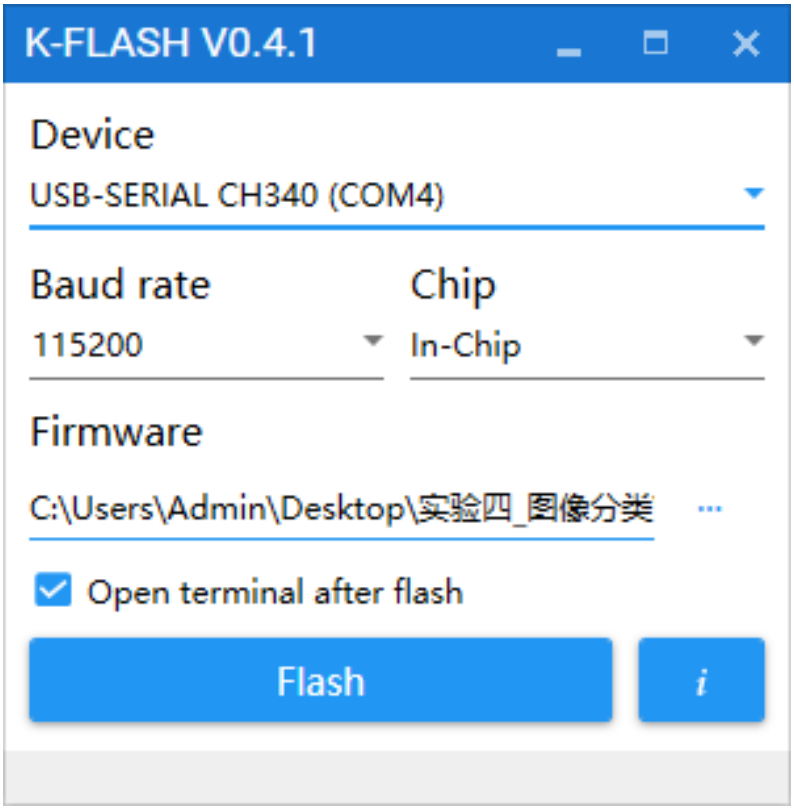
K-FLASH V0.4.1
Device
USB-SERIAL CH340 (COM4)
Baud rate
115200
Chip
In-Chip
Firmware
C:\Users\Admin\Desktop\实验四_图像分类
Open terminal after flash
Flash
i

图 **4.1:** *K-Flash*

第 5 章

实验现象

5.1 实验现象

如果编译 & 烧写无误，K-Flash 会自动打开 Windows 终端，自动连接实验板，系统启动后，我们的程序会自动运行，会在终端串口显示预测的结果。在 LCD 上会显示被预测图片和预测类别结果。

