
实验八 车牌识别

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

Tuesday 23rd 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	2
2 实验器材	3
3 实验步骤	4
3.1 自定义代码详解	4
3.2 AI 模型训练	11
3.3 模型信息	15
3.4 模型部署	15
3.5 嵌入式 AI 模型应用	17
3.5.1 代码流程	17
3.5.2 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	22
6 附录	24
6.1 嵌入式 AI 开发 API 文档	24
6.2 LCD API 说明手册	25

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 AI 模型：车牌识别模型。

车牌识别模型的主要功能是：输入任意一张中文车牌图片，识别出车牌字符。该模型目的明确、任务较困难，训练数据集是经过一个处理过程的 CCPD2019 中文车牌数据集。

本实验数据集大，训练集达到 20 多万张，强烈建议使用 GPU 训练，但使用 CPU 同样也可以训练（训练速度极慢）。数据集中一共有 208075 张训练图片和 6436 张测试图片，每张图片的尺寸为 94x24 (W, H)。

传统车牌检测和识别都是在小规模数据集上进行实验和测试，所获得的算法模型无法胜任环境多变、角度多样的车牌图像检测和识别任务。为此，中科大团队建立了 CCPD 数据集，这是一个用于车牌识别的大型国内停车场车牌数据集，该团队同事在 ECCV2018 国际会议上发了相应论文，论文和数据集下载地址：<https://github.com/detectRecog/CCPD>。

本实验采用的是 CCPD2019 年度的数据集，经过处理将原数据集中的车牌图截下来，并缩放成 94x24 尺寸用于本模型进行训练。

CCPD2019 数据集原图的图片样例如下图所示：

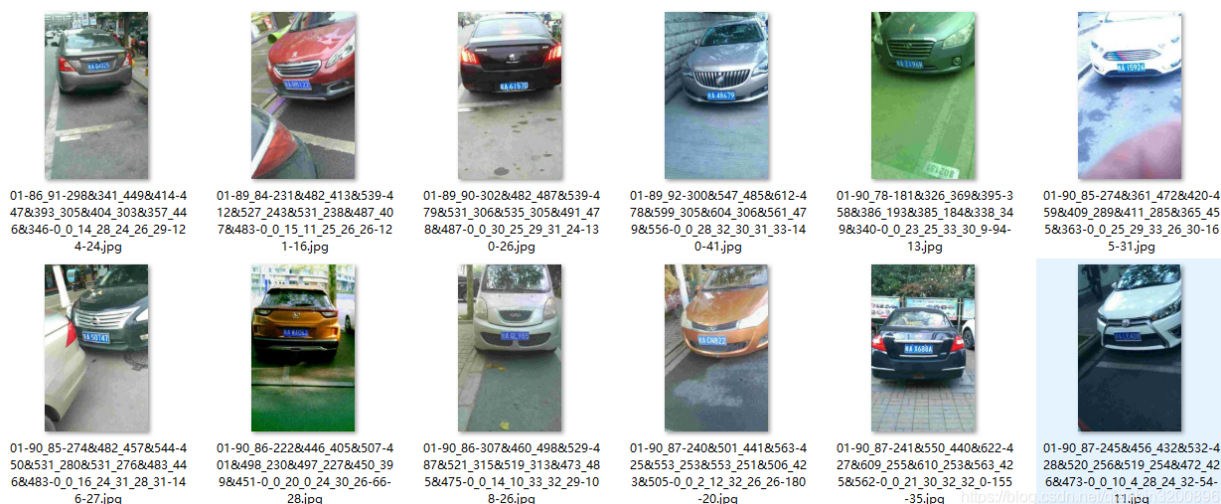


图 1.1: CCPD2019 数据集原图

本实验车牌数据示例：



图 1.2: 实验 8 训练数据示例

本实验中采取的损失函数为 CTC (Connectionist Temporal Classification)。CTC 是一种避开输入和输出手动对齐的一种方式，非常适合输入输出不对齐的一种字符预测任务。想较深入了解 CTC 推荐参考 Hannun 等人在 distill.pub 发表的文章 (<https://distill.pub/2017/ctc/>)，或者去看论文原文：《Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks》。本实验对模型输出采取贪婪算法处理，最终得到本实验模型预测的车牌字符。

1.2 实验目的

1. 掌握使用 Tensorflow 框架训练车牌识别模型
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：输入一张中文车牌图片，成功实现一次模型推理，并将车牌图片和预测车牌字符打印到 LCD 上

第 2 章

实验器材

1. 上位机（电脑）
2. EgdeAI 实验板
3. 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：Tensorflow GPU 版及相关驱动安装参考 <https://tensorflow.google.cn/install/gpu?hl=zh-cn>。

3.1 自定义代码详解

本实验中使用的评价函数、数据生成函数非 **Tensorflow API**，为自定义文件，并且模型结构搭建代码较多，故多以 **Python** 包的形式进行存放，各部分定义文件如下：

- 评价函数定义文件：`./Model_train/metric/metric.py`
- 数据生成函数定义文件：`./Model_train/dataload/data.py`
- 模型结构搭建代码文件：`./Model_train/model/LPRNet.py`

