# 项目8 TensorFlow Lite

项目描述

TensorFlow生态系统有着丰富的工具链，TensorFlow Serving是使用广泛的高性能的服务器端部署平台，TensorFlow.js支持使用JavaScript在浏览器端部署，TensorFlow Lite加速了端侧机器学习的发展，它支持Android、IOS、嵌入式设备、以及极小的MCU设备。全球超过40亿设备部署了TensorFlow Lite，谷歌、Uber、网易、爱奇艺、腾讯等公司的应用都使用了TensorFlow Lite。

本项目为一个图像识别项目，基于TensorFlow Lite，优化MobileNet模型并在Android手机上实现识别四种花的种类，掌握如何通过相应工具将模型转化成适合手机设备的格式，并在Android应用中部署转换后的模型。

项目目标

知识目标

* 了解TensorFlow Lite的发展历史
* 了解TTensorFlow Lite的应用
* 掌握TensorFlow Lite的整体架构
* 掌握TensorFlow Lite转换器作用
* 掌握FlatBuffers格式
* 掌握TensorFlow Lite解释执行器特点及工作过程

技能目标

* 能通过相应工具将模型转化
* 能在Android应用中部署转换后的模型
* 能熟练Android Studio
* 能配置build.gradle构建项目
* 能熟练掌握迁移学习改造模型，开发相应AI应用

## 8.1 认识TensorFlow Lite

2015 年底Google 开源了端到端的机器学习开源框架 TensorFlow，它既支持大规模的模型训练，也支持各种环境的部署，包括服务器和移动端的部署，支持各种语言，包括 Python，C++，Java，Swift 甚至 Javascript。而近年来移动化浪潮和交互方式的改变，使得机器学习技术开发也在朝着轻量化的端侧发展，TensorFlow 团队又在 2017 年底上线了 TensorFlow Lite，一个轻量、快速、兼容度高的专门针对移动式应用场景的深度学习工具，把移动端及 IoT 设备端的深度学习技术的门槛再次大大降低。

### 8.1.1 TensorFlow Lite发展历史

TFLite是在边缘设备上运行TensorFlow模型推理的官方框架，它跨平台运行，包括Android、iOS以及基于Linux的IoT设备和微控制器。

伴随移动和 IoT 设备的普及，世界以超乎想象的方式存在被连接的可能，如今已有超过 32 亿的手机用户和 70 亿的联网 IoT 设备。而随着手机成本不断降低，并且随着微控制器（MCU）和微机电系统（MEMs）的发展，高性能低功耗的芯片使得“万物”智能具有了可能性。Google开始了TF Mobile项目，尝试简化TensorFlow并在移动设备上运行，它是一个缩减版的TensorFlow，简化了算子集，也缩小了运行库。

TFMini是Google内部用于计算机视觉场景的解决方案，它提供了一些转换工具压缩模型，进行算子融合并生成代码。它将模型嵌入到二进制文件中，这样就可以在设备上运行和部署模型。TFMini针对移动设备做了很多优化，但在把模型嵌入到实际的二进制文件中时兼容性存在较大挑战，因此TFMini并没有成为通用的解决方案。

基于TF Mobile的经验，也继承了TFMini和内部其他类似项目的很多优秀工作，Google设计了TFLite：

1) 更轻量。TensorFlow Lite 二进制文件的大小约为 1 MB（针对 32 位 ARM build）；如果仅使用支持常见图像分类模型（InceptionV3 和 MobileNet）所需的运算符，TensorFlow Lite 二进制文件的大小不到 300 KB。

2) 特别为各种端侧设备优化的算子库。

3) 能够利用各种硬件加速。

### 8.1.2 TensorFlow Lite的应用

全球有超过40亿的设备上部署着TFLite，例如Google Assistant，Google Photos等、Uber、Airbnb、以及国内的许多大公司如网易、爱奇艺和WPS等，都在使用TFLite。端侧机器学习在图像、文本和语音等方面都有非常广泛应用。

TensorFlow Lite能解决的问题越来越多元化，这带来了应用的大量繁荣。在移动应用方面，网易使用TFLite做OCR处理，爱奇艺使用TFLite来进行视频中的AR效果，而WPS用它来做一系列文字处理。图像和视频方面广泛应用，比如Google Photos，Google Arts & Culture。

离线语音识别方面有很多突破，比如Google Assistant宣布了完全基于神经网络的移动端语音识别，效果和服务器端十分接近，服务器模型需要2 GB大小，而手机端只需要80 MB。端侧语音识别非常有挑战，它的进展代表着端侧机器学习时代的逐步到来.一方面依赖于算法的提高，另外一方面TFLite框架的高性能和模型优化工具也起到了很重要的作用。Google Pixel 4手机上发布了Live Caption，自动把视频和对话中的语言转化为文字，大大提高了有听力障碍人群的体验。另外一方面，模型越来越小，无处不再，Google Assistant的语音功能部署在非常多元的设备上，比如手机端、手表、车载和智能音箱上，全球超过10亿设备。

TFLite可以支持微控制器(MCU)，可以应用于IoT领域，MCU是单一芯片的小型计算机，没有操作系统，只有内存，也许内存只有几十KB。TFLite发布了若干MCU上可运行的模型，比如识别若干关键词的语音识别模型和简单的姿态检测模型，模型大小都只有20 KB左右，基于此可构建更智能的IoT应用，例如出门问问智能音箱使用TFLite来做热词唤醒，科沃斯扫地机器人使用TFLite在室内避开障碍物。如何让用户用更少的时间进行清扫工作是科沃斯不断追求的目标，它使用了机器视觉的帮助，可以识别这个过程中的一些障碍物，选择了用 TensorFlow Lite 部署深度神经网络，将推理速度提高了 30%，提高了用户的体验。

TFLite也非常适合工业物联智能设备的开发，因为它很好地支持如树莓派及其他基于Linux SoC的工业自动化系统.创新奇智应用TFLite开发智能质检一体机、智能读码机等产品，应用到服装厂质检等场景

## 8.2 TensorFlow Lite体系结构

TensorFlow Lite 是一组工具，可帮助开发者在移动设备、嵌入式设备和 IoT 设备上运行 TensorFlow 模型。它支持设备端机器学习推断，延迟较低，并且二进制文件很小。

### 8.2.1 TensorFlow Lite整体架构

TensorFlow Lite 包括两个主要组件：

* TensorFlow Lite 解释器(Interpreter)
* TensorFlow Lite 转换器(Converter)
* 算子库(Op kernels)
* 硬件加速代理(Hardware accelerator delegate)

TFLite采用更小的模型格式，并提供了方便的模型转换器，可将 TensorFlow 模型转换为方便解释器使用的格式，并可引入优化以减小二进制文件的大小和提高性能。比如SavedModel或GraphDef格式的TensorFlow模型，转换成TFLite专用的模型文件格式，在此过程中会进行算子融合和模型优化，以压缩模型，提高性能。

TensorFlow Lite采用更小的解释器，可在手机、嵌入式 Linux 设备和微控制器等很多不同类型的硬件上运行经过专门优化的模型。安卓应用只需 1 兆左右的运行环境，在 MCU 上甚至可以小于 100KB。

TFLite算子库目前有130个左右，它与TensorFlow的核心算子库略有不同，并做了移动设备相关的优化。

在硬件加速层面，对于 CPU 利用了 ARM 的 NEON 指令集做了大量的优化。同时，Lite 还可以利用手机上的加速器，比如 GPU 或者 DSP等。另外，最新的安卓系统提供了 Android 神经网络 API（Android NN API)，让硬件厂商可以扩展支持这样的接口。

图8-1展示了在TensorFlow 2.0中TFLite模型转换过程，用户在自己的工作台中使用TensorFlow API构造TensorFlow模型，然后使用TFLite模型转换器转换成TFLite文件格式(FlatBuffers格式)。在设备端，TFLite解释器接受TFLite模型，调用不同的硬件加速器比如GPU进行执行。