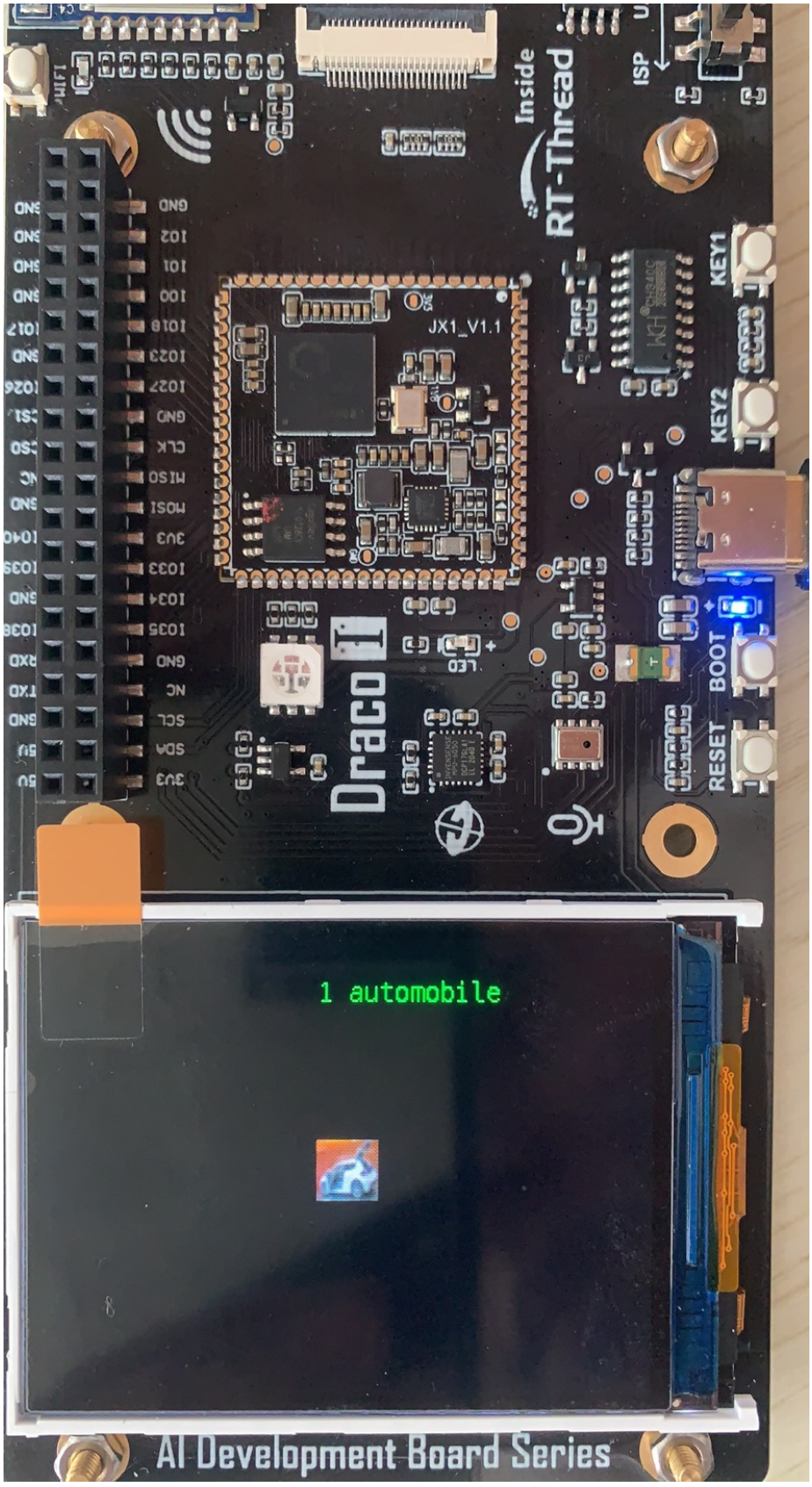
Inside
RT-Thread
ISP
KEY1
KEY2
BOOT
RESET
JX1_V1.1
Draco I
1 automobile
AI Development Board Series

图 **5.1:** *烧录结果*

5.2 如何使用自定义图片作为模型输入

PC 端处理自定义模型输入图片，保存为.h 头文件

- K210 指定的图片数据输入格式是 CHW （chanel，heigh，width）

  如果是灰度图，C=1，如果是 RGB 图，C=3
- cifar10 分类模型输入为 3x32x32 （数据类型为 uint8）
- cifar10 分类模型输出是 1x10 （数据类型为 float32）

  将自定义图片转换为部署到 K210 中的模型所需格式的步骤：

1. 读取图片

```
image_raw = cv2.imread(image_path)
```

2. 转成 RGB 图（cv2 默认读取图片以 BGR 格式，而训练的时候是 RGB 格式训练，需要训练的输入格式保持一致）

```
image = cv2.CvtColor(image_raw, cv2.COLOR_BGR2RGB)
```

3. 将自定义图片 resize 成尺寸为 32x32

```
image = cv2.resize(image, shape)  # 本实验中 shape=(32, 32)
```

4. 保存为文件

```
with open(dataset_h, "w+") as f:
    print(f"#ifndef _{dataset_h.stem.upper()}_H_\n#define _{dataset_h.stem.upper
        ()}_H_\n", file=f)
    print(f"// {image_path} {shape}, HW", file=f)
    print(f"const static uint8_t {dataset_h.stem.upper()}[] __attribute__((
        aligned(128))) = {'{'}", file=f)
    # print(", ".join([str(i) for i in image.flatten()]), file=f)
    print(", ".join(map(lambda i: str(i), image.flatten())), file=f)
    print("};\n\n#endif", file=f)
```

详细的代码在：`Utils/save_chw_img.py`，根据所提供的 `dataset_h` 参数，其文件保存在对应目录下。该代码的使用详见 `Utils` 中的说明文档。

示例数据在 `Applications/testdata` 文件夹下，模型不用重新训练

第 6 章

API 使用说明

6.1 嵌入式 AI 开发 API 文档

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

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

描述: 查找已注册模型

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

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

描述: 初始化模型句柄, 挂载模型信息, 准备运行环境.

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

Paramaters	Description
ai	rt_ai_t 模型句柄

Paramaters	Description
callback	运行完成回调函数
arg	运行完成回调函数参数
Return	–
0	成功
非 0	失败

描述: 模型推理计算

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

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

描述: 获取模型运行的结果, 结果获取后.

`rt_ai_libs/readme.md` 文件中有详细说明

6.2 LCD API 说明手册

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

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

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

```
*/
```