

12.2.3 小节的代码第 13 行，是以文本方式将冻结图的内容打印出来，直观性相对较差。还可以通过 import_to_tensorboard 工具将生成的冻结图文件导入到概要日志中，并用 TensorBoard 工具进行查看。具体做法如下：

12.4.1 使用 import_to_tensorboard 工具

在 TensorFlow 的安装路径中可以找到 import_to_tensorboard 工具的脚本。具体如下：

```
Anaconda3\lib\site-packages\tensorflow\python\tools\ import_pb_to_tensorboard.py
```

代码文件“import_pb_to_tensorboard.py”是一个可以单独在命令行中运行的脚本文件，可以在命令行中使用它，也可以在代码中使用它。

1. 在命令行里使用 import_to_tensorboard 工具

在命令行里使用 import_to_tensorboard 工具，可以输入如下命令：

```
python import_pb_to_tensorboard.py --model_dir 冻结图文件路径 --log_dir 导出的概要日志路径
```

该命令在执行时，会调用脚本中的 import_to_tensorboard 函数。该函数的具体定义如下：

```
def import_to_tensorboard(model_dir, log_dir)
```

2. 在代码中使用 import_to_tensorboard 工具

编写代码来使用 import_to_tensorboard 工具。具体如下：

```
from tensorflow.python.tools import import_pb_to_tensorboard
input_graph_path = "log/expert-graph-yes.pb" #冻结图文件
import_pb_to_tensorboard.import_to_tensorboard(input_graph_path,'./pbvisualize')
```

上面代码运行后，会在“./pbvisualize”路径下生成概要日志。该概要日志可以被 TensorBoard 工具读取。

12.4.2 用 TensorBoard 工具查看模型结构

首先启动 TensorBoard 工具，接着在浏览器中查看模型结构。具体如下：

1. 在命令行中启动 TensorBoard 工具

在 12.4.1 小节中的程序运行之后，会在“./pbvisualize”路径下得到概要日志。

在命令行窗口中，将当前位置切换到 pbvisualize 文件夹的上级目录（作者的路径为 G:\python3），并启动 TensorBoard 工具。输入命令如下：

```
G:\python3>tensorboard --logdir=./pbvisualize
```

该命令执行之后，输出结果如图 12-4 所示。



图 12-4 启动 TensorBoard

如图 12-4 所示，最后 1 行是 TensorBoard 工具的访问地址。

在浏览器中输入 “<http://LAPTOP-RUQFT3OP:6006>” 可以看到模型结构，如图 12-5 所示。

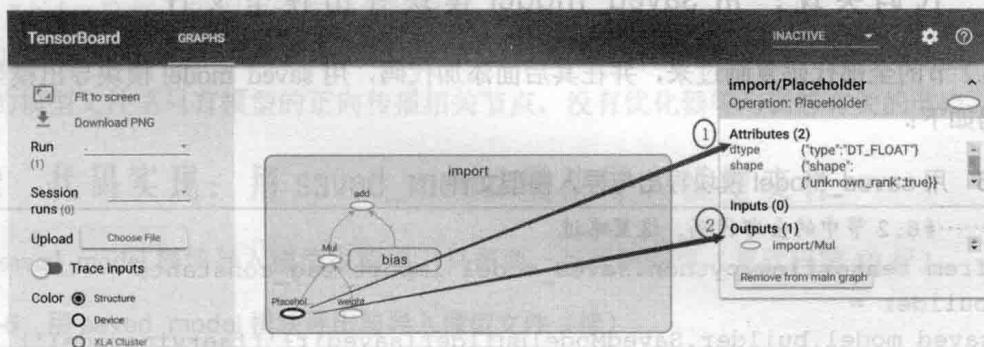


图 12-5 在 TensorBoard 中查看网络结构

单击图 12-5 中任意一个节点，都可以看到具体的属性信息。以图 12-5 中左下方节点为例：

- 该节点的属性 Attributes 是 2，表示有两个属性（见图 12-5 中标注 1 的内容）。
- 该节点的输入节点 Inputs 是 0，表示没有输入节点（见标注 2 的内容）。
- 该节点的输出节点 Outputs 是 1，表示有 1 个输出节点（见标注 2 的内容）。



提示：

在 Windows 系统中，TensorBoard 工具的运行并不是太稳定。有时在浏览器中会弹出类似“xxx 拒绝了我们的连接请求”这样的提示，提示无法访问日志结果。在这种情况下，可以尝试将访问地址改成 localhost 或 127.0.0.1。

例如：

<http://localhost:6006>

<http://127.0.0.1:6006>

如果还是访问不了，则可以尝试将所有的安全软件退出，并关闭 Windows 系统自带的防火墙，再运行 TensorBoard 工具。

12.5 实例 64：用 saved_model 模块导出与导入模型文件

用 saved_model 模块生成的是一种冻结图文件。与 12.3 节的冻结图文件不同之处是，用 saved_model 模块生成的模型文件集成了打标签操作，可以被更方便地部署在生产环境中。

实例描述

开发一个模型，让模型在一组混乱的数据集中找到 $y \approx 2x$ 的规律。在模型训练好之后：

- (1) 用 saved_model 模块生成适用于 TF Serving 的模型。
- (2) 比较该模型与 12.3 节中生成的冻结图文件的区别。
- (3) 通过编写代码载入该模型，并进行数据预测。

本实例在 6.2 节的模型上面做简单改进，并完成模型的生成与载入功能。具体如下：

12.5.1 代码实现：用 saved_model 模块导出模型文件

将 6.2 节的全部代码复制过来，并在其后面添加代码，用 saved_model 模块导出模型文件。具体代码如下：

代码 12-6 用 saved_model 模块导出与导入模型文件

```

01     .....#6.2 节中的全部代码，这里略过
02     from tensorflow.python.saved_model import tag_constants
03     builder =
04         tf.saved_model.builder.SavedModelBuilder(savedir+'tfservingmodel')
05         #将节点的定义和值加到 builder 中
06         builder.add_meta_graph_and_variables(sess, [tag_constants.SERVING])
07         builder.save()

```

上面代码的具体解读如下。

- 代码第 2 行：载入了 tag_constants 库。
- 代码第 3 行：将模型的所在路径传入 tf.saved_model.builder.SavedModelBuilder 函数中，生成 builder 对象。
- 代码第 5 行：将图中的节点和值传入 builder 对象中。其中第 2 个参数是字符串类型，代表标签。该参数需要与载入模型时的标签相对应。
- 代码第 6 行：将 builder 对象中的内容保存到文件中。