下面对这些文件中的代码进行详细讲解：

1. 评价函数定义

本实验采取**贪婪算法**对模型输出进行处理，最终得到预测车牌字符。贪婪算法也即暴力解法，算法思路就是通过求得每个时间步中的字符输出概率最大的索引为该时间步所预测的字符。最终我们也将用 **c** 语言来实现，具体 **c** 代码将在 3.5 章节中介绍。评价函数代码在 `./Model_train/metric/metric.py` 文件中，具体如下：

```
def metric(test_dataset, lprnet):  
    '''  
    参数： 1. test_dataset: 测试数据集，为 tf.keras.data.Dataset 实例  
          2. lprnet: 模型实例  
    返回：  
          1. acc: 在测试集上的准确率  
          2. tp: 预测正确样本数量  
          3. tp_error: 预测错误样本数量  
          4. t: 测试总时间  
    '''  
    tp, tp_error = 0, 0 # tp 为正确预测数量，tp_error 是错误预测数量
```

```

start_time = time.time() # 记录起始时间
# 遍历各个batch 中的数据
for cur_batch, (test_imgs, test_labels) in enumerate(test_dataset):
    test_labels = tf.cast(test_labels, tf.int32) #将test_labels转为tf.int32
    数据类型
    prebs = lprnet(test_imgs) # 得到模型输出,(batch_size, class_num,
    frames)
    preb_labels = list() # 储存每个batch中的预测序列 (batch_size, frames
    )
    for i in range(prebs.shape[0]): # 遍历每个 batch
        preb = prebs[i, :, :] # 第 i 的 batch 的输出 (class_num,
        frames)
        preb_label = list() # 储存每个batch的输出序列, (batch_size,
        frames)
        # 贪婪算法求解预测结果
        for j in range(preb.shape[1]): # 遍历每个时间步中的输出字符概率分布
            preb_label.append(np.argmax(preb[:, j])) # 得到概率最大的字符索引
        no_repeat_blank_label = list() # 储存经过去重和去空白字符后的输出序列
        pre_c = preb_label[0] # 首先拿出第一个字符, 用于比较后面字符是否重复
        if pre_c != len(CHARS) - 1: # 若第一个字符不是空白字符, 则添加到输出序列中
            no_repeat_blank_label.append(pre_c)
        # 下面一个 for 循环是去重和去空白字符处理
        for c in preb_label:
            if (pre_c == c) or (c == len(CHARS) - 1):
                if c == (len(CHARS) - 1):
                    pre_c = c
                    continue
                no_repeat_blank_label.append(c)
                pre_c = c
            preb_labels.append(no_repeat_blank_label)
        # 计算准确率
        for i, label in enumerate(preb_labels):
            # 长度不匹配, 则计算为预测错误, 且跳过该次循环
            if len(label) != len(test_labels[i]):
                tp_error += 1
                continue
            # 输出序列与label完全相等, 预测正确, 反之预测错误
            if (np.asarray(test_labels[i]) == np.asarray(label)).all():
                tp += 1
            else:
                tp_error += 1

end_time = time.time() # 记录结束时间
t = end_time - start_time # 对测试集做测试的总运行时间
acc = tp * 1.0 / (tp + tp_error) # 准确率

```

```
return acc, tp, tp_error, t
```

2. 建立车牌数据通道

本实验的数据集采用的是经过处理的 CCPD2019 的车牌数据集，数据处理文件为 `./Dataset/deal_ccpd_data/` 文件夹中，有兴趣的同学可以自行研读。本实验提供了处理后的训练数据。数据生成函数代码在 `./Model_train/dataload` 文件中，具体如下：

```
import tensorflow as tf
import numpy as np
import cv2
import os

# 车牌字符列表，最后一个为空白字符。英语字符 O 和 I 在车牌字符中不会用到。
CHARS = ['京', '沪', '津', '渝', '冀', '晋', '蒙', '辽', '吉', '黑',
         '苏', '浙', '皖', '闽', '赣', '鲁', '豫', '鄂', '湘', '粤',
         '桂', '琼', '川', '贵', '云', '藏', '陕', '甘', '青', '宁',
         '新',
         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
         'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K',
         'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
         'W', 'X', 'Y', 'Z', '-']

# 生成车牌字符字典，用于生成 label 编码
CHARS_DICT = {char: i for i, char in enumerate(CHARS)}

# 对数据进行归一化
def preprocess(inputs, std=255., mean=0., expand_dims=None):
    inputs = (inputs - mean) / std
    if expand_dims is not None:
        np.expand_dims(inputs, expand_dims)
    return inputs

# 检查异常 label
def check(label):
    if label[2] != CHARS_DICT['D'] and label[2] != CHARS_DICT['F'] \
        and label[-1] != CHARS_DICT['D'] and label[-1] != CHARS_DICT['F']:
        print("Error label, please check it")
        return False
    else:
        return True

# 生成 tf.data.Dataset 实例用于训练
# 本实验提供的数据集，每张图片的命名即车牌字符。
def load_data(img_dir, lpr_len=7, img_size=(94, 24)):
    '''
    参数:
    1. img_dir: 数据集路径
    2. lpr_len: 车牌字符长度
    '''
```



```

3. img_size: 缩放后的车牌图片尺寸 (W, H)
返回:
1. imgs_dataset: 车牌图片数据, 为 tf.data.Dataset 实例
2. labels: 车牌标签数据, 为 tf.data.Dataset 实例
3. imgs_num: 数据集大小, int 型
'''

imgs_num = len(os.listdir(img_dir)) # 数据集大小
imgs = [] # 储存图片数据
labels = [] # 储存 label 数据
for i, img in enumerate(os.listdir(img_dir)):
    # 生成label数据
    label = []
    lpr_name, suffix = os.path.splitext(img) # 返回值分别为车牌号码和图片后缀
    for c in lpr_name: # 生成车牌编码
        label.append(CHARS_DICT[c])
    if len(label) == 8:
        if check(label) == False: # 检查异常车牌
            print(img)
            assert 0, "Error label!"
    # 生成image数据
    img = cv2.imread(os.path.join(img_dir, img)) # 读取图片, 默认为 BGR 格式, uint8型
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # 转换成 RGB 格式
    img = img.astype(np.float32) # 转换为 float32 型 (训练都是 float32 型)
    height, width, _ = img.shape
    if height != img_size[1] or width != img_size[0]:
        img = cv2.resize(img, img_size) # 缩放为所需要的尺寸
    imgs.append(img)
    labels.append(label)
# 生成 tf.data.Dataset 实例
imgs_dataset = tf.data.Dataset.from_tensor_slices(imgs)
labels = tf.data.Dataset.from_tensor_slices(labels)

return imgs_dataset, labels, imgs_num