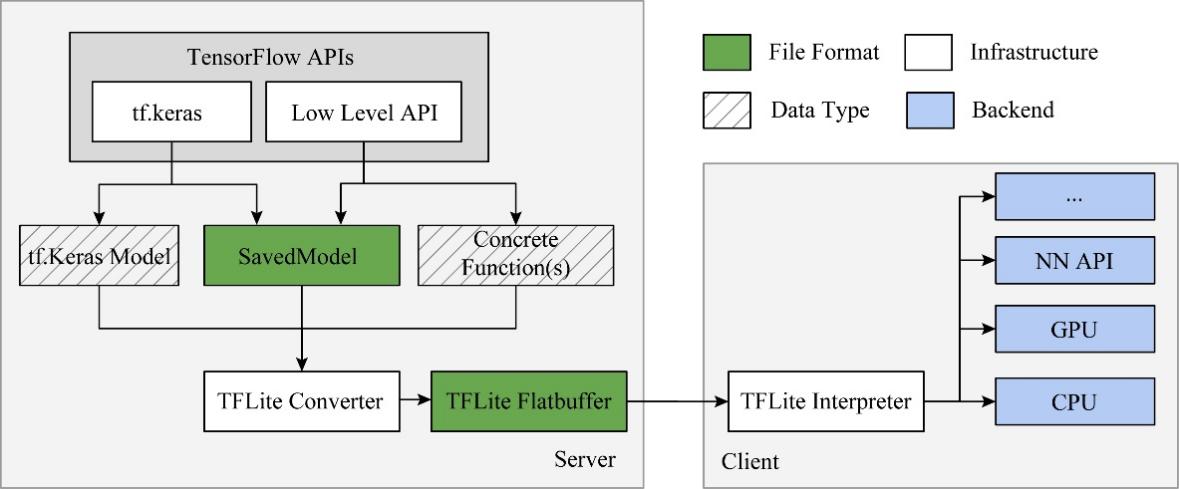


图8-1 TFLite模型转换过程

### 8.2.2 TensorFlow Lite转换器

TFLite转换器可以接受不同形式的模型，包括Keras Model和SavedModel，开发者可以用tf.Keras或者低层级的TensorFlow API来构造TensorFlow模型，然后使用Python API或者命令行的方式调用转换器。例如：

1. Python API

调用tf.lite.TFLiteConverter，可用TFLiteConverter.from\_saved\_model()，或TFLiteConverter.from\_keras\_model()；

1. 命令行

tflite\_convert --saved\_model\_dir=width=5,height=17,dpi=110tmpwidth=5,height=17,dpi=110mobilenet\_saved\_model --output\_file=width=5,height=17,dpi=110tmpwidth=5,height=17,dpi=110mobilenet.tflite

转换器做了以下优化工作：

1. 算子优化和常见的编译优化，比如算子融合、常数折叠或无用代码删除等。TFLite实现了一组优化的算子内核，转化成这些算子能在移动设备上实现性能大幅度提升。
2. 量化的原生支持。在模型转换过程中使用训练后量化非常简单，不需要改变模型，最少情况只需多加一行代码，设置converter.optimizations=[tf.lite.Optimize.DEFAULT]。

### 8.2.3 FlatBuffers格式

TFLite模型文件格式采用FlatBuffers，更注重考虑实时性，内存高效，这在内存有限的移动环境中是极为关键的。它支持将文件映射到内存中，然后直接进行读取和解释，不需要额外解析。将其映射到干净的内存页上，减少了内存碎片化。

TFLite代码中schema.fbs文件使用FlatBuffers定义了TFLite模型文件格式，关键样例代码如图8-2所示。TFLite模型文件是一个层次的结构：

1. TFLite模型由子图构成，同时包括用到的算子库和共享的内存缓冲区。
2. 张量用于存储模型权重，或者计算节点的输入和输出，它引用Model的内存缓冲区的一片区域，提高内存效率。
3. 每个算子实现有一个OperatorCode，它可以是内置的算子，也可以是自定制算子，有一个名字。
4. 每个模型的计算节点包含用到的算子索引，以及输入输出用到的Tensor索引。
5. 每个子图包含一系列的计算节点、多个张量，以及子图本身的输入和输出。

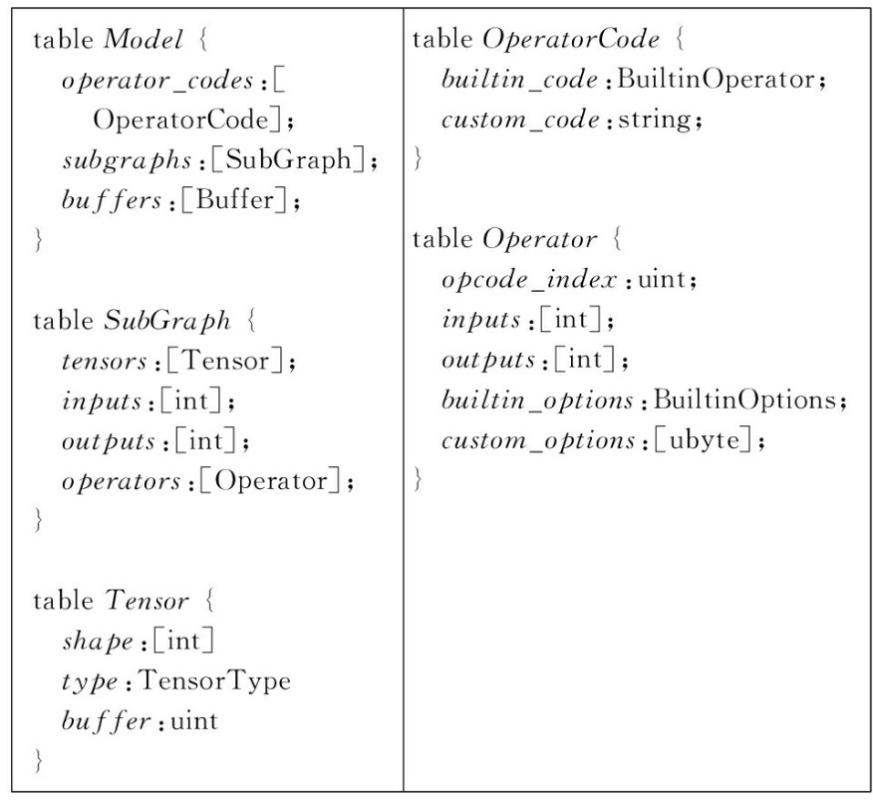


图8-2 TFLite schema.fbs样例代码

### 8.2.4 TensorFlow Lite解释执行器

TFLite解释执行器针对移动设备从头开始构建，具有以下特点:

1. 轻量级

在32 b安卓平台下，编译核心运行时得到的库大小只有100 KB左右，如果加上所有TFLite的标准算子，编译后得到的库大小是1 MB左右。它依赖的组件较少，力求实现不依赖任何其他组件。

1. 快速启动

既能够将模型直接映射到内存中，同时又有一个静态执行计划，在转换过程中基本上可以提前直接映射出将要执行的节点序列。采取了简单的调度方式，算子之间没有并行执行，而算子内部可以多线程执行以提高效率。

1. 内存高效

在内存规划方面，采取了静态内存分配。当运行模型时，每个算子会执行prepare函数，它们会分配一个单一的内存块，而这些张量会被整合到这个大的连续内存块中，不同张量之间甚至可以复用内存以减少内存分配.

使用解释执行器通常需要包含四步：

1. 加载模型

将TFLite模型加载到内存中，该内存包含模型的执行图。

1. 转换数据

模型的原始输入数据通常与所期望的输入数据格式不匹配.例如，可能需要调整图像大小，或更改图像格式，以兼容模型。

1. 运行模型推理

使用TFLite API执行模型推理。

1. 解释输出

解释输出模型推理结果，比如模型可能只返回概率列表，而我们需要将概率映射到相关类别，并将其呈现给最终用户。

TFLite提供了多种语言的API，正式支持的有Java，C++和Python，实验性的包括C，Object C，C#和Swift。可以从头自己编译TFLite，也可以利用已编译好的库，Android开发者可以使用JCenter Bintray的TFLite AAR，而iOS开发者可通过CocoaPods在iOS系统上获取。

## 8.3 任务1：TensorFlow Lite开发工作流程

使用 TensorFlow Lite 的工作流程包括如下步骤，如图8-3：

1. 选择模型

可以使用自己的 TensorFlow 模型、在线查找模型，或者从的TensorFlow预训练模型中选择一个模型直接使用或重新训练。

1. 转换模型

如果使用的是自定义模型，请使用TensorFlow Lite转换器将模型转换为TensorFlow Lite 格式。

1. 部署到设备

使用TensorFlow Lite解释器（提供多种语言的 API）在设备端运行模型。

1. 优化模型

使用模型优化工具包缩减模型的大小并提高其效率，同时最大限度地降低对准确率的影响。



图8-3 TensorFlow Lite 的工作流程

### 8.3.1 选择模型

TensorFlow Lite 允许在移动端（mobile）、嵌入式（embeded）和物联网（IoT）设备上运行 TensorFlow 模型。TensorFlow 模型是一种数据结构，这种数据结构包含了在解决一个特定问题时，训练得到的机器学习网络的逻辑和知识。