代码运行后，输出如下结果：

```
INFO:tensorflow:SavedModel written to: b'log/tfservingmodel\\saved_model.pb'
```

在输出信息的同时，程序会在 log 文件夹下生成模型文件，如图 12-6 所示。

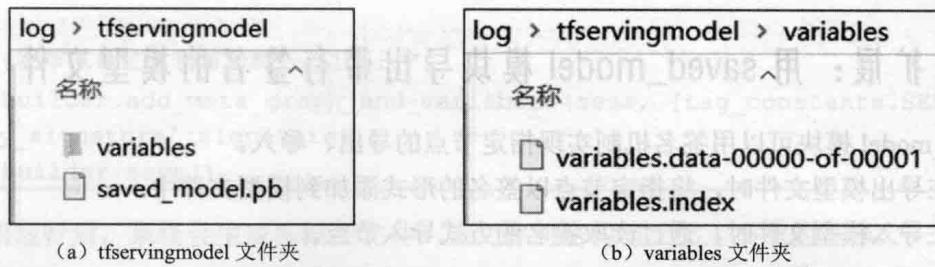


图 12-6 saved_model 生成线性回归模型

从图 12-6 (a) 中可以看到, tf-servingmodel 文件夹包含了一个文件和一个文件夹:

- 文件 saved_model.pb 是模型的定义文件。
- 文件夹 variables 中放置了具体的模型文件。

图 12-6 (b) 中可以看到, variables 文件夹包含了两个模型文件:

- *.data-00000-of-00001 文件, 模型中参数的值。
- *.index 文件: 模型中节点符号的定义。

可以看到, variables 文件夹中的模型结构与检查点文件的结构完全一样。只不过, 本实例所生成的模型文件里只有模型的正向传播相关节点, 没有优化器等与训练有关的节点。

12.5.2 代码实现: 用 saved_model 模块导入模型文件

用 saved_model 模块导入模型很简单, 只需要一行代码即可(见代码第 10 行)。

代码 12-6 用 saved_model 模块导出与导入模型文件 (续)

```

07 tf.reset_default_graph()
08
09 with tf.Session() as sess:
10     meta_graph_def = tf.saved_model.loader.load(sess,
11         [tag_constants.SERVING], savedir+'tf-servingmodel')
12     my_graph = tf.get_default_graph() #获得当前图
13     result = my_graph.get_tensor_by_name('add:0')#获得当前图中的 z 赋值给 result
14     x = my_graph.get_tensor_by_name('Placeholder:0')#获得当前图中的 X 赋值给 x
15     y = sess.run(result, feed_dict={x: 5})#传入 5, 进行预测
    print(y)

```

在代码第 10 行中, 用 tf.saved_model.loader.load 方法恢复模型。其中的参数说明如下:

- 第 1 个参数是会话。
- 第 2 个参数是标签, 必须要与生成模型时的一致。
- 第 3 个参数是模型的路径。

代码运行后, 输出如下结果:

```

INFO:tensorflow:Restoring parameters from
b'log/tf-servingmodel\\variables\\variables'
[10.140872]

```

12.5.3 扩展：用 saved_model 模块导出带有签名的模型文件

saved_model 模块可以用签名机制实现指定节点的导出、导入。

- (1) 在导出模型文件时，将指定节点以签名的形式添加到模型文件中。
- (2) 在导入模型文件时，通过读取签名的方式导入节点。



提示：

saved_model 模块可以给导出的模型添加多个标签。每个标签的结构都由输入节点、输出节点、标签名称 3 部分组成。并且，输入节点、输出节点的名字可以任意指定。

在使用模型文件时，通过不同的标签可以取到不同的输入节点、输出节点名字，并根据具体的名字取出张量，再来调用模型。

saved_model 模块的签名机制可以使导出的模型支持不同场景，并按照不同输入节点名称、输出节点名称进行部署。

1. 导出带有签名的模型文件

编写代码，按照以下步骤导出带有签名的模型文件：

- (1) 用 saved_model 模块的 builder.SavedModelBuilder 类实例化一个 builder 对象。
- (2) 构建标签的输入节点 inputs。该输入节点的名字为“input_x”。该名字是模型文件中输入节点的名字（可以任意取名）。
- (3) 构建标签的输出节点 outputs。该输入节点的名字为“output”。
- (4) 调用 build_signature_def 函数，并将标签的输入节点、输出节点和标签的名字(sig_name)传入，生成具体的标签。
- (5) 用 builder 对象的 add_meta_graph_and_variables 方法将标签添加到模型中。
- (6) 调用 builder 对象导出带有标签的模型文件。

具体代码如下：

代码 12-7 用 saved_model 模块生成与载入带签名的模型

```

01     from tensorflow.python.saved_model import tag_constants
02     builder =
03         tf.saved_model.builder.SavedModelBuilder(savedir+'tfservingmodel')
04
05     # 定义输入签名, X 为输入 tensor
06     inputs = {'input_x': tf.saved_model.utils.build_tensor_info(X)}
07
08     # 定义输出签名, z 为最终需要的输出结果 tensor
09     outputs = {'output' : tf.saved_model.utils.build_tensor_info(z)}
10
11     signature =
12         tf.saved_model.signature_def_utils.build_signature_def(inputs, outputs,
13         'sig_name')
```

```

10 #将节点的定义和值加到 builder 中
11 builder.add_meta_graph_and_variables(sess, [tag_constants.SERVING],
12 {'my_signature':signature})
13 builder.save()

```

代码运行后，系统会生成冻结图文件。该文件的结构与 12.5.1 小节的一样。

提示：

需要将 12.5.1 小节例子中生成的模型文件删掉，才可以运行本节代码，否则会报错，说已经存在该模型文件。

2. 导入模型文件，并根据签名找到网络节点

编写代码实现如下步骤：

- (1) 用 saved_model 模块中的 loader.load 方法导入冻结图文件。
- (2) 用 signature_def 方法从导入的模型文件中取出签名。
- (3) 以字典取值的方式取出输入、输出节点。
- (4) 向模型注入数据，并输出结果。

具体代码如下：

代码 12-7 用 saved_model 模块生成与载入带签名的模型（续）

```

14 tf.reset_default_graph()
15
16 with tf.Session() as sess:
17     meta_graph_def = tf.saved_model.loader.load(sess,
18         [tag_constants.SERVING], savedir+'tfervingmodel')
19     #从 meta_graph_def 中取出 SignatureDef 对象
20     signature = meta_graph_def.signature_def
21     #从 signature 中找出具体输入输出的 tensor name
22     x = signature['my_signature'].inputs['input_x'].name
23     result = signature['my_signature'].outputs['output'].name
24
25     y = sess.run(result, feed_dict={x: 5})#传入 5，进行预测
26     print(y)