```

3. 模型结构搭建

lpr 模型搭建代码在 `./Model_train/model/LPRNet.py` 文件中。该模型采用了一些特殊网络结构设计思想和算子的选择, 具体说明如下:

- **ResNet 思想:** 使得信息进行跳跃式连接, 降低了卷积层的信息减损。关于 Resnet 思想推荐参考原文: <https://arxiv.org/pdf/1512.03385.pdf>。
- **深度可分离卷积 (DepthwiseConv2D):** 深度可分离卷积可以在在精度降低不明显的前提下, 有效降低参数数量, 提高模型推理速度。深度可分离卷积的思想推荐参考知乎上的一篇文章: <https://zhuanlan.zhihu.com/p/92134485>。
- **Dropout:** 之前的实验也使用过该算子, 目的是为了防止模型出现过拟合, 该方法简单有效。

模型搭建代码如下:

```
import tensorflow as tf
import tensorflow.keras.layers as ly

# 网络中的小模块定义
class small_basic_block(tf.keras.Model):
    def __init__(self, filters):
        super().__init__()
        self.block1 = tf.keras.Sequential([
            ly.Conv2D(filters=filters // 2, kernel_size=3, strides=1, padding='same'),
            ly.BatchNormalization(),
            ly.ReLU(),
            ly.DepthwiseConv2D(kernel_size=1, strides=1, padding='same'),
            ly.BatchNormalization(),
            ly.ReLU(),
            ly.DepthwiseConv2D(kernel_size=1, strides=1, padding='same'),
            ly.BatchNormalization(),
            ly.ReLU(),
        ])
        self.block2 = tf.keras.Sequential([
            ly.Conv2D(filters=filters, kernel_size=3, strides=1),
            ly.BatchNormalization(),
            ly.ReLU(),
        ])
    def call(self, inputs):
        x = ly.Concatenate(axis=-1)([inputs, self.block1(inputs)])
        output = self.block2(x)

        return output

# lpr 网络构建
class LPRNet(tf.keras.Model):
    def __init__(self, lpr_len, class_num, dropout_rate=0.5):
        super().__init__()
        self.lpr_len = lpr_len
        self.class_num = class_num
        self.dropout_rate = dropout_rate

        self.net = tf.keras.Sequential([
            ly.Conv2D(filters=32, kernel_size=3, strides=1, name='input'),
            ly.BatchNormalization(),
            ly.ReLU(),
            small_basic_block(filters=64),
            small_basic_block(filters=64),
            ly.Conv2D(filters=64, kernel_size=3, strides=(1, 2)),
            ly.Dropout(dropout_rate),
            ly.BatchNormalization(),
            ly.ReLU(),
        ])
```

```
        ly.AveragePooling2D(pool_size=3, strides=2)
    ])

    self.container = tf.keras.Sequential([
        ly.Conv2D(filters=class_num, kernel_size=1, strides=1, name='
        container')
    ])

    def call(self, inputs):
        x = self.net(inputs)
        x = self.container(x)
        x = tf.transpose(x, [0, 3, 1, 2])
        output = tf.reduce_mean(x, axis=2)

        return output
```