有多种方式可以获得 TensorFlow 模型，从使用预训练模型（pre-trained models）到训练自己的模型。为了在 TensorFlow Lite 中使用模型，模型必须转换成一种特殊格式。TensorFlow Lite 提供了转换 、运行 TensorFlow 模型所需的所有工具。

为了避免重复开发，Google将训练好的模型放在TensorFlow Hub，如图8-4所示。开发人员可以复用这些已经训练好且经过充分认证的模型，节省训练时间和计算资源。这些训练好的模型即可以直接部署，也可以用于迁移学习。

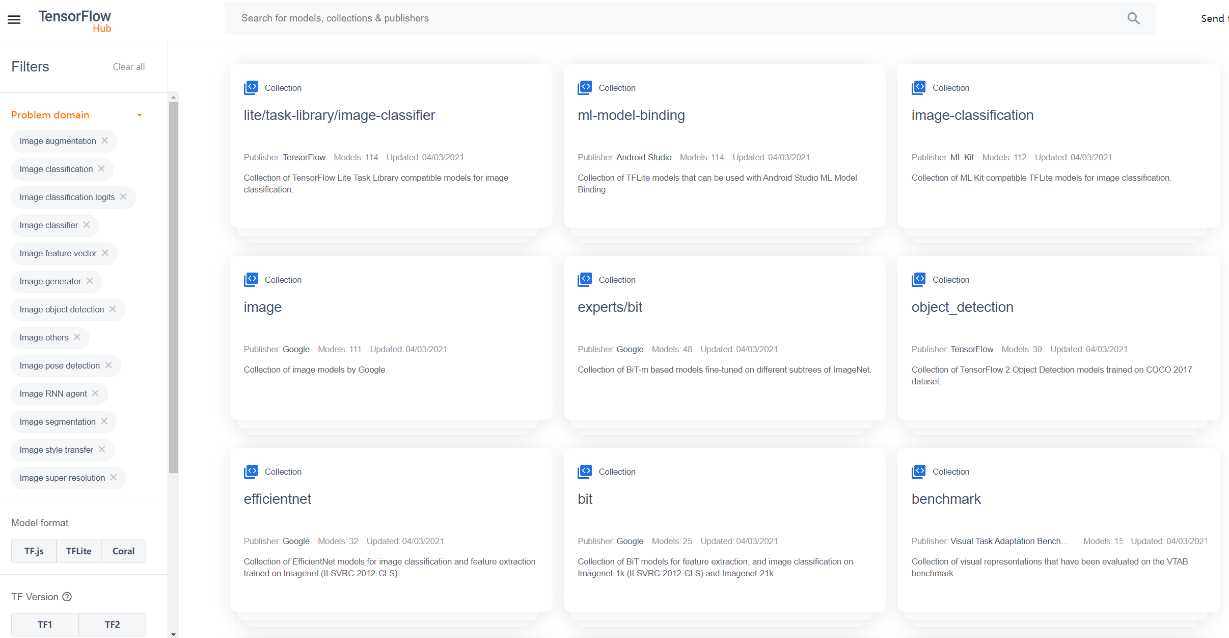


图8-4 TensorFlow Hub

打开TensorFlow Hub网站的主页，在页面左侧可以选取类别，例如Text，Image，Video和Publishers等选项，或在搜索框中输入关键字搜索所需要的模型。

以MobileNet为例，搜索到的模型如图8-5所示，在选择模型是请注意TensorFlow的版本。

可以直接下载模型，或者使用hub.KerasLayer。

m = tf.keras.Sequential([

hub.KerasLayer("https://hub.tensorflow.google.cn/google/imagenet/mobilenet\_v3\_small\_100\_224/feature\_vector/5", trainable=False),

tf.keras.layers.Dense(num\_classes, activation='softmax')

])

m.build([None, 224, 224, 3]) # Batch input shape.

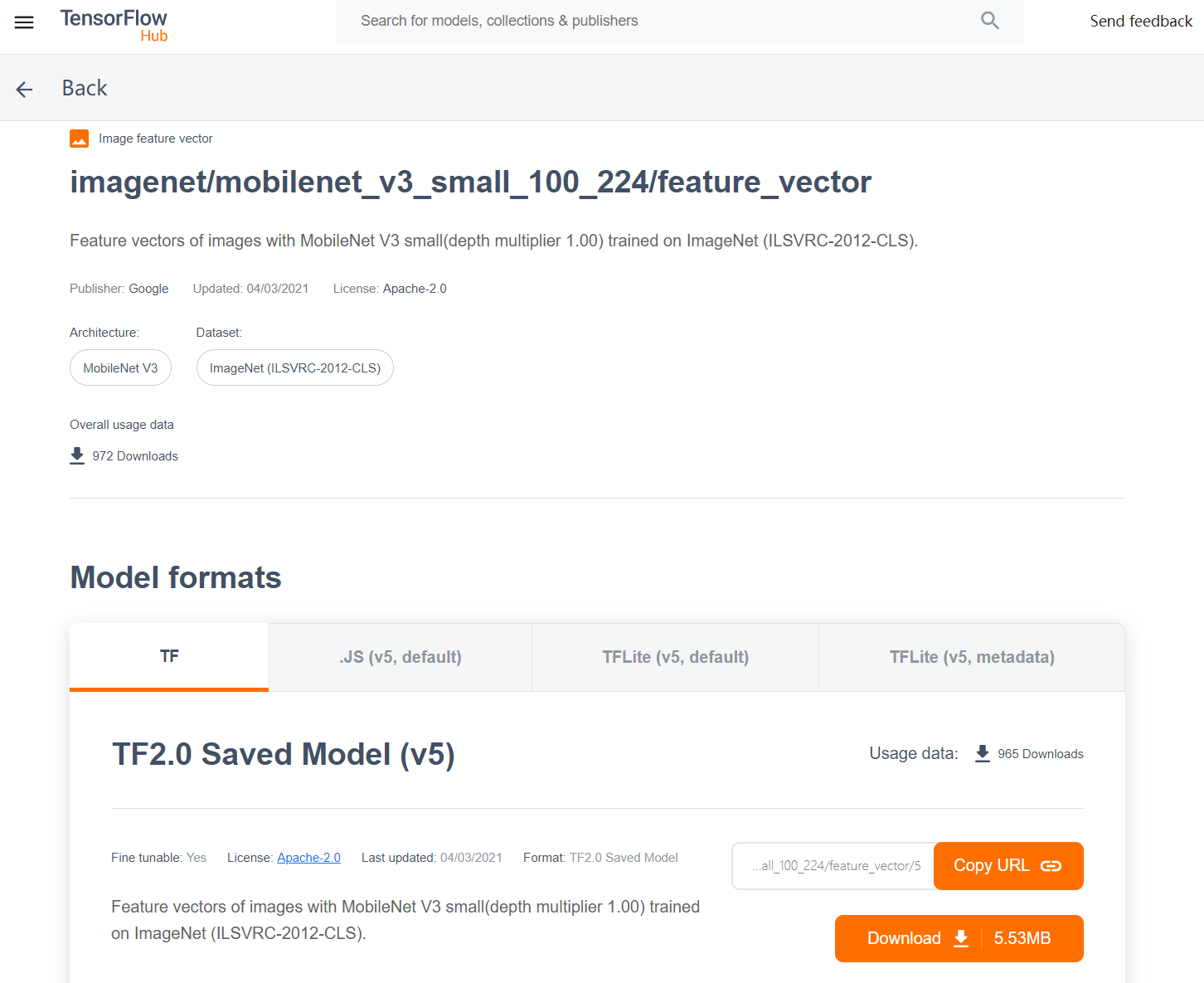


图8-5 MobileNet下载页面

### 8.3.2 模型转换

TensorFlow Lite转换器将输入的TensorFlow 模型生成 TensorFlow Lite 模型，一种优化的 FlatBuffer 格式，以 .tflite 为文件扩展名，可以通过命令行与Python API使用此转换器。

Google推荐使用Python API进行转换，命令行工具只提供了基本的转化功能。转换后的原模型为 FlatBuffers 格式。 FlatBuffers主要应用于游戏场景，是为了高性能场景创建的序列化库，相比Protocol Buffer有更高的性能和更小的大小等优势，更适合于边缘设备部署。

1. 命令行

TensorFlow Lite 转换器命令行工具 tflite\_convert是与TensorFlow一起安装的，在终端运行如下命令：

$ tflite\_convert --help

`--output\_file`. Type: string. Full path of the output file.

`--saved\_model\_dir`. Type: string. Full path to the SavedModel directory.

`--keras\_model\_file`. Type: string. Full path to the Keras H5 model file.