```

代码运行后，输出如下结果：

```
[10.140872]
```

从结果中可以看到，程序成功导入模型文件，并能够进行预测。

12.6 实例 65：用 saved_model_cli 工具查看及使用 saved_model 模型

实例描述

在命令行中，用 saved_model_cli 工具查看和使用 12.5 节生成的 saved_model 模型。具体要求如下：

- (1) 找出模型中的 signature、输入、输出节点等相关信息。
- (2) 以命令行的方式向模型输入数据，使其运行并输出结果。

saved_model_cli 工具共有两个主要的参数。

- show 参数：侧重用于查看模型中的信息。
- run 参数：侧重于运行模型。

12.6.1 用 show 参数查看模型

本节使用的模型为 12.5 节所生成的模型文件。以路径 G:\python3\log\tservingmodel 为例。具体代码如下：

1. 查看模型文件中的签名

- (1) 在命令行中查看模型文件中的 tag（标签）。具体命令如下：

```
saved_model_cli show --dir G:\python3\log\tservingmodel
```

该命令执行后，可以看到如下结果输出：

```
The given SavedModel contains the following tag-sets:  
serve
```

输出结果的最后一行是 serve，表示模型中的 tag（标签）名字。该名字对应于 12.5.3 节中的代码第 12 行（tag_constants.SERVING 字符串）。

- (2) 查看 tag 下的签名。具体命令如下：

```
saved_model_cli show --dir G:\python3\log\tservingmodel --tag_set serve
```

该命令执行后，输出如下内容：

```
The given SavedModel MetaGraphDef contains SignatureDefs with the following keys:  
SignatureDef key: "my_signature"
```

输出结果的最后一行是 my_signature。该值对应于 12.5.3 小节中的代码第 12 行签名字典中的 key 值“my_signature”。

2. 查看模型文件中输入、输出节点的名称

在命令行中，可以用 saved_model_cli show 工具中的“--signature_def”参数查看模型的输

入、输出节点名称。具体命令如下：

```
saved_model_cli show --dir G:\python3\log\tservingmodel --tag_set serve
--signature_def my_signature
```

该命令执行后，输出如下内容：

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['my_signature']:
The given SavedModel SignatureDef contains the following input(s):
  inputs['input_x'] tensor_info:
    dtype: DT_FLOAT
    shape: unknown_rank
    name: Placeholder:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['output'] tensor_info:
    dtype: DT_FLOAT
    shape: unknown_rank
    name: add:0
Method name is: sig_name
```

从上面的输出内容可以看出，模型输入的节点张量为 `input_x`（见输出结果的第4行），输出的节点张量为 `output`（见输出结果的第9行）。

3. 查看模型文件中的全部信息

在命令行中，可以用 `saved_model_cli show` 工具中的“`--all`”参数查看模型文件中的全部信息。具体命令如下：

```
saved_model_cli show --dir G:\python3\log\tservingmodel --all
```

该命令执行后，输出的结果与本节“2. 查看模型文件中的输入、输出节点名称”中的输出结果一致。（该命令可以将模型文件中所有的标签都打印出来。）

12.6.2 用 `run` 参数运行模型

用 `saved_model_cli` 工具的 `run` 参数时，需要先指定好模型的路径、`tag`（标签）及签名，再往模型里面输入数据，并运行结果。

在输入数据部分，可以用参数来指定不同的输入方式。

- `--inputs`：后面跟具体的文件。文件类型支持 numpy 文件(`npy`、`npz`)和 pickle 文件(`plk`)。
- `--input_exprs`：指定某个变量，向模型注入数据。
- `--input_examples`：用字典向模型注入数据。

以“`--input_exprs`”为例，具体命令如下：

```
saved_model_cli run --dir G:\python3\log\tservingmodel --tag_set serve
--signature_def my_signature --input_exprs "input_x=4.2"
```

输出结果为：

[8.522742]

更多使用方式可以参考 TensorFlow 中的源码。具体路径如下：

```
Anaconda3\lib\site-packages\tensorflow\python\tools\saved_model_cli.py
```

12.6.3 扩展：了解 scan 参数的黑名单机制

在 TensorFlow 的每个版本中，都会有一个黑名单列表_OP_BLACKLIST。该黑名单中定义了当前版本中不推荐使用的 OP（操作符），即这些 OP 有可能会使当前版本出现性能或兼容问题。例如：TensorFlow 1.10 版本的 OP 黑名单为 WriteFile、ReadFile 操作符。

在 saved_model_cli 工具中，还可以用参数 scan 扫描模型中是否存在被 TensorFlow 当前版本纳入黑名单的 OP（操作符）。这样可以提前了解模型文件与 TensorFlow 当前版本的兼容性。

12.7 实例 66：用 TF-Hub 库导入、导出词嵌入模型文件

在 5.5 节中介绍了用 TF-Hub 库对模型进行微调的方法。本节将基于 5.5 节实现导入、导出支持 TF-Hub 库的模型文件。

实例描述

模拟一个训练好的词嵌入文件。用 TF-Hub 库将词嵌入文件包装成模型，并导出。再用 TF-Hub 库将导出的模型载入，并用该模型进行词嵌入的转换。

本实例将介绍词嵌入模型的使用方法。在样本不充足的情况下，使用已经训练好的词嵌入模型可以增加模型的泛化性。



提示：

如何训练词嵌入模型，可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 9.7 节 word2vec 的训练方法。

12.7.1 代码实现：模拟生成通用词嵌入模型

通用的词嵌入模型常以 key-value 的格式保存，即把词所对应的向量一一列出来。这种方式具有更好的通用性，它可以不依赖任何框架。

编写代码，模拟一个已经训练好的词嵌入模型。具体代码如下：

代码 12-8 TF-Hub 模型例子

```
01 import os
02 import shutil
03 import tempfile
04 import numpy as np
05 import tensorflow as tf
06 import tensorflow_hub as hub
```

```

08 # 定义词嵌入内容
09 _MOCK_EMBEDDING = "\n".join([
10     *[f"{word} {float_val} {float_val} {float_val}" for word, float_val in [
11         ("cat", 1.11), ("dog", 1.2), ("mouse", 0.5), ("ocean", 0.6)
12     ]])
13 with tf.gfile.GFile(_embedding_file_path, mode="w") as f:
14     f.write(_MOCK_EMBEDDING)

```

代码第 9 行是模拟的词嵌入数据。每个词对应于 3 个维度的特征值。

代码运行后可以看到，在本地目录下生成一个名为 `mock_embedding_file.txt` 的文件。接下来将该词嵌入文件转化成具体可用的模型文件。

12.7.2 代码实现：用 TF-Hub 库导出词嵌入模型

在 TF-Hub 库中，所有的模型操作都是通过 `ModuleSpec` 类型对象实现的。

定义函数 `make_module_spec`，用来生成 `ModuleSpec` 类型对象。在 `make_module_spec` 函数中支持的参数有：

- 字典文件 (`vocabulary_file`)。
- 字典大小 (`vocab_size`)。
- 词嵌入的全部特征数据 (`embeddings_dim`)。
- 未识别的预留字符个数 (`num_oov_buckets`)。
- 是否支持预处理 (`preprocess_text`)。

在 `make_module_spec` 函数内部，可用以下两种内嵌函数构建模型。

- 内嵌函数 `module_fn`: 创建一般模型。该函数可以将输入的单个词转为词嵌入。
- 内嵌函数 `module_fn_with_preprocessing`: 创建支持预处理的模型。该函数可以对输入的多个词做符号过滤、对齐处理，还可以对其生成的词嵌入做归约运算。

具体代码如下：

代码 12-8 TF-Hub 模型例子（续）

```

15 def parse_line(line):#解析词嵌入文件中的一行
16     columns = line.split()
17     token = columns.pop(0)
18     values = [float(column) for column in columns]
19     return token, values
20
21 def load(file_path, parse_line_fn):#按照指定的方法加载词嵌入
22     vocabulary = []
23     embeddings = []
24     embeddings_dim = None
25     for line in tf.gfile.GFile(file_path):
26         token, embedding = parse_line_fn(line)
27         if not embeddings_dim:
28             embeddings_dim = len(embedding)

```

```

29     elif embeddings_dim != len(embedding):
30         raise ValueError(
31             "Inconsistent embedding dimension detected, %d != %d for token %s",
32             embeddings_dim, len(embedding), token)
33     vocabulary.append(token)
34     embeddings.append(embedding)
35     return vocabulary, np.array(embeddings)
36
37 #返回TF-Hub的spec模型
38 def make_module_spec(vocabulary_file, vocab_size, embeddings_dim,
39                      num_oov_buckets, preprocess_text):
40     def module_fn():                      #正常的、不带预处理功能的模型
41         tokens = tf.placeholder(shape=[None], dtype=tf.string, name="tokens")
42         embeddings_var = tf.get_variable(  #定义词嵌入变量
43             initializer=tf.zeros([vocab_size + num_oov_buckets,
44                                   embeddings_dim]),
45             name='embedding', dtype=tf.float32)
46
47         lookup_table = tf.contrib.lookup.index_table_from_file(
48             vocabulary_file=vocabulary_file,
49             num_oov_buckets=num_oov_buckets)
50
51         ids = lookup_table.lookup(tokens)
52         combined_embedding = tf.nn.embedding_lookup(params=embeddings_var,
53                                                       ids=ids)
54
55     hub.add_signature("default", {"tokens": tokens},
56                      {"default": combined_embedding})
57
58     def module_fn_with_preprocessing():    #定义函数，创建带有预处理功能的网络模型
59         sentences = tf.placeholder(shape=[None], dtype=tf.string,
60                                      name="sentences")
61
62         #用正则表达式删除特殊符号
63         normalized_sentences = tf.regex_replace(
64             input=sentences, pattern=r"\pP", rewrite="")
65
66         #按照空格分词得到稀疏矩阵
67         tokens = tf.string_split(normalized_sentences, " ")
68
69         embeddings_var = tf.get_variable(  #定义词嵌入变量
70             initializer=tf.zeros([vocab_size + num_oov_buckets,
71                                   embeddings_dim]),
72             name='embedding', dtype=tf.float32)
73
74         #用字典将词变为词向量
75         lookup_table = tf.contrib.lookup.index_table_from_file(
76             vocabulary_file=vocabulary_file,
77             num_oov_buckets=num_oov_buckets)
78
79
80
81
82
83
84
85
86
87
88
89
90
91

```

```

72     # 将稀疏矩阵用词嵌入转化
73     sparse_ids = tf.SparseTensor(
74         indices=tokens.indices,
75         values=lookup_table.lookup(tokens.values),
76         dense_shape=tokens.dense_shape)
77
78     # 为稀疏矩阵添加空行
79     sparse_ids, _ = tf.sparse_fill_empty_rows(
80         sparse_ids, lookup_table.lookup(tf.constant("")))

81     # 结果进行平方和再开根号的规约计算
82     combined_embedding = tf.nn.embedding_lookup_sparse(
83         params=embeddings_var, sp_ids=sparse_ids,
84         sp_weights=None, combiner="sqrt")
85
86     # 添加签名
87     hub.add_signature("default", {"sentences": sentences},
88                       {"default": combined_embedding})
89
90
91     if preprocess_text:
92         return hub.create_module_spec(module_fn_with_preprocessing)
93     else:
94         return hub.create_module_spec(module_fn)

```

代码第 15、21 行是两个辅助函数——`parse_line` 与 `load`，这两个函数用来读取生成好的模拟词嵌入文件。

代码第 55 行是生成预处理模型的关键函数。该函数的步骤如下：

- (1) 用正则表达式对输入进行字符过滤，去掉不符合要求的字符。
- (2) 将其用空格分开，得到稀疏矩阵形式的数组。
- (3) 定义变量用来存放所有的词嵌入，以便查找。
- (4) 将稀疏矩阵数组中的词转为词向量。

(5) 用 `tf.nn.embedding_lookup_sparse` 函数进行基于稀疏矩阵的词嵌入转化。其中的参数 `combiner` 代表规约运算的方式。这里传入的是 `sqrt` 算法，表示对一个句子中的多个词嵌入结果进行平方加和再开根号运算（见代码 83 行）。

- (6) 添加签名，并用 `create_module_spec` 函数返回模型的 `ModuleSpec` 对象。

在代码第 88 行，添加签名是个很重要的环节。整个 TF-Hub 库都是通过签名与模型进行交互的。

在本实例中统一使用默认的 `default` 作为签名。如果签名不是 `default`，则需要在调用模型时进行指定。

另外，如果模型的输入、输出是多个值，则需要将其用字典的形式进行传递。

从图 12-7 中可以看到，相比 `serve_model` 模块生成的模型文件，TF-Hub 模型文件多出了

12.7.3 代码实现：导出 TF-Hub 模型

编写代码实现如下步骤：

- (1) 将字典保存到文件中。
- (2) 将字典文件名传入 make_module_spec 函数中，生成 ModuleSpec 对象。
- (3) 用 hub.Module 函数将 ModuleSpec 对象转化成真正的模型 m。
- (4) 用 m 的 export 方法将模型保存到本地。

具体代码如下：

代码 12-8 TF-Hub 模型例子（续）