使用 **Netron** 神经网络可视化工具，我们可以看到该模型网络结构如下所示：

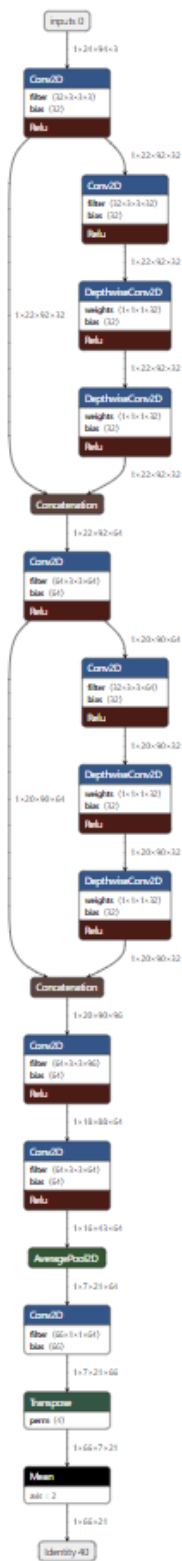


图 3.1: 模型结构

3.2 AI 模型训练

1. 在指定路径打开 Jupyter notebook, 并打开 Model_train/lpr_train.ipynb 模型训练文件

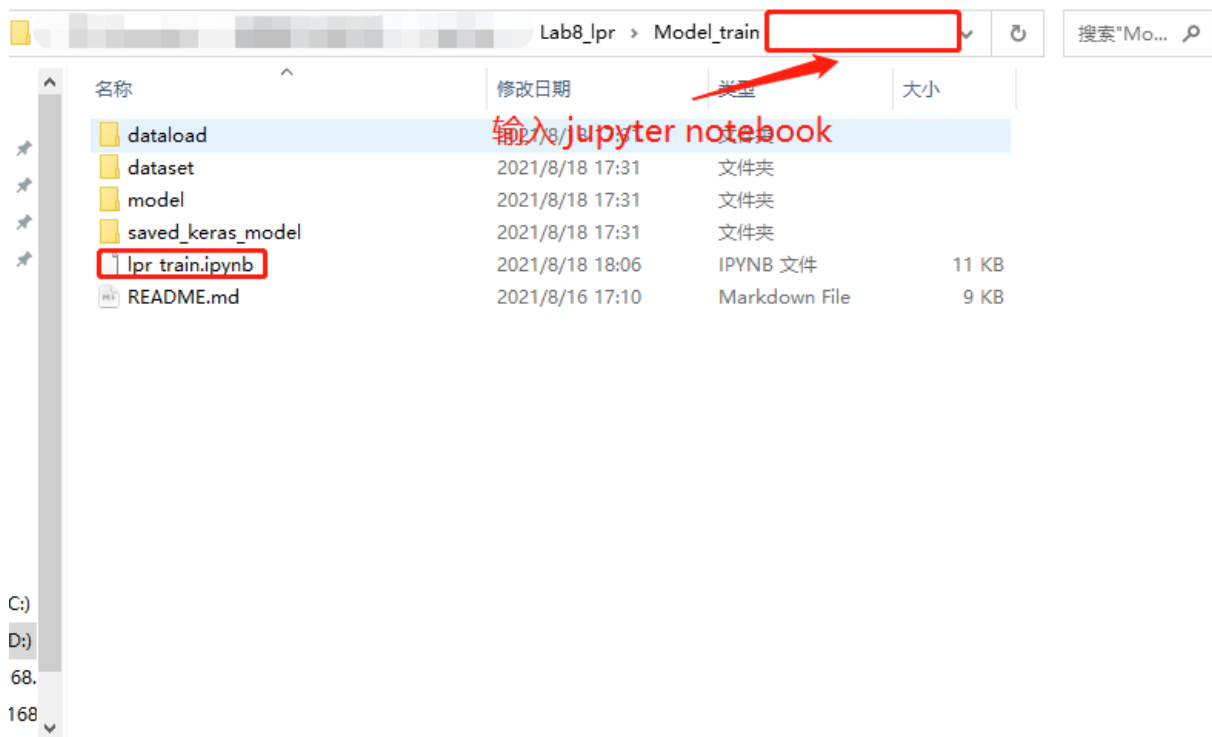


图 3.2: 模型训练步骤 1



图 3.3: 模型训练步骤 1.1

本章节代码全部来自 lpr_train.ipynb 训练代码文件

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

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # 忽略2级以下警告
os.environ['CUDA_VISIBLE_DEVICES'] = '0' # 使用标号为0的 GPU 训练

from dataload import * # 导入数据处理文件的所有代码
from model import * # 导入模型定义文件的所有代码
from metric import * # 导入评价函数定义文件的所有代码
import tensorflow as tf
import numpy as np
import time

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR) # 忽略训练中的警告
```

说明：本实验中不是使用 `model.fit` 来拟合模型，在训练由于 `BatchNormalization` 和 `ReLU` 算子没有参数可以进行反向传播，这会导致打印出许多警告。但这些警告对模型训练没有任何影响，故在这里使用 `tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)` 来忽略这些警告，使得模型训练打印出来的信息更简洁有用。

因为训练数据集很大，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，只是速度很慢。

3. 超参数赋值

```
epochs = 300 # 模型训练循环次数
train_img_dir = '../Dataset/dataset/train' # 训练集路径
test_img_dir = '../Dataset/dataset/eval' # 测试集路径
initial_lr = 0.001 # 初始学习率
dropout_rate = 0.5 # dropout 丢弃率
lpr_len = 7 # 车牌字符长度
train_batch_size = 128 # 训练时的批量
test_batch_size = 128 # 测试时的批量
saved_model_folder = '../Models/keras_model' # keras 模型保存路径
pretrained_model = '' # 预训练模型路径，该参数赋值可有可无
```

4. 数据通道生成

```
# 加载训练数据集
tr_imgs, tr_labels, train_imgs_num = load_data(img_dir=train_img_dir, lpr_len=lpr_len)
tr_imgs = tr_imgs.map(preprocess, num_parallel_calls=4)
train_dataset = tf.data.Dataset.zip((tr_imgs, tr_labels)).shuffle(train_imgs_num).batch(train_batch_size).prefetch(tf.data.experimental.AUTOTUNE).cache()

# 加载测试数据集
te_imgs, te_labels, test_imgs_num = load_data(img_dir=test_img_dir, lpr_len=lpr_len)
te_imgs = te_imgs.map(preprocess, num_parallel_calls=4)
test_dataset = tf.data.Dataset.zip((te_imgs, te_labels)).batch(test_batch_size).prefetch(tf.data.experimental.AUTOTUNE).cache()
```

API 部分参数说明:

- `tr_imgs.map`: 将 `tr_imgs` 中的数据经过 `preprocess` 函数之后返回, 建立 `num_parallel_calls=4` 个进程同时处理数据, 该参数请同学们根据自己的电脑配置自行选择
 - `tf.data.Dataset.zip`: 将图片数据和标签数据打包在一起, 返回完整训练集
 - 方法 `shuffle`: 打乱数据集顺序, `train_imgs_num` 为训练集大小
 - 方法 `batch`: 批处理, 参数为 `batch_size`
 - 方法 `prefetch`: 在下一个 `batch` 训练前提前拉取数据集, 提高训练速度
 - 方法 `cache`: 提前将拉取到的数据集放入内存中, 提高训练速度
- 详细可查看 Tensorflow 官方 API 说明文档: https://tensorflow.google.cn/versions/r2.5/api_docs/python/tf/data/Dataset。

5. 模型训练

```
# 训练模型
# 创建保存模型的文件夹
if not os.path.exists(saved_model_folder):
    os.mkdir(saved_model_folder)

# 实例化模型
lprnet = LPRNet(lpr_len=lpr_len, class_num=len(CHARS), dropout_rate=dropout_rate)
print("***** Successful to build network! *****\n")

# 加载预训练模型
if pretrained_model:
    lprnet.load_weights(pretrained_model)
    print("***** Successful to load pretrained model! *****")

# 优化器使用 Adam
optimizer = tf.keras.optimizers.Adam(learning_rate=initial_lr)

# 模型训练
top_acc = 0. # 用来记录训练过程中在测试集上最高的准确率, 我们保存的是准确率最高的模型
for cur_epoch in range(1, epochs + 1):
    batch = 0
    for batch_index, (train_imgs, train_labels) in enumerate(train_dataset):
        start_time = time.time() # 训练1个batch的初始时间
        with tf.GradientTape() as tape:
            train_logits = lprnet(train_imgs) # shape:[N,66,21] 21为 frames, 66
            # 为 class_num
            train_labels = tf.cast(train_labels, tf.int32) # shape:[N,7]
            train_logits = tf.transpose(train_logits, [2, 0, 1]) # shape:[21,N,66]
            logits_shape = train_logits.shape # 返回 train_logits 的形状大小数值
```

```

logit_length = tf.fill([logits_shape[1]], logits_shape[0]) #shape:(N
),用frames 填充
label_length = tf.fill([logits_shape[1]], lpr_len) # shape:(N),用7
填充
loss = tf.nn.ctc_loss(labels=train_labels,
                      logits=train_logits,
                      label_length=label_length,
                      logit_length=logit_length,
                      logits_time_major=True,
                      blank_index=len(CHARS) - 1)

loss = tf.reduce_mean(loss) # 一个batch中, 去loss为平均值
grads = tape.gradient(loss, lprnet.variables) # 计算loss关于lprnet.
variables的导数
optimizer.apply_gradients(grads_and_vars=zip(grads, lprnet.variables)) #
更新权重
end_time = time.time()
batch = batch + int(np.shape(train_imgs)[0])
print('\r' + "Epoch {0}/{1} || ".format(cur_epoch, epochs) # 打印 epoch
情况
      + "Batch {0}/{1} || ".format(batch, train_imgs_num) # 打印
      batch 情况
      + "Loss:{0} || ".format(loss) # 打印 loss
      + "A Batch time:{0:.4f}s || ".format(end_time - start_time) #
      打印一个 batch 的训练时间
      + "Learning rate:{0:.8f} || ".format(optimizer.lr.numpy().item
      ()), end=' '*20) # 打印当前学习率
acc, tp, tp_error, t = metric(test_dataset, lprnet) # 做测试
print("\n***** Prediction {0}/{1} || Acc:{2:.2f}% *****".format(tp, tp +
tp_error, acc*100))
print("***** Test speed: {}s 1/{0} *****".format(t / (tp + tp_error), tp
+ tp_error))

# 保存测试准确率最高的模型, 这里不是使用model.fit训练的, 则保存格式为 pb 格
式
if acc >= top_acc:
    top_acc = acc
    lprnet.save(saved_model_folder, save_format='tf')

```

本实验模型训练采用 TensorFlow 的自动求导机制, 跟之前 `model.fit` 看起来不同, 但本质上是一样的。在即时执行模式下, TensorFlow 引入了 `tf.GradientTape()` 这个“求导记录器”来实现自动求导。细节详见 TensorFlow API 官方文档: https://tensorflow.google.cn/versions/r2.5/api_docs/python/tf/GradientTape。

6. 模型文件转成 RT-AK 部署所支持的格式 (.tflite 文件)

```

# 将 keras 模型转为 tflite
cvtmodel = tf.keras.models.load_model(saved_model_folder)
converter = tf.lite.TFLiteConverter.from_keras_model(cvtmodel)
tflite_model = converter.convert()

```



```
with open('lprnet' + '{}'.format(np.around(top_acc * 100)) + '.tflite', "wb") as f:
    f.write(tflite_model)
print("\n ***** Successful to convert tflite model! ***** \n")
```

3.3 模型信息

已训练好的模型文件在文件夹 `Models` 中，文件夹 `keras_model` 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：

- 本实验训练的车牌识别模型结构简单，总参数量为 164,034，`tflite` 模型文件大小为 661 KB，适合部署至 K210
- `keras` 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 * 66 * 21，类型为 `float32`，66 为字符总个数 (包括一个空白字符)，21 为时间步长

3.4 模型部署

在 `RT-AK/rt_ai_tools` 路下打开 Windows 终端

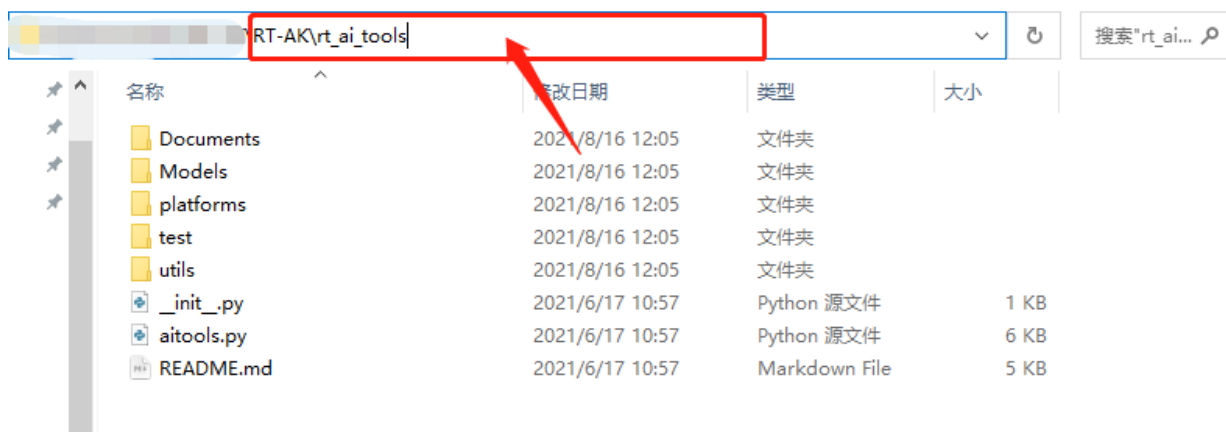


图 3.4: RT-AK 使用

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 --inference_type=uint8 --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`，`--inference_type` 是模型量化后数据类型，`--dataset` 是模型量化所需要用到的数据集，一般 200 张图片左右。

```
D:\RTAK-tools\RT-AK\RT-AK\rt_ai_tools>python aitools.py --project=D:\RT_AK_project\k210\k210_lpr\project --model=C:\User
s\Admin\Desktop\lprnet84.0.tflite --model_name=lprnet --inference_type=uint8 --dataset=D:\RT_AK_project\k210\k210_lpr\lp
r_quantitative_data --platform=k210 --embed_gcc=D:\RTAK-tools\unpack-riscv-none-embed-gcc-8.3.0-1.2\bin
2021-08-19 09:52:56 git_plugin [INFO]: git version 2.31.1.windows.1
fatal: not a git repository (or any of the parent directories): .git
fatal: not a git repository (or any of the parent directories): .git
fatal: not a git repository (or any of the parent directories): .git
2021-08-19 09:52:57 git_plugin [INFO]: Install plugin_k210 successfully...
2021-08-19 09:52:57 is_support_model_type [INFO]: The model is lprnet84.0.tflite
2021-08-19 09:52:57 set_env [INFO]: ncc version 0.2...
1. Import graph...
2. Optimize Pass 1...
3. Optimize Pass 2...
4. Quantize...
4.1. Add quantization checkpoints...
4.2. Get activation ranges...
Plan buffers...
Run calibration...
[=====] 100% 62.635s
4.5. Quantize graph...
5. Lowering...
6. Optimize Pass 3...
7. Generate code...
Plan buffers...
Emit code...
Working memory usage: 1382400 B

SUMMARY
INPUTS
0      Input_0 1x3x24x94
OUTPUTS
0      Identity 1x66x21

2021-08-19 09:56:00 convert_kmodel [INFO]: Convert model to kmodel successfully...
2021-08-19 09:56:00 hex_read_model [INFO]: Save hex kmodel successfully...
2021-08-19 09:56:00 rt_ai_model_gen [INFO]: Generate rt_ai_lprnet_model.h successfully...
2021-08-19 09:56:00 load_rt_ai_example [INFO]: Generate rt_ai_facelandmark_model.c successfully...
2021-08-19 09:56:00 set_gcc_path [INFO]: Set GNU Compiler Toolchain successfully...
2021-08-19 09:56:00 enable_platform [INFO]: No need to enable RT-AK Lib again...
2021-08-19 09:56:00 load_rt_ai_lib [INFO]: RT-AK Libs loading successfully...
```

图 3.5: RT-AK 部署

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/lprnet_kmodel.c	kmodel 的十六进制储存
applications/rt_ai_lprnet_model.c	与目标平台相关的信息
applications/rt_ai_lprnet_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
lprnet.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“3.5 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

3.5 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 Lab8_LPR\Applications。

3.5.1 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 kpu 时钟初始化 `sysctl_clock_enable(SYSCTL_CLOCK_AI)`

1.2 LCD 初始化 `lcd_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. 车牌识别业务逻辑层：

3.1 对于每个时间步，对长度为 66 的向量求得对应的车牌字符的最大值索引，得到预测字符序列

3.2 对预测字符序列进行去重去空白字符处理：

3.2.1 位置相邻的重复字符只留一个

3.2.2 去除空白字符

3.3 得到输出序列

3.4 解码得到输出车牌字符

3.5.2 核心代码说明

核心代码详见 Lab8_lpr\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 model 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
greedy_solution(output, lp_label); // 获取车牌字符索引,赋值给 lp_label

// 车牌字符连接，便于 LCD 打印
strcpy(license_plate, " ");
for(int i=0; i<7; i++){
    strcat(license_plate, CHARS[lp_label[i]]);}

printf("LPRNet prediction: %s\n", license_plate);; // 在终端打印出预测结果

/* LCD display result */
lcd_draw_string(320/2 - 16, 20, license_plate, GREEN);
lcd_draw_picture(320/2 - 16, 240/2 - 16, 94, 24, LCD_DISPLAY0);

```

处理模型输出得到车牌预测字符的贪婪算法如下：

```

// 贪心算法对模型输出结果进行后处理，得到车牌字符索引
static int *greedy_solution(float *logit, int *lp){

    int lpchars_num = 66; // 车牌字符总数
    int frames = 21; // 时间步长度
    int pre_label[frames]; // 储存长度为 frames 的车牌预测字符索引，包括重复字符和
        blank
    // 对每个时间步，求出最大值索引
    for(int i=0; i<frames; i++){
        int pred = 0;
        for(int j=0; j<lpchars_num; j++){
            if(logit[i + j*frames] > logit[i + pred*frames]){
                pred = j;
            }
        }
        pre_label[i] = pred;
    }

    int no_repeat_lpr[frames]; // 储存去重和去 blank 之后的车牌预测字符索引
    for(int i=0; i<frames; i++){
        no_repeat_lpr[i] = -1;
    }
}

```

```
int preb = pre_label[0];
int blank_index = (lpchars_num - 1); // blank 字符索引
// 字符去重和去 blank 字符步骤
int q = 0;
if(preb != (lpchars_num - 1)){
    no_repeat_lpr[0] = preb;
    q += 1;
}

for(int k=0; k<frames; k++){
    if((pre_label[k] == preb) || (pre_label[k] == blank_index)){
        if(pre_label[k] == blank_index){
            preb = pre_label[k];
        }
        continue;
    }
    no_repeat_lpr[q] = pre_label[k];
    q += 1;
    preb = pre_label[k];
}

for(int m=0; m<7; m++){
    lp[m] = no_repeat_lpr[m];
}

return lp;
}
```

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。

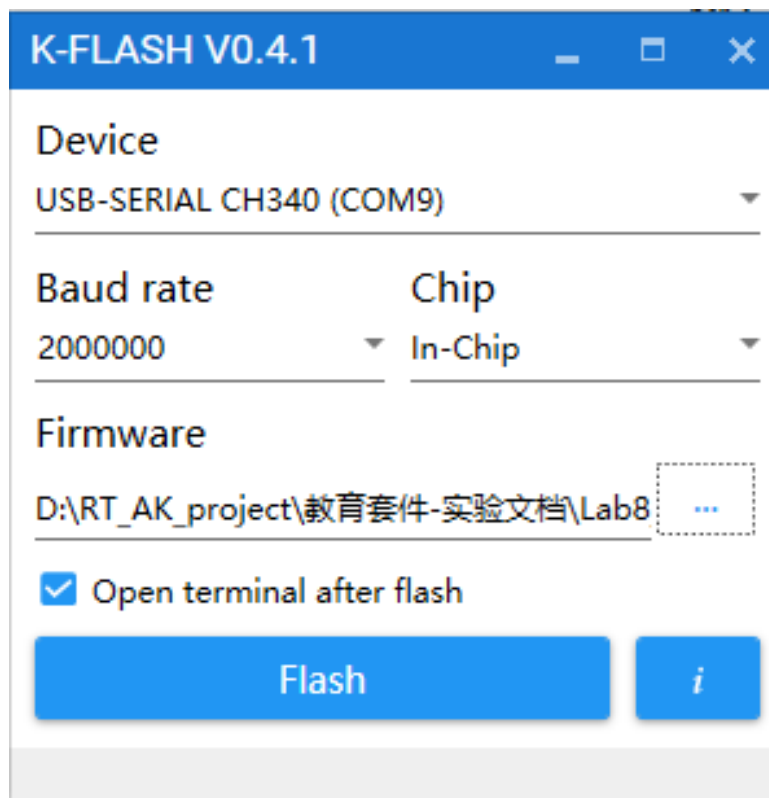
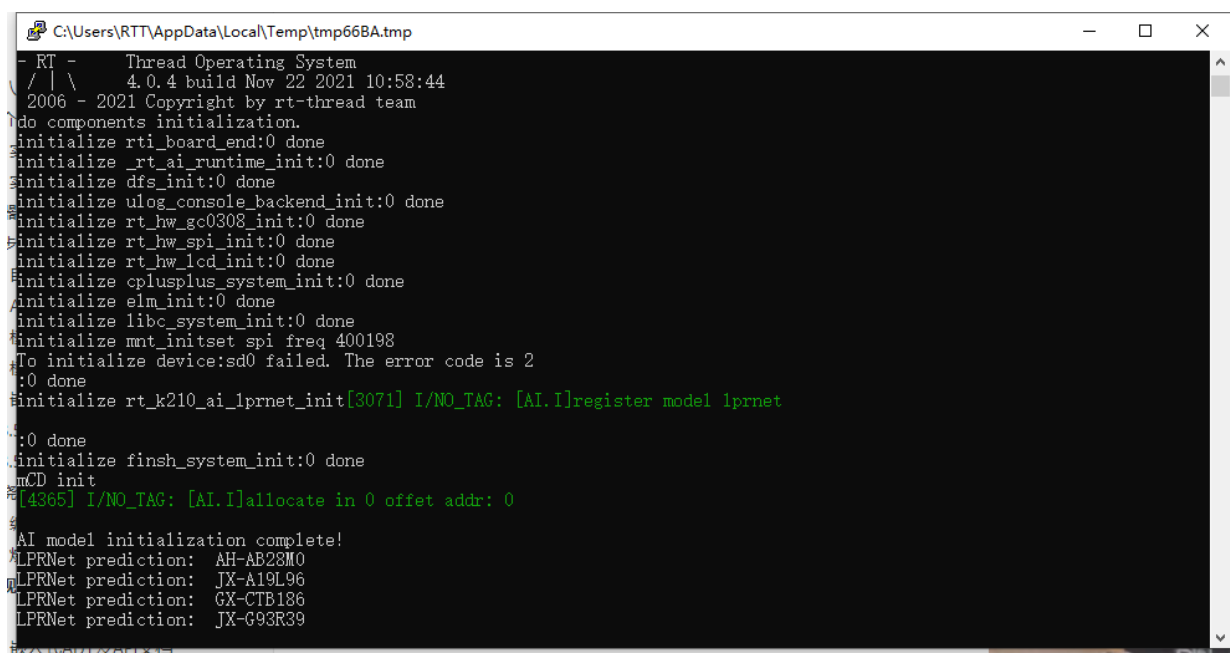


图 4.1: K-Flash

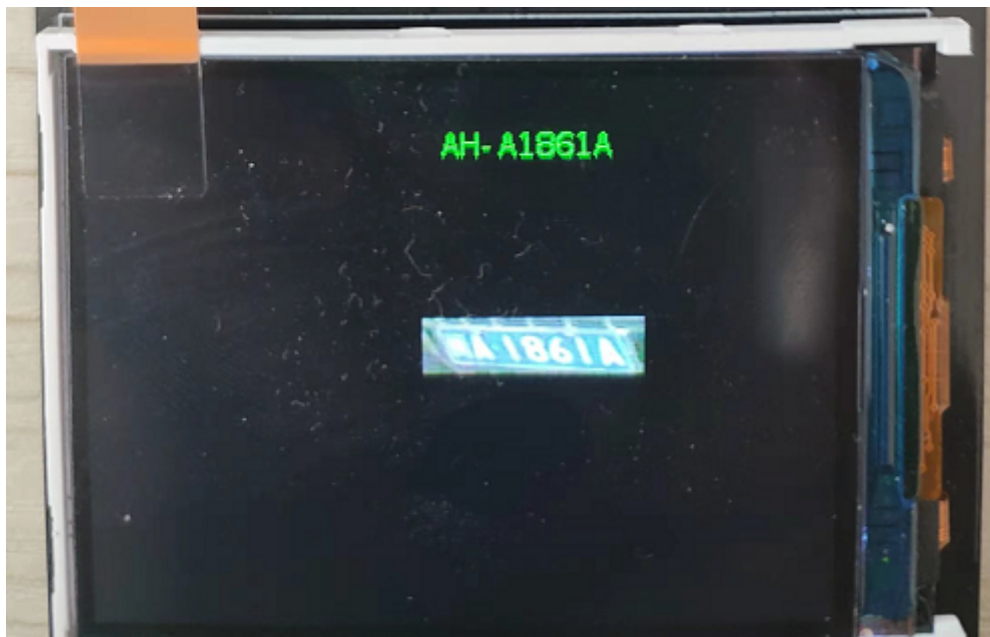
第 5 章

实验现象

如果编译 & 烧写无误，K-Flash 会自动打开 Windows 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。其中 `run_time_cost` 是模型推理一帧的时间，单位是 `ms`。并且会在 `k210 LCD` 打印模型输入图片以及模型预测结果。如下图，预测结果为 `AH-A25Y86`，`AH` 为安徽简称（终端以及 `LCD` 上不能打印中文，而省份简称有些会重复，故如此设置输出）。



```
C:\Users\RTT\AppData\Local\Temp\tmp668A.tmp
- RT -      Thread Operating System
/ | \      4.0.4 build Nov 22 2021 10:58:44
2006 - 2021 Copyright by rt-thread team
do components initialization.
initialize rti_board_end:0 done
initialize rt_ai_runtime_init:0 done
initialize dfs_init:0 done
initialize ulog_console_backend_init:0 done
initialize rt_hw_gc0308_init:0 done
initialize rt_hw_spi_init:0 done
initialize rt_hw_lcd_init:0 done
initialize cplusplus_system_init:0 done
initialize elm_init:0 done
initialize libc_system_init:0 done
initialize mnt_initset spi freq 400198
To initialize device:sd0 failed. The error code is 2
:0 done
initialize rt_k210_ai_lprnet_init[3071] I/NO_TAG: [AI.I]register model lprnet
:0 done
initialize finsh_system_init:0 done
LCD init
[4365] I/NO_TAG: [AI.I]allocate in 0 offset addr: 0
AI model initialization complete!
LPRNet prediction: AH-AB28M0
LPRNet prediction: JX-A19L96
LPRNet prediction: GX-CTB186
LPRNet prediction: JX-G93R39
```

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);
```

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

描述: 模型推理计算

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

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

描述: 获取模型运行的结果, 结果获取后.

在工程文件中的`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  
*/
```