`--enable\_v1\_converter`. Type: bool. (default False) Enables the converter and flags used in TF 1.x instead of TF 2.x.

参数说明如下：

* output\_file. 类型: string. 指定输出文件的绝对路径。
* saved\_model\_dir. 类型: string. 指定含有 TensorFlow 1.x 或者 2.0 使用 SavedModel 生成文件的绝对路径目录。
* keras\_model\_file. Type: string. 指定含有 TensorFlow 1.x 或者 2.0 使用 tf.keras model 生成 HDF5 文件的绝对路径目录。

在 TensorFlow模型导出时支持两种模型导出方法和格式SavedModel和Keras Sequential。

转换转换 SavedModel示例如下：

tflite\_convert \

--saved\_model\_dir=/tmp/mobilenet\_saved\_model \

--output\_file=/tmp/mobilenet.tflite

转换转换 Keras H5示例如下：

tflite\_convert \

--keras\_model\_file=/tmp/mobilenet\_keras\_model.h5 \

--output\_file=/tmp/mobilenet.tflite

1. Python API

在 TensorFlow 2.0 中，将TensorFlow模型格式转换为TensorFlow Lite 的 Python API 是 tf.lite.TFLiteConverter。在 TFLiteConverter 中有以下的类方法：

* TFLiteConverter.from\_saved\_model()：用来转换 SavedModel 格式模型。
* TFLiteConverter.from\_keras\_model()：用来转换 tf.keras 模型。
* TFLiteConverter.from\_concrete\_functions()：用来转换 concrete functions。

若要详细了解 TensorFlow Lite converter API，请运行 print(help(tf.lite.TFLiteConverter))。TensorFlow 2.x 模型是使用 SavedModel 格式存储的，并通过高阶 tf.keras.\* API（Keras 模型）或低阶 tf.\* API（用于生成具体函数）生成。

以下示例演示了如何将 SavedModel 转换为 TensorFlow Lite 模型。

import tensorflow as tf

# Convert the model

converter = tf.lite.TFLiteConverter.from\_saved\_model(saved\_model\_dir) # path to the SavedModel directory

tflite\_model = converter.convert()

# Save the model.

with open('model.tflite', 'wb') as f:

f.write(tflite\_model)

以下示例演示了如何将 Keras 模型转换为 TensorFlow Lite 模型。

import tensorflow as tf

# Create a model using high-level tf.keras.\* APIs

model = tf.keras.models.Sequential([

tf.keras.layers.Dense(units=1, input\_shape=[1]),

tf.keras.layers.Dense(units=16, activation='relu'),

tf.keras.layers.Dense(units=1)

])

model.compile(optimizer='sgd', loss='mean\_squared\_error') # compile the model

model.fit(x=[-1, 0, 1], y=[-3, -1, 1], epochs=5) # train the model

# (to generate a SavedModel) tf.saved\_model.save(model, "saved\_model\_keras\_dir")

# Convert the model.

converter = tf.lite.TFLiteConverter.from\_keras\_model(model)

tflite\_model = converter.convert()

# Save the model.

with open('model.tflite', 'wb') as f:

f.write(tflite\_model)

## 8.3.3 模型推理

TensorFlow Lite 解释器接收一个模型文件，执行模型文件在输入数据上定义的运算符，输出推理结果，通过模型运行数据以获得预测的过程。

解释器适用于多个平台，提供了一个简单的 API，用于从 Java、Swift、Objective-C、C++ 和 Python 运行 TensorFlow Lite 模型。

Java 调用解释器的方式如下：

try (Interpreter interpreter = new Interpreter(tensorflow\_lite\_model\_file)) {

interpreter.run(input, output);

}

如果手机有GPU， GPU 比 CPU 执行更快的浮点矩阵运算，速度提升能有显著效果。例如在有GPU加速的手机上运行MobileNet图像分类，模型运行速度可以提高 5.5 倍。

TensorFlow Lite 解释器可以配置委托（Delegates）以在不同设备上使用硬件加速。GPU 委托（GPU Delegates）允许解释器在设备的 GPU 上运行适当的运算符。

下面的代码显示了从 Java 中使用 GPU 委托的方式:

GpuDelegate delegate = new GpuDelegate();

Interpreter.Options options = (new Interpreter.Options()).addDelegate(delegate);

Interpreter interpreter = new Interpreter(tensorflow\_lite\_model\_file, options);

try {

interpreter.run(input, output);

}

TensorFlow Lite 解释器很容易在Android与iOS平台上使用。Android 开发人员应该使用 TensorFlow Lite AAR。iOS 开发人员应该使用 CocoaPods for Swift or Objective-C。

TensorFlow Lite 解释器同样可以部署在Raspberry Pi 和基于 Arm64 的主板的嵌入式 Linux系统上。

## 8.3.4 优化模型

TensorFlow Lite 提供了优化模型大小和性能的工具，通常对准确性影响甚微。模型优化的目标是在给定设备上，实现性能、模型大小和准确性的理想平衡。根据任务的不同，你会需要在模型复杂度和大小之间做取舍。如果任务需要高准确率，那么你可能需要一个大而复杂的模型。对于精确度不高的任务，就最好使用小一点的模型，因为小的模型不仅占用更少的磁盘和内存，也一般更快更高效。

量化使用了一些技术，可以降低权重的精确表示，并且可选的降低存储和计算的激活值。量化的好处有:

* 对现有 CPU 平台的支持。
* 激活值得的量化降低了用于读取和存储中间激活值的存储器访问成本。
* 许多 CPU 和硬件加速器实现提供 SIMD 指令功能，这对量化特别有益。

TensorFlow Lite 对量化提供了多种级别的对量化支持。

* Tensorflow Lite post-training quantization 量化使权重和激活值的 Post training 更简单。
* Quantization-aware training 可以以最小精度下降来训练网络；这仅适用于卷积神经网络的一个子集。

以下的 Python 代码片段展示了如何使用预训练量化进行模型转换：

import tensorflow as tf

converter = tf.lite.TFLiteConverter.from\_saved\_model(saved\_model\_dir)

converter.optimizations = [tf.lite.Optimize.OPTIMIZE\_FOR\_SIZE]

tflite\_quant\_model = converter.convert()

open("converted\_model.tflite", "wb").write(tflite\_quant\_model)

## 8.4 任务2：实现花卉识别

下面将使用TensorFlow Lite实现花卉识别app，在Android设备上运行图像识别模型MobileNets\_v2来识别花卉。本项目实施步骤如下：

1. 通过迁移学习实现花卉识别模型
2. 使用TFLite转换器转换模型。
3. 在Android应用中使用TFLite解释器运行它。
4. 使用 TensorFlow Lite支持库预处理模型输入和后处理模型输出。

最后实现一个在手机上运行的app，可以实时识别照相机所拍摄的花卉，如图8-6所示。

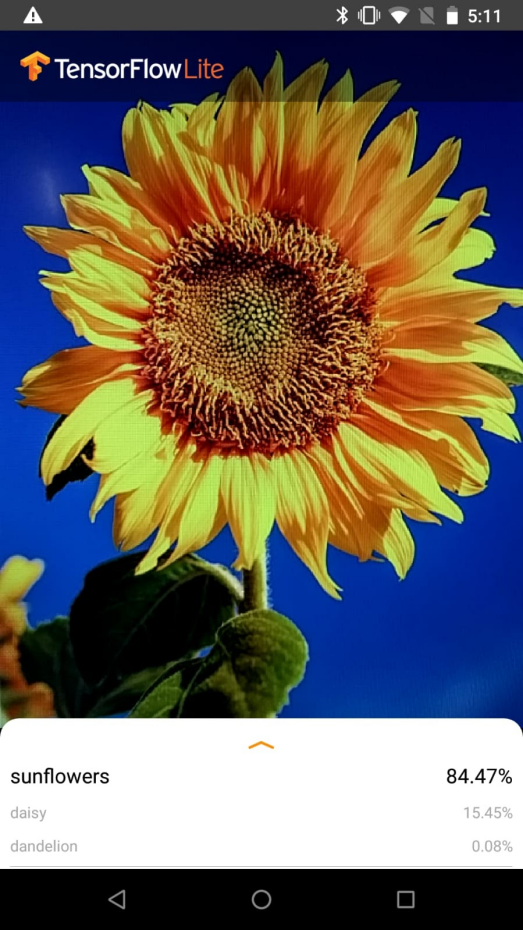


图8-6 花卉识别app

## 8.4.1 选择模型

选择MobileNet V2进行迁移学习，实现识别花卉模型。MobileNet V2是基于一个流线型的架构，它使用深度可分离的卷积来构建轻量级的深层神经网。可用于图像分类任务，比如猫狗分类、花卉分类等等。提供一系列带有标注的花卉数据集，该算法会载入在ImageNet-1000上的预训练模型，在花卉数据集上做迁移学习。

使用小型数据集时，通常会利用在同一域中的较大数据集上训练的模型所学习的特征。通过实例化预先训练的模型，并在顶部添加全连接的分类器来完成的。预训练的模型被“冻结”并且仅在训练期间更新分类器的权重。在这种情况下，卷积基提取了与每幅图像相关的所有特征，只需训练一个分类器，根据所提取的特征集确定图像类。

通过微调进一步提高性能，调整预训练模型的顶层权重，以便模型学习特定于数据集的高级特征，当训练数据集很大并且非常类似于预训练模型训练的原始数据集时，通常建议使用此技术。

1. 导入相关库

In[1]:

import tensorflow as tf

assert tf.\_\_version\_\_.startswith('2')

import os

import numpy as np

import matplotlib.pyplot as plt

1. 准备数据集

该数据集可以在http://download.tensorflow.org/example\_images/flower\_photos.tgz下载。每个子文件夹都存储了一种类别的花的图片，子文件夹的名称就是花的类别的名称。平均每一种花有734张图片，图片都是RGB色彩模式的。

In[2]:

\_URL = "http://download.tensorflow.org/example\_images/flower\_photos.tgz"

zip\_file = tf.keras.utils.get\_file(origin=\_URL,

fname="flower\_photos.tgz",

extract=True)

base\_dir = os.path.join(os.path.dirname(zip\_file), 'flower\_photos')

数据集解压后存放在.keras\datasets\flower\_photos目录下。

2016/02/11 04:52 <DIR> daisy

2016/02/11 04:52 <DIR> dandelion

2016/02/09 10:59 418,049 LICENSE.txt

2016/02/11 04:52 <DIR> roses

2016/02/11 04:52 <DIR> sunflowers

2016/02/11 04:52 <DIR> tulips

将数据集划分为训练集和验证集。训练前需要手动加载图像数据，完成包括遍历数据集的目录结构、加载图像数据以及返回输入和输出。可以使用 Keras 提供的 ImageDataGenerator 类，它是keras.preprocessing.image模块中的图片生成器，负责生成一个批次一个批次的图片，以生成器的形式给模型训练

ImageDataGenerator的构造函数包含许多参数，用于指定加载后如何操作图像数据，包括像素缩放和数据增强。

接着需要一个迭代器来逐步加载单个数据集的图像。这需要调用flow\_from\_directory（）函数并指定该数据集目录，如 train、validation 目录，函数还允许配置与加载图像相关的更多细节。 target\_size参数允许将所有图像加载到一个模型需要的特定的大小，设置为大小为(224, 224)的正方形图像。

batch\_size默认的为32，意思是训练时从数据集中的不同类中随机选出的32个图像，该值设置为64。 在评估模型时，可能还希望以确定性顺序返回批处理，这可以通过将 shuffle参数设置为False。

In[3]:

IMAGE\_SIZE = 224

BATCH\_SIZE = 64

datagen = tf.keras.preprocessing.image.ImageDataGenerator(

rescale=1./255,

validation\_split=0.2)

train\_generator = datagen.flow\_from\_directory(

base\_dir,

target\_size=(IMAGE\_SIZE, IMAGE\_SIZE),

batch\_size=BATCH\_SIZE,

subset='training')

val\_generator = datagen.flow\_from\_directory(

base\_dir,

target\_size=(IMAGE\_SIZE, IMAGE\_SIZE),

batch\_size=BATCH\_SIZE,

subset='validation')

Out[3]:

Found 2939 images belonging to 5 classes.

Found 731 images belonging to 5 classes.

In[4]:

for image\_batch, label\_batch in train\_generator:

break

image\_batch.shape, label\_batch.shape

Out[4]:

((64, 224, 224, 3), (64, 5))

保存标签文件：

In[5]:

print (train\_generator.class\_indices)

labels = '\n'.join(sorted(train\_generator.class\_indices.keys()))

with open('labels.txt', 'w') as f:

f.write(labels)

1. 迁移学习改造模型

实例化一个预加载了ImageNet训练权重的MobileNet V2模型。

In[6]:

IMG\_SHAPE = (IMAGE\_SIZE, IMAGE\_SIZE, 3)

# Create the base model from the pre-trained model MobileNet V2

base\_model = tf.keras.applications.MobileNetV2(input\_shape=IMG\_SHAPE,

include\_top=False,

weights='imagenet')

Out[6]:

Downloading data from [https://storage.googleapis.com/tensorflow/keras-applications/mobilenet\_v2/mobilenet\_v2\_weights\_tf\_dim\_ordering\_tf\_kernels\_1.0\_224\_no\_top.h5](https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5" \t "_blank)

9412608/9406464 [==============================] - 2s 0us/step

MobileNet V2模型默认是将图片分类到1000类，每一类都有各自的标注。因为本问题分类只有5类，所以构建模型的时候增加include\_top=False参数，表示不需要原有模型中最后的神经网络层（分类到1000类），以便增加自己的输出层。

由于是通过迁移学习改造模型，所以不改变基础模型的各项参数变量，因为这样才能保留原来大规模训练的优势。使用model.trainable = False，设置在训练中，基础模型的各项参数变量不会被新的训练修改数据。

您需要选择用于特征提取的MobileNet V2层，显然，最后一个分类层（在“顶部”，因为大多数机器学习模型的图表从下到上）并不是非常有用。相反，您将遵循通常的做法，在展平操作之前依赖于最后一层，该层称为“瓶颈层”，与最终/顶层相比，瓶颈层保持了很多通用性。随后在原有模型的后面增加一个池化层，对数据降维。最后是一个5个节点的输出层，因为需要的结果只有5类。

要从特征块生成预测，请用5x5在空间位置上进行平均，使用tf.keras.layers.GlobalAveragePooling2D层将特征转换为每个图像对应一个1280元素向量。

In[7]:

base\_model.trainable = False

model = tf.keras.Sequential([

  base\_model,

  tf.keras.layers.Conv2D(32, 3, activation='relu'),

  tf.keras.layers.Dropout(0.2),

  tf.keras.layers.GlobalAveragePooling2D(),

  tf.keras.layers.Dense(5, activation='softmax')

])

1. 编译，训练模型

在训练之前先编译模型，损失函数使用类别交叉熵。

In[8]:

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss='categorical\_crossentropy',

metrics=['accuracy'])

Out[8]:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

mobilenetv2\_1.00\_224 (Functi (None, 7, 7, 1280) 2257984

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d (Conv2D) (None, 5, 5, 32) 368672

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 5, 5, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_average\_pooling2d (Gl (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 5) 165

=================================================================

Total params: 2,626,821

Trainable params: 368,837

Non-trainable params: 2,257,984

训练模型，训练和验证准确性/损失的学习曲线如图8-7所示。

In[9]:

epochs = 10

history = model.fit(train\_generator, steps\_per\_epoch=len(train\_generator),

epochs=epochs, validation\_data=val\_generator,

validation\_steps=len(val\_generator))

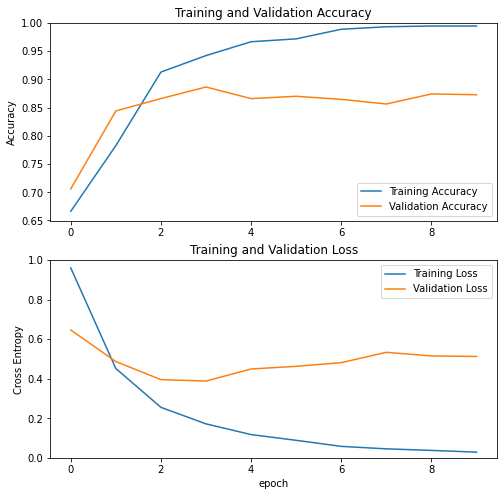


图8-7 学习曲线

1. 微调

设置model.trainable = False参数后，训练期间将不更新预训练网络的权重，只在MobileNet V2基础模型上训练了几层。如果希望进一步提高性能的方法是训练预训练模型的顶层的权重以及刚添加的分类器的训练。

只有在训练顶层分类器并将预先训练的模型设置为不可训练之后，才应尝试此操作。如果在预先训练的模型上添加一个随机初始化的分类器并尝试联合训练所有层，则梯度更新的幅度将太大，并且预训练模型将忘记它学到的东西。

应该尝试微调少量顶层而不是整个MobileNet模型，前几层学习非常简单和通用的功能，这些功能可以推广到几乎所有类型的图像，随着层越来越高，这些功能越来越多地针对训练模型的数据集。微调的目的是使这些专用功能适应新数据集，而不是覆盖通用学习。

首先取消冻结模型的顶层，代码如下：

In[10]:

base\_model.trainable = True

# Let's take a look to see how many layers are in the base model

print("Number of layers in the base model: ", len(base\_model.layers))

# Fine tune from this layer onwards

fine\_tune\_at = 100

# Freeze all the layers before the `fine\_tune\_at` layer

for layer in base\_model.layers[:fine\_tune\_at]:

layer.trainable = False

Out[10]:

Number of layers in the base model: 155

取消冻结base\_model，MobileNet V2模型网络一共155层，前100层仍设置为无法训练，然后重新编译模型，并恢复训练。使用低学习率编译模型，代码如下：

In[11]:

model.compile(loss='categorical\_crossentropy',

              optimizer = tf.keras.optimizers.Adam(1e-5),

              metrics=['accuracy'])

model.summary()

如果你训练得更早收敛，这将使你的准确率提高几个百分点。

history\_fine = model.fit(train\_generator,

                         steps\_per\_epoch=len(train\_generator),

                         epochs=5,

                         validation\_data=val\_generator,

                         validation\_steps=len(val\_generator))

经过微调后，模型精度几乎达到98%，当微调MobileNet V2基础模型的最后几层并在其上训练分类器时，验证损失远远高于训练损失，模型可能有一些过度拟合。

1. 转换为TFLite格式

使用tf.saved\_model.save保存模型，然后将将模型保存为tf lite兼容格式。

SavedModel 包含一个完整的 TensorFlow 程序——不仅包含权重值，还包含计算。它不需要原始模型构建代码就可以运行。

saved\_model\_dir = 'save/fine\_tuning'

tf.saved\_model.save(model, saved\_model\_dir)

converter = tf.lite.TFLiteConverter.from\_saved\_model(saved\_model\_dir)

tflite\_model = converter.convert()

with open('save/fine\_tuning/assets/model.tflite', 'wb') as f:

f.write(tflite\_model)

模型文件保存在save\fine\_tuning\assets目录下。

## 8.4.2 Android部署

我们已经使用MobileNet V2 创建、训练和导出了自定义TensorFlow Lite模型，已经导出以下经过训练的TF Lite模型文件和标签文件。接下来将在手机端部署，运行一个使用该模型识别花卉图片的Android 应用。

1. 准备工作

Tensor Flow官网提供了很多有趣的TensorFlow Lite示例，可从github下载源码：

git clone https://github.com/tensorflow/examples.git

项目代码位于目录examples/lite/codelabs/flower\_classification/android/，start目录下为项目模板，finish目录下是项目完整代码。

安装Android Studio，确认Android Studio版本 3.0+以上，如图8-8。

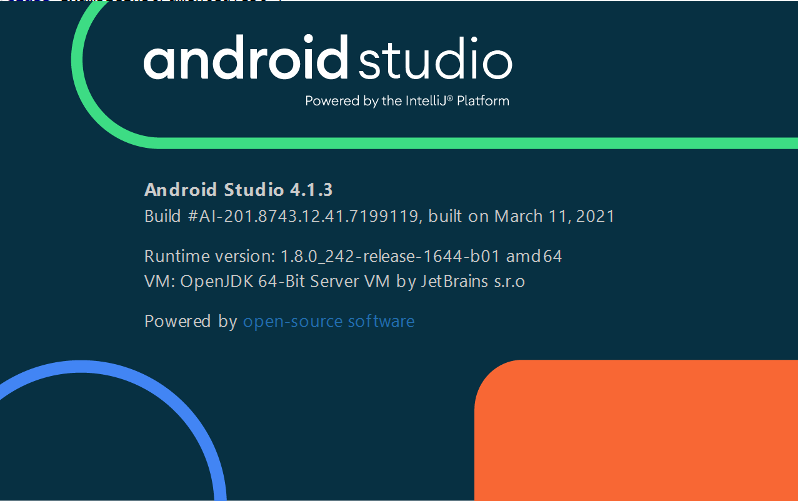


图8-8 Android Studio版本信息

打开 Android Studio Android Studio“启动”图标。 该工具加载后，从以下弹出式窗口中选择 Android Studio“打开项目”图标“打开现有 Android Studio 项目”(Open an existing project)：

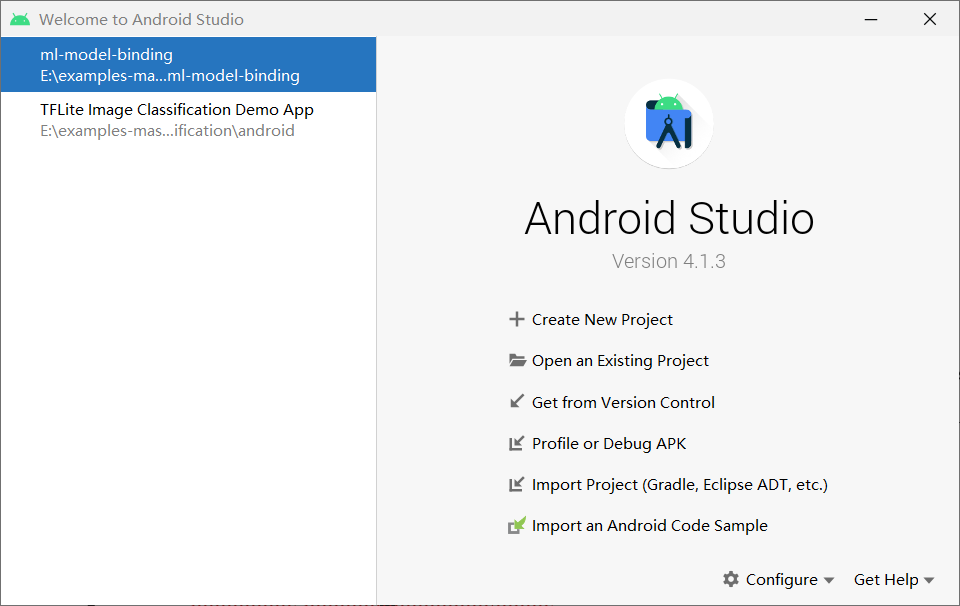


图8-9 使用 Android Studio 打开项目

工作目录中选择 examples/lite/codelabs/flower\_classification/android/finish。

将TensorFlow Lite 模型文件model.tflite，标签文件label.txt 拷贝到项目文件夹下/android/start/app/src/main/assets/。

1. 配置build.gradle

首次打开项目时，会看到一个“Gradle同步”(Gradle Sync) 弹出式窗口，询问是否要使用 Gradle 封装容器。在Gradle同步前先将模型文件拷贝到assets目录下。

要使用tensorflow lite需要导入对应的库，这里通过修改build.gradle来实现。

在dependencies下增加'org.tensorflow:tensorflow-lite:+'，代码如下：

implementation('org.tensorflow:tensorflow-lite:0.0.0-nightly') { changing = true }

implementation('org.tensorflow:tensorflow-lite-gpu:0.0.0-nightly') { changing = true }

implementation('org.tensorflow:tensorflow-lite-support:0.0.0-nightly') { changing = true }

在android下增加 aaptOptions，以防止Android在生成应用程序二进制文件时压缩TensorFlow Lite模型文件。代码如下：

aaptOptions {

  noCompress "tflite"

}

运行Sync Gradle开始Android环境部署， 运行结果如图8-10：

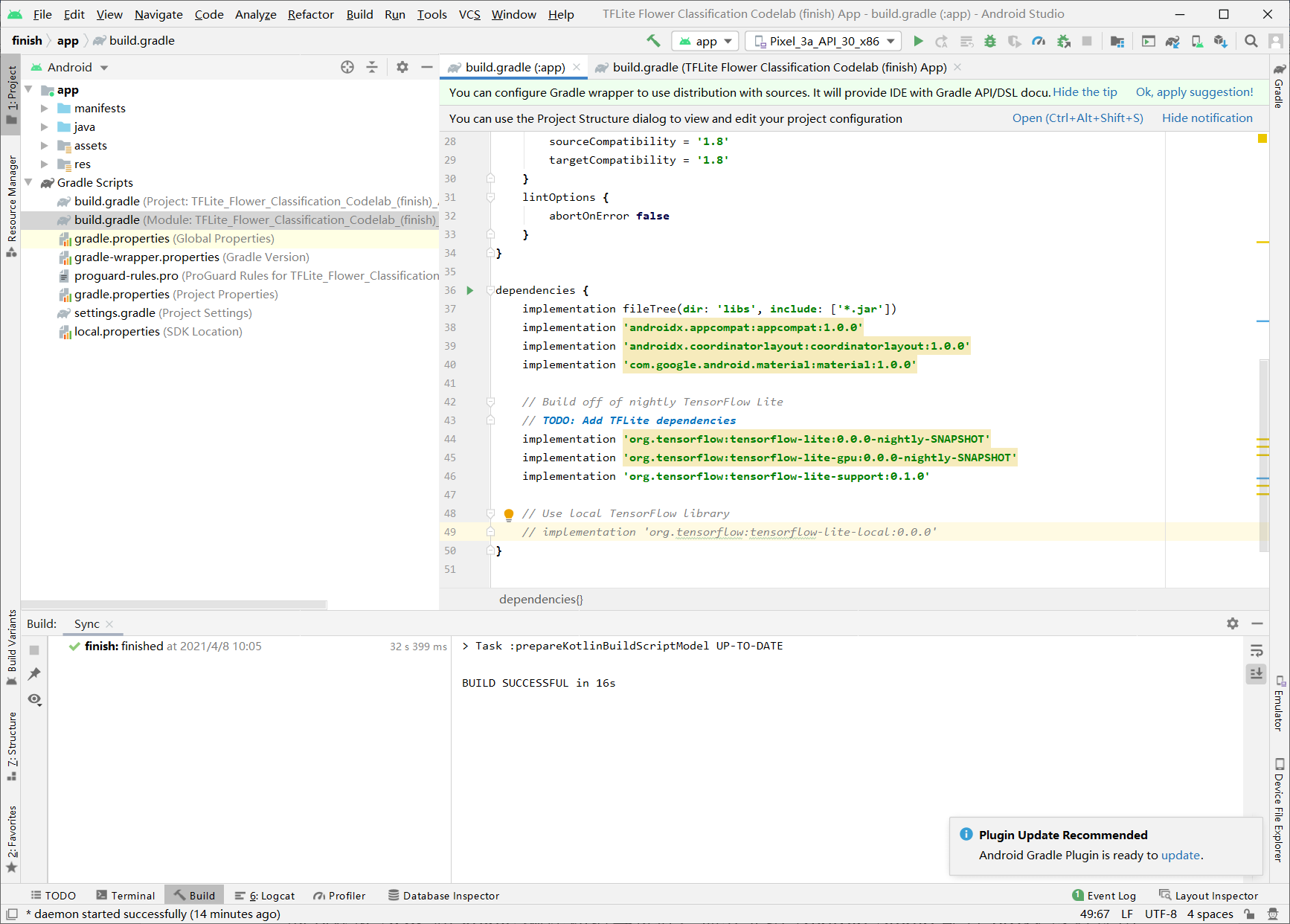


图8-10 Gradle同步结果

因为获取SDK和gradle编译环境等资源，需要先给Android Studio配置proxy或者使用国内的镜像。可将 build.gradle 中的maven源 google() 和 jcenter() 分别替换为国内镜像，如下：

buildscript {

repositories {

maven { url ' https://maven.aliyun.com/repository/google '}

maven { url ' https://maven.aliyun.com/repository/jcenter '}

}

dependencies {

classpath 'com.android.tools.build:gradle:3.5.1'

}

}

allprojects {

repositories {

maven { url ' https://maven.aliyun.com/repository/google '}

maven { url ' https://maven.aliyun.com/repository/jcenter '}

}

}

1. 初始化TensorFlow Lite解释器

推理过程是通过解释器（interpreter）来执行，首先是读取模型，将.tflite模型加载到内存中，其中包含了模型的执行流图。

修改ClassifierFloatMobileNet.java文件的ClassifierFloatMobileNet类，添加model.tflite 和 label.txt，代码如下：

public class ClassifierFloatMobileNet extends Classifier {

...

// TODO: Specify model.tflite as the model file and labels.txt as the label file

@Override

protected String getModelPath() {

return "model.tflite";

}

@Override

protected String getLabelPath() {

return "labels.txt";

}

...

}

在Classifier.java文件中的Classifier类里声明TFLite 解释器tflite，如果有GPU，还需要声明GPU代理gpuDelegate。

protected Classifier(Activity activity, Device device, int numThreads) throws IOException {

...

// TODO: Declare a GPU delegate

private GpuDelegate gpuDelegate = null;

/\*\* An instance of the driver class to run model inference with Tensorflow Lite. \*/

// TODO: Declare a TFLite interpreter

protected Interpreter tflite;

...

}

在Classifier类构造函数中创建tflite实例。

protected Classifier(Activity activity, Device device, int numThreads) throws IOException {

...

switch (device) {

case GPU:

// TODO: Create a GPU delegate instance and add it to the interpreter options

gpuDelegate = new GpuDelegate();

tfliteOptions.addDelegate(gpuDelegate);

break;

case CPU:

break;

}

tfliteOptions.setNumThreads(numThreads);

// TODO: Create a TFLite interpreter instance

tflite = new Interpreter(tfliteModel, tfliteOptions);

...

}

1. 执行推理

TensorFlow Lite解释器初始化后，开始编写代码以识别输入图像。 TensorFlow Lite无需使用ByteBuffer来处理图像， 它提供了一个方便的支持库来简化图像预处理，同样还可以处理模型的输出，并使TensorFlow Lite解释器更易于使用。下面需要做的工作有：

1. 数据转换（Transforming Data）：将输入数据转换成模型接收的形式或排布，如resize原始图像到模型输入大小；
2. 执行推理（Running Inference）：这一步使用API来执行模型。其中包括了创建解释器、分配张量等；
3. 解释输出（Interpreting Output）：用户取出模型推理的结果，并解读输出，如分类结果的概率。

首先处理摄像头的输入图像，修改Classifier.java文件中的loadImage方法，代码如下：

private TensorImage loadImage(final Bitmap bitmap, int sensorOrientation) {

...

// TODO: Define an ImageProcessor from TFLite Support Library to do preprocessing

ImageProcessor imageProcessor =

new ImageProcessor.Builder()

.add(new ResizeWithCropOrPadOp(cropSize, cropSize))

.add(new ResizeOp(imageSizeX, imageSizeY, ResizeMethod.NEAREST\_NEIGHBOR))

.add(new Rot90Op(numRoration))

.add(getPreprocessNormalizeOp())

.build();

return imageProcessor.process(inputImageBuffer);

...

}

修改recognizeImage方法执行推理，将预处理后的图像提供给TensorFlow Lite解释器。代码如下:

public List<Recognition> recognizeImage(final Bitmap bitmap, int sensorOrientation) {

...

// TODO: Run TFLite inference

tflite.run(inputImageBuffer.getBuffer(), outputProbabilityBuffer.getBuffer().rewind());

...

}

最后从模型输出中获取类别及其概率。

public List<Recognition> recognizeImage(final Bitmap bitmap, int sensorOrientation) {

...

// TODO: Use TensorLabel from TFLite Support Library to associate the probabilities with category labels  
 Map<String, Float> labeledProbability =

new TensorLabel(labels, probabilityProcessor.process(outputProbabilityBuffer))

.getMapWithFloatValue();

...

}

labeledProbability是将每个类别映射到其概率的对象。 TensorFlow Lite支持库提供了一个方便的实用程序，可将模型输出转换为概率图，使用getTopKProbability（..）方法从labeledProbability中提取前几名最可能的标签。

1. 试运行应用

应用可以在Android 设备上运行，也可以在 Android Studio 模拟器中运行。如果计算机没有摄像头就必须选择Android 设备运行该应用。

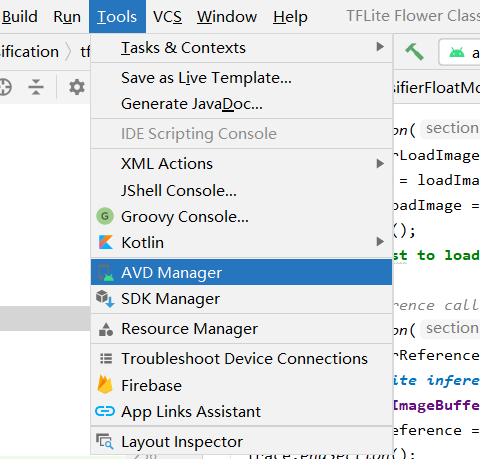


图8-11 设置模拟器

Android Studio 可轻松设置模拟器，选择“Tools”， “AVD Manager”。

如需设置模拟器的相机，需要在“Android 虚拟设备管理器”(Android Virtual Device Manager)中创建一个新设备。从 ADVM 主页面中选择“创建虚拟设备”(Create Virtual Device)：

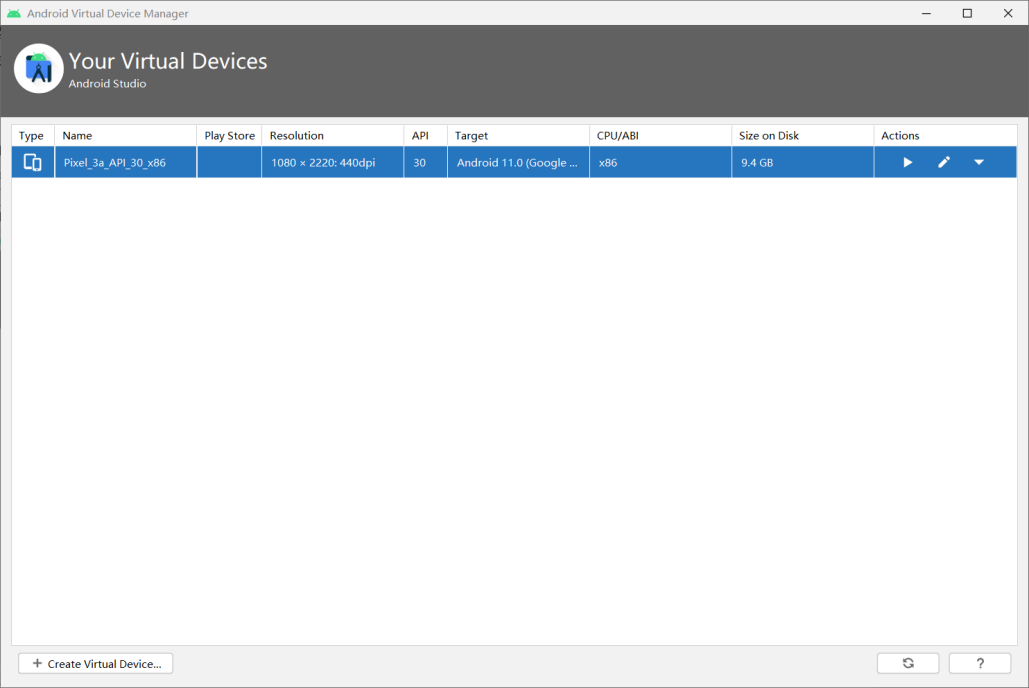


图8-12 创建虚拟设备

然后在“验证配置”(Verify Configuration) 页面（虚拟设备设置的最后一页）上，选择“显示高级设置”(Show Advanced Settings)：

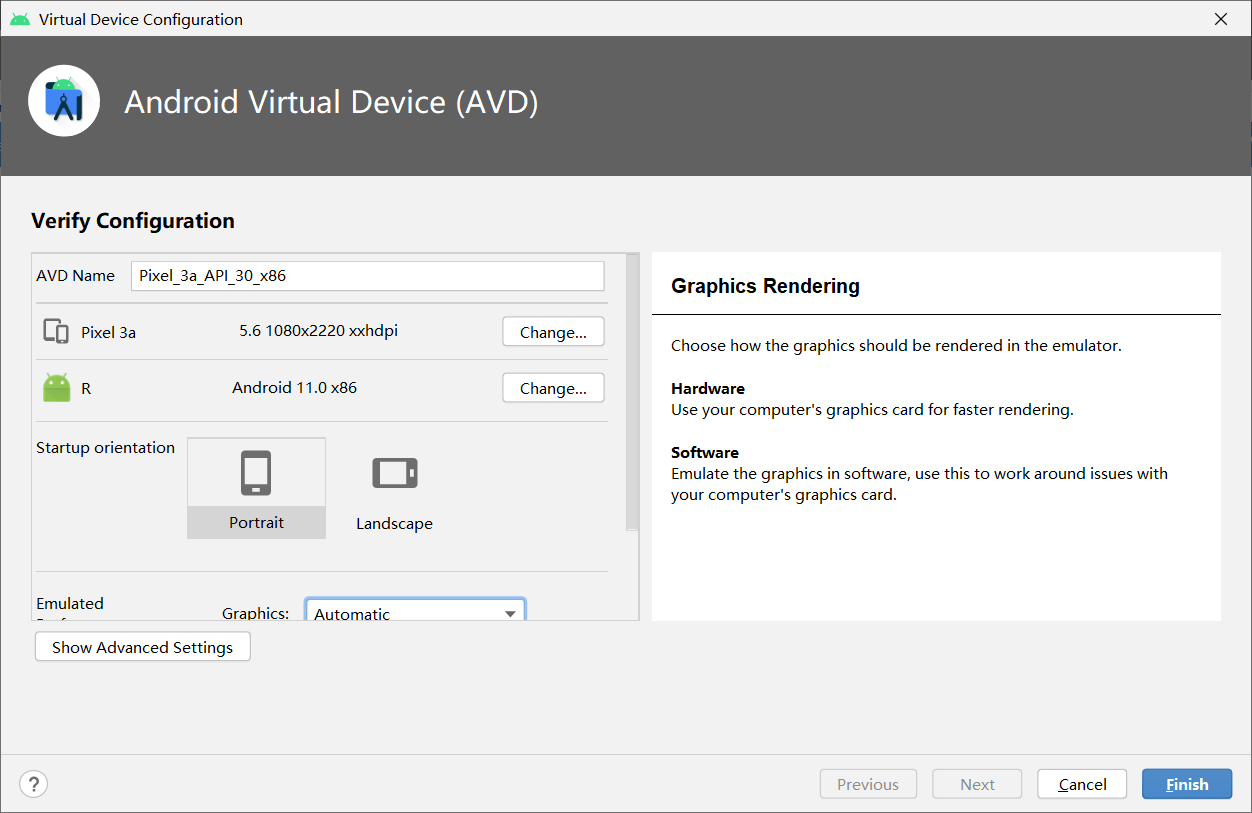


图8-13 高级设置

如图在Android 设备上运行，设置手机启用“开发者模式”和“USB 调试”，否则无法将该应用从 Android Studio 加载到您的手机上。

如需启动构建和安装过程，运行Gradle同步。

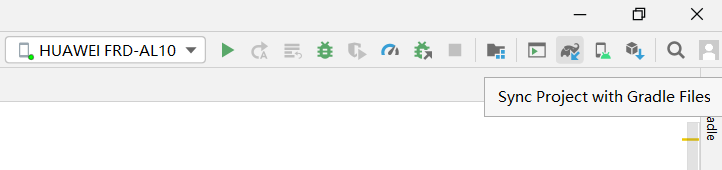


图8-14 运行 Gradle 同步

运行 Gradle 同步后，请选择单击Android Studio“开始”图标Android Studio“播放”图标，需要从如图8-15窗口中选择设备，运行结果如图8-16所示。

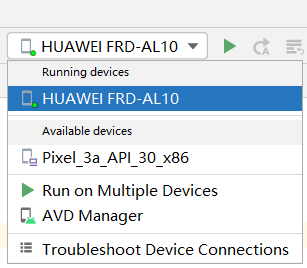
 

图8-15 选择设备 图8-16 运行结果

## 拓展项目

TensorFlow 官网提供了很多TensorFlow Lite 示例应用，探索经过预先训练的 TensorFlow Lite 模型，了解如何在示例应用中针对各种机器学习应用场景使用这些模型。并且分别提供了Android设备、iOS设备以及Raspberry Pi上的应用实现代码，如图8-17所示。

本项目任务要求是基于TensorFlow Lite开发一个安卓示例应用程序，应用程序利用设备的摄像头来实时地检测和显示一个人的关键部位。

通过PoseNet模型实现人体姿势估计，PoseNet可以通过检测关键身体部位的位置来估计图像或者视频中的人体姿势。例如，该模型可以估计图像中人的手肘和/或膝盖位置。

可参考示例代码：

<https://github.com/tensorflow/examples/tree/master/lite/examples/posenet/android>。



图8-17 图像分类示例应用

该PoseNet示例应用程序功能是捕捉摄像头拍摄的帧，并实时覆盖图像上的关键点。应用程序对每张传入的摄像头图像执行以下操作：

* 从摄像头预览中获取图像数据并转换成格式。
* 创建一个位图对象来保存来自 RGB 格式帧数据的像素。将位图裁剪并缩放到模型输入的大小，以便将其传递给模型。
* 从 PoseNet 库中调用estimateSinglePose()函数来获取Person对象。
* 将位图缩放回屏幕大小，在Canvas对象上绘制新的位图。
* 使用从Person对象中获取的关键点位置在画布上绘制骨架。显示置信度超过特定阈值（默认值为 0.2）的关键点。