```

95 # 导出 TF-Hub 模型
96 def export(export_path, vocabulary, embeddings, num_oov_buckets,
97             preprocess_text):#模型是否支持预处理
98     #建立临时文件夹
99     tmpdir = tempfile.mkdtemp()
100    #建立目录
101    vocabulary_file = os.path.join(tmpdir, "tokens.txt")
102    #将字典 vocabulary 写入文件
103    with tf.gfile.GFile(vocabulary_file, "w") as f:
104        f.write("\n".join(vocabulary))
105    spec = make_module_spec(vocabulary_file, len(vocabulary),
106                             embeddings.shape[1], num_oov_buckets, preprocess_text)
107    try:
108        with tf.Graph().as_default():
109            #将 spec 转化为真正的模型
110            m = hub.Module(spec)
111            p_embeddings = tf.placeholder(tf.float32)
112            #为定义好的词嵌入赋值（恢复模型）
113            load_embeddings = tf.assign(m.variable_map['embedding'],
114                                         p_embeddings)
115            with tf.Session() as sess:
116                #以注入的方式将模型权重恢复到模型中去
117                sess.run([load_embeddings], feed_dict={p_embeddings: embeddings})
118                m.export(export_path, sess)#生成模型
119            finally:
120                shutil.rmtree(tmpdir)
121
122    os.makedirs('./emb', exist_ok=True)          #创建模型目录
123    os.makedirs('./peremb', exist_ok=True)
124
125
126
127
128

```

```

129 export_module_from_file(          # 生成一个词嵌入模型
130     embedding_file=_embedding_file_path,
131     export_path='./emb',
132     parse_line_fn=parse_line,
133     num_oov_buckets=1,
134     preprocess_text=False)
135
136 # 生成一个带有预处理的词嵌入模型
137 export_module_from_file(
138     embedding_file=_embedding_file_path,
139     export_path='./peremb',
140     parse_line_fn=parse_line,
141     num_oov_buckets=1,
142     preprocess_text=True)

```

代码第 115 行是恢复模型权重的操作。用 `tf.assign` 函数将词嵌入赋值给模型中的张量。需要注意的是，模型中的张量是通过 `m.variable_map` 字典中的名字得到的。这个名字是在代码第 42 和第 64 行定义张量 `embeddings_var` 时指定的。



提示：

如果代码第 52、87 行添加签名时指定的签名不是 `default`，则在代码第 112 行获取模型时，还需要指定具体的签名才行。

另外，如果模型的输入、输出节点有多个，并以字典的形式传入，还需要指定字典类型。

代码如下：

```
outputs = hub_module("输入字典", signature="自定义签名", as_dict=True)
```

该代码执行后，得到的 `outputs` 是一个字典。可以通过字典里的 `key` 来获取对应的结果。

代码如下：

```
features = outputs["key"]# 通过字典的 key 取出结果。
```

代码第 129、137 行，分别生成了一个普通的词嵌入模型和一个带有预处理的词嵌入模型。

代码执行后，在本地目录下生成两个文件夹 `emb` 与 `peremb`。以 `peremb` 为例，其文件结构如图 12-7 所示。

python3 > peremb

名称

- assets
- variables
- saved_model.pb
- tfhub_module.pb

图 12-7 生成的 TF-Hub 模型文件

从图 12-7 中可以看到，相比 `saved_model` 模块生成的模型文件，TF-Hub 模型文件多出了

assets 文件夹和 tflib_module.pb 文件：

- assets 文件夹中是字典文件。
- tflib_module.pb 文件中是 TF-Hub 库可以独立使用的模型文件。

12.7.4 代码实现：用 TF-Hub 库导入并使用词嵌入模型

编写代码实现如下步骤：

(1) 将 12.7.3 小节生成的两个模型目录分别传入 hub.Module 函数里，实现模型文件的导入。

(2) 定义两个模拟的字符串列表数据，传入模型进行计算。

具体代码如下：

代码 12-8 TF-Hub 模型例子（续）

```

143 with tf.Graph().as_default():
144     hub_module = hub.Module('./emb')          #载入模型
145     tokens = tf.constant(["cat", "lizard", "dog"])#定义模拟数据
146
147     perhub_module = hub.Module('./peremb')
148     pertesttokens = tf.constant(["cat", "cat cat", "lizard. dog", "cat? dog",
149     ""])
150     embeddings = hub_module(tokens)           #将数据传入模型
151     perembeddings = perhub_module(pertesttokens)
152     with tf.Session() as session:             #启动会话
153         session.run(tf.tables_initializer())    #初始化
154         session.run(tf.global_variables_initializer())
155         print(session.run(embeddings))        #输出计算结果
156         print(session.run(perembeddings))

```

上面代码执行后，输出如下结果：

```

[[1.11 2.56 3.45] [0. 0. 0.] [1. 2. 3.]]
[[1.11      2.56      3.45      ]
 [1.5697771 3.6203866 4.879037]
 [0.70710677 1.4142135 2.1213205]
 [1.4919955 3.224407  4.5608387]
 [0.          0.          0.        ]]

```

输出结果的第 1 行是不带预处理模型的输出内容，其中有 3 个列表。每个列表是传入词的词嵌入结果。

输出结果的倒数 5 行是带预处理功能模型的输出内容。它们分别是 5 个句子对应的词向量经过 sqrtn 算法规约计算后的结果。

第 13 章

部署 TensorFlow 模型——模型与项目的深度结合

深度学习模型本质上是工程项目中的算法模块，最终还需要被部署到生产环境中，与工程程序结合起来使用。本章将通过实例介绍部署模型的方法。

13.1 快速导读

在学习实例之前，有必要了解一下部署模型方面的基础知识。

13.1.1 什么是 gRPC 服务与 HTTP/REST API

gRPC 服务、HTTP/REST API 是 TF Serving 模块对外支持服务的两种通信技术。通过这两种通信技术，可以远程使用 TensorFlow 模型。

1. RPC

了解 gRPC 之前，先来介绍一下远程过程调用协议（Remote Procedure Call Protocol, RPC）。

现有两台服务器（服务器 A、服务器 B）。一个应用程序部署在 A 服务器上，它要去调用 B 服务器上的函数或方法。由于 A 服务器上的应用程序和 B 服务器上的应用程序不在一个内存空间，需要用网络将 A 服务器上的调用语义传递到 B 服务器上，才可以调用 B 服务器上的应用程序。

2. gRPC

gRPC 是谷歌发布的首款基于 Protocol Buffers（Google 公司开发的一种数据描述语言）的 RPC 框架，是一个高性能、开源、通用的 RPC 框架，面向移动和 HTTP 2.0 设计。

gRPC 具有双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使得其在移动设备上表现更好，更省电、省空间。

当前 gRPC 可以分为 gRPC、gRPC-Java、gRPC-Go 三个版本。

- gRPC 版本支持 C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C# 语言。
- gRPC-Java 版本支持 Java 语言。
- gRPC-Go 版本支持 Go 语言。

用 gRPC 版本实现 gRPC 服务的例子，可以参考本书 13.8 节。

3. gRPC 的服务

gRPC 的服务有以下 4 种。

- 单项 RPC：客户端发送一个请求给服务端，从服务端获取一个应答。它类似于普通的函数调用。
- 服务端流式 RPC：客户端发送一个请求给服务端，可以从服务端获取一个可读数据流。客户端从数据流里一直读取数据，直到没有更多消息为止。
- 客户端流式 RPC：客户端发送一个请求给服务端，并从服务端获取一个可写数据流。客户端向数据流里一直写入数据，直到将数据全部写入，然后等待服务端读取这些消息并返回应答。
- 双向流式 RPC：客户端与服务端都可以通过读写数据流来发送数据。这两个数据流操作是相互独立的，客户端和服务端可以按其指定的任意顺序读写。例如，服务端可以在写应答前等待所有的客户端消息，也可以先读一个消息再写一个消息；还可以采用读写相结合的其他方式。每个数据流里消息的顺序会保持不变。

4. HTTP/REST API

HTTP/REST API 主要是让远程服务方式以 HTTP 的 URL 通信方式对外暴露出来。这使得访问远程服务就像访问 URL 一样方便。

13.2 节是一个用 HTTP/REST API 方式使用 TF Serving 服务的例子。

13.1.2 了解 TensorFlow 对移动终端的支持

用 TensorFlow 训练好的模型，可以运行在安卓、苹果系统的移动终端上。配合 TF_Lite 模块，模型可以运行得更加流畅。TF_Lite 模块与模型的部署关系如图 13-1 所示。

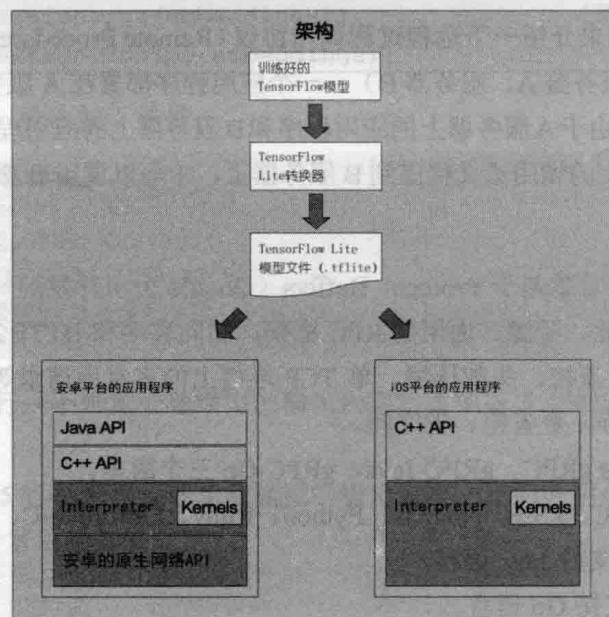


图 13-1 TF-Lite 模块与模型的部署关系

另外，在 GitHub 网站上也提供了大量的帮助文档，供用户学习使用。具体链接如下：

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/g3doc>

在 12.1.5 小节简单介绍过 TF-Lite 模块转化冻结图的方法。13.3 节还会通过一个在安卓系统上部署的例子详细介绍具体操作。

13.1.3 了解树莓派上的人工智能

树莓派（Raspberry Pi）是一个采用 ARM 架构的开放式嵌入式系统，外形小巧，却具有强大的系统功能和接口资源。它由英国的慈善组织“Raspberry Pi 基金会”开发。

迄今为止树莓派已经有多个型号，见表 13-1。

表 13-1 树莓派型号

项 目	Raspberry Pi 2 Model B	Raspberry Pi Zero	Raspberry Pi 3 Model B
发布时间	2015-02	2015-11	2016-02
Soc (系统级芯片)	BCM2836	BCM2835	BCM2837
CPU	ARM Cortex-A7 900MHz, 单核	ARM 1176JZF-S 核心 700MHz, 单核	ARM Cortex-A53 1.2GHz, 四核
GPU	Broadcom 公司的 VideoCore IV 图像处理器 加载 OpenGL ES 2.0 驱动程序 支持 1080p 30fps, h.264/MJPEG-4 AVC 高清解码		
RAM	1GB	512MB	1GB
USB 接口	USB 2.0×4	Micro USB 2.0×1	USB 2.0×4
SD 卡接口	Micro SD 卡接口	Micro SD 卡接口	Micro SD 卡接口
网络接口	10/100 以太网接口 (RJ45 接口)	无	10/100 以太网接口 (RJ45 接口)

1. 树莓派的主流型号

目前市场的主流型号为 3 Model B。该型号配备一枚博通（Broadcom）生产的 ARM 架构 4 核 1.2GHz BCM2837 处理器、1GB LPDDR2 内存，使用 SD 卡当作储存媒体，且拥有 1 个 Ethernet 接口、4 个 USB 接口，以及 HDMI（支持声音输出）和音频接口。除此之外，它还支持蓝牙 4.1 和 WI-FI，可以运行 Linux 系统和 Windows IOT 系统，可以应用在嵌入式和物联网领域，完成一些特定的功能。

2. 树莓派上的人工智能

树莓派硬件的运算能力有限，很难在其上直接运行较大的复杂 AI 模型。

如果在树莓派上运行 AI 模型，需要做二次优化。一般会有两个主要的大方向：修改模型、加速框架。

- 修改模型：用更低的权重精度和权重剪枝。

- 加速框架：通过计算技巧（例如：优化矩阵之间的乘法），或者使用 GPU、DSP 或 FPGA 等硬件来加速框架的执行时间。

本书主要侧重于修改模型。在 13.5 节中将介绍一个把优化后的模型运行在树莓派上的例子。

13.2 实例 67：用 TF_Serving 部署模型并进行远程使用

训练好的模型在使用过程中有多种场景。TensorFlow 中提供了一种 TF_Serving 接口，可以将模型部署在远端服务器上，并以服务的方式对外提供接口。

实例描述

用 TF_Serving 接口将一个线性的回归模型部署在 Linux 服务器上，让模型以服务的形式对外提供接口。用 gRPC 与 HTTP/REST API 远程访问模型，使其计算出结果，并返回结果。

本节使用的线性回归模型与 12.5 节一致。在准备好模型文件之后，还需要安装 TF_Serving 模块。

13.2.1 在 Linux 系统中安装 TF_Serving

在 Linux 系统中在线安装 TF_Serving 时，因为要使用“apt-get”命令从 storage.googleapis.com 下载对应的软件包，所以必须保证本机 IP 所在的网络可以到达 storage.googleapis.com 域名地址（可以使用 ping 命令进行测试）。具体操作可以分为以下几个步骤。

1. 检测 Linux 版本

以作者的本地机器为例，输入命令后显示如下：

```
root@user-NULL:~# cat /proc/version
Linux version 4.13.0-36-generic (build@lgw01-amd64-033) (gcc version 5.4.0 20160609
(Ubuntu 5.4.0-6ubuntu1~16.04.9)) #40~16.04.1-Ubuntu SMP Fri Feb 16 23:25:58 UTC 2018
```

2. 添加下载地址

输入如下命令，向“apt-get”添加 TF_Serving 安装包的下载地址：

```
echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server tensorflow-model-server-universal" | sudo tee
/etc/apt/sources.list.d/tensorflow-serving.list && \
curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-serving.release.pu
b.gpg | sudo apt-key add -
```

3. 更新 apt-get

通过以下命令切换到 sudo 账户，并升级 apt-get：

```
sudo su
sudo apt-get update
```

中，在运行过程中，有可能会提示“没有数字签名”错误。可以忽略该提示，不影响正常使用。

4. 下载 tensorflow-model-server

使用以下命令进行 tensorflow-model-server 软件包的安装：

```
apt-get install tensorflow-model-server
```

在安装过程中，仍然会提示“没有数字签名是否允许安装”。直接输入“y”即可。



提示：

默认的 tensorflow-model-server 版本需要安装在支持 SSE4 和 AVX 指令集的服务器上。如果本地的机器过于老旧不支持该指令集，需要安装 tensorflow-model-server-universal 版本。具体命令为：

```
apt-get install tensorflow-model-server-universal
```

如果已经安装好 tensorflow-model-server，则需要将 tensorflow-model-server 卸载后才能再安装 tensorflow-model-server-universal。卸载 tensorflow-model-server 的命令如下：

```
apt-get remove tensorflow-model-server
```

如果在已有的 tensorflow-model-server 上做更新，可以输入如下命令：

```
apt-get upgrade tensorflow-model-server
```

13.2.2 在多平台中用 Docker 安装 TF_Serving

Docker 作为一个加独立的跨平台工具，可以将应用环境与开发环境独立开来。它可以将所有的环境、配置、代码，甚至 Linux 底层，都打包在一起，使用者不需要考虑新的服务器环境是否兼容。这给工程部署带来了方便。

1. 安装 Docker

Docker 有 CE（免费版）和 EE（付费版）两个版本。可以安装在各个主流操作系统之上。关于 Docker 的安装方法，可以参考官方帮助文档：

```
https://docs.docker.com/install/
```

如在 Windows 10 中安装，则需要额外对系统进行配置，可使用其自带的 Hyper-V（虚拟机）功能来实现：打开“控制面板”→程序→启用或关闭 Windows 功能→选中 Hyper-V。但是这个功能只在 Windows 10 的企业版有。

如果当前的 Windows 10 系统里没有 Hyper-V 选项，则需要安装 DockerToolbox。下载地址如下：

```
https://get.daocloud.io/toolbox/
```

在 Ubuntu 安装 Docker 的实例可以参考 13.5.2 小节。

2. 在 Docker 中使用 TF_Serving

在安装好 Docker 之后，可以使用以下命令下载一个带有 TF_Serving 的镜像。

```
docker pull tensorflow/serving
```

还可以手动去以下地址下载更多其他版本的镜像文件：

<https://hub.docker.com/r/tensorflow/serving/tags/>

更多操作说明可以参考官方网站的帮助文档，这里不再详述。

<https://www.tensorflow.org/serving/docker>

13.2.3 编写代码：固定模型的签名信息

在 12.5.3 小节中，用函数 `saved_model` 在模型中添加签名。为了让生成的模型支持 TF_Serving 服务，在 TensorFlow 中对模型的签名做了统一的规定。在签名中规定，模型在处理分类、预测、回归这三种任务时，必须使用对应的输入与输出接口。具体接口的定义在 `tensorflow.saved_model.signature_constants` 模块下，见表 13-2。

表 13-2 统一的签名接口规则

任 务	输入与输出
分类任务： <code>CLASSIFY_METHOD_NAME</code> ("tensorflow/serving/classify")	输入： <code>CLASSIFY_INPUTS</code> ("inputs") 输出（分类结果）： <code>CLASSIFY_OUTPUT_CLASSES</code> ("classes") 输出（分类概率）： <code>CLASSIFY_OUTPUT_SCORES</code> ("scores")
预测任务： <code>PREDICT_METHOD_NAME</code> ("tensorflow/serving/predict")	输入： <code>PREDICT_INPUTS</code> ("inputs") 输出： <code>PREDICT_OUTPUTS</code> ("outputs")
回归任务： <code>REGRESS_METHOD_NAME</code> ("tensorflow/serving/regress")	输入： <code>REGRESS_INPUTS</code> ("inputs") 输出： <code>REGRESS_OUTPUTS</code> ("outputs")

另外，还提有一个默认的接口 `DEFAULT_SERVING_SIGNATURE_DEF_KEY` ("serving_default")，用于扩展。



提示：

在表 13-2 中，任务列里的签名是必须的，且只能有这 3 种签名。如使用其他的签名则会报错误。

“输入与输出”列中的签名是可选的，可以使用其他签名，但要求服务端与客户端必须严格匹配。在没有特殊需求的情况下，建议使用规定的签名，以避免客户端与服务器名称不匹配情况的发生。

改写代码“12-7 用 `saved_model` 模块生成与载入带签名的模型.py”，并仿照 12.5.3 小节

中的方法为模型添加规定签名。具体代码如下：

代码 13-1 支持远程调用的模型

```

01 .....
02 from tensorflow.python.saved_model import tag_constants
03 builder =
04     tf.saved_model.builder.SavedModelBuilder(savedir+'tfervingmodelv1')
05     # 定义输入签名, X 为输入 tensor
06     inputs = {'input_x': tf.saved_model.utils.build_tensor_info(X)}
07     # 定义输出签名, z 是最终需要的输出结果 tensor
08     outputs = {'output' : tf.saved_model.utils.build_tensor_info(z)}
09     # 添加支持远程调用的签名
10     signature = tf.saved_model.signature_def_utils.build_signature_def(
11         inputs=inputs,
12         outputs=outputs,
13         method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME)
14
15     # 将节点的定义和值加到 builder 中, 并加入了标签
16     builder.add_meta_graph_and_variables(sess, [tag_constants.SERVING],
17     {'my_signature':signature})
17     builder.save()

```

上面的代码与代码文件“12-7 用 saved_model 模块生成与载入带签名的模型.py”只有 1 行不同——代码第 10 行用 tf.saved_model.signature_def_utils.build_signature_def 函数生成签名。向其中传入的参数需要符合表 13-2 的规范。本实例要实现的是一个预测任务，所以需要传入 PREDICT_METHOD_NAME。



提示：

预测任务是最灵活的签名方式，可以覆盖分类和回归两种任务。代码第 10 行中的 build_signature_def 函数还可以替换成更高级的接口调用，具体如下：

- 生成回归签名函数：regression_signature_def。
- 生成分类签名函数：classification_signature_def。
- 生成预测签名函数：predict_signature_def。

这 3 个函数是在 build_signature_def 函数基础上进行封装的。它们使用起来更加方便，但是灵活性会差一些。regression_signature_def 与 classification_signature_def 函数支持单一的输入，并统一按照表 13-2 中的签名规则，将其传入到张量节点中即可。具体可以参考源码定义。例如，以作者本地源码为例，路径如下：

C:\local\Anaconda3\lib\site-packages\tensorflow\python\saved_model\signature_def_utils_impl.py

代码运行之后，在本地的 log\tfervingmodelv1 下可以找到生成的模型文件。

13.2.4 在 Linux 中开启 TF_Serving 服务

在 13.2.4 小节中，生成的模型文件所在的文件夹为 `tf-serving-modelv1`。下面将该文件夹整个传到服务器上。

1. 构建模型版本号

在使用 `tensorflow_model_server` 命令之前，还需要对模型文件夹结构做一些改变。默认情况下，模型文件必须要放在具有数字命名的文件夹里，才可以被 `tensorflow_model_server` 命令启动。其中的数字代表该模型的版本号。

在 `tf-serving-modelv1` 下定义一个新的文件夹 `123456`（代表版本号），并将模型文件全部移动到 `123456` 下面。具体操作如下：

```
cd tf-serving-modelv1/
mkdir 123456
mv saved_model.pb 123456/
mv variables 123456/
```

2. 启动 gRPC 服务

直接使用 `tensorflow_model_server` 命令，并指定端口和文件路径。具体如下：

```
tensorflow_model_server --port=9000 --model_base_path= /test/tf-serving-modelv1/
```

如果看到类似如下信息，则代表服务已经启动。

```
.....tensorflow/cc/saved_model/loader.cc:259] SavedModel load for tags { serve };
Status: success. Took 37293 microseconds.
.....tensorflow_serving/core/loader_harness.cc:86] Successfully loaded servable
version {name: default version: 123456}
.....model_servers/server.cc:285] Running gRPC ModelServer at 0.0.0.0:9000 ...
```

上面是从输出结果中挑选的 3 条信息，分别以省略号开始。其中解读如下：

- 第 1 条显示结果有“`Status: success`”的信息，表示模型已经成功载入。
- 第 2 条显示结果有 `name` 和 `version` 信息，它们分别代表模型名称和版本号。在 `tensorflow_model_server` 命令中，还可以通过“`--model_name`”参数为模型指定具体名称。
- 第 3 条显示结果表示 gRPC 服务已经正常启动，监听的端口为 `9000`。

这里只是列举了 `tensorflow_model_server` 的主要参数。如想了解 `tensorflow_model_server` 中的更多参数及使用，请参考 GitHub 网站上的源代码文件。具体链接如下：

https://github.com/tensorflow/serving/blob/master/tensorflow_serving/model_servers/main.cc

更多示例和文档，也可以参考如下链接：

<https://github.com/tensorflow/serving>

https://github.com/tensorflow/serving/blob/master/tensorflow_serving/g3doc

3. 启动HTTP/REST API服务

启动HTTP/REST API服务的命令，只需要将“--port”参数换作“--rest_api_port”。其他参数和含义都完全一样。当HTTP/REST API服务启动成功后，可以在输出信息中找到如下信息：

```
.....model_servers/server.cc:301] Exporting HTTP/REST API at:localhost:8500 ...
```

上面的输出结果表示HTTP/REST API已经成功启动，监听本机的8500端口。



提示：

在tensorflow_model_server命令中，参数“--port”和“--rest_api_port”是可以同时出现的，但它们必须使用不同的端口。

4. 在后台启动服务

如想把该服务作为后台命令启动，可以在后面加上&符号，并指定输出的日志(log)文件。具体如下：

```
tensorflow_model_server --port=9000 --model_base_path=/test/tfservingmodelv1/ &>
log &
```

启动模型过程中的输出将会被保存到当前目录下的log文件中。

13.2.5 编写代码：用gRPC访问远程TF_Serving服务

用gRPC访问远程TF_Serving服务时，需要在代码中引入tensorflow-serving-api模块，来实现本机与TF_Serving服务的通信。tensorflow-serving-api模块的安装命令如下：

```
pip install tensorflow-serving-api
```

编写代码，实现如下步骤：

- (1) 在代码中引入tensorflow-serving-api中的prediction_service_pb2_grpc模块。
- (2) 实例化prediction_service_pb2_grpc.PredictionServiceStub类，得到对象stub。
- (3) 用predict_pb2.PredictRequest函数建立一个请求对象request。
- (4) 为request对象添加模型名称、签名、输入节点等信息。
- (5) 将请求对象request传入stub对象的Predict方法中，与远端服务器建立一个连接。



提示：

`prediction_service_pb2_grpc.PredictionServiceStub`类封装了多个向服务端请求的远程调用方法，其中包括：Classify(分类)、Regress(回归)、Predict(预测)。这些方法分别与13.2.3小节中表13-2中的签名接口规则相对应。

具体代码如下：

代码13-2 grpc客户端

```
01 import grpc
```

```

02 import numpy as np
03 import tensorflow as tf
04 import time
05 from tensorflow_serving.apis import predict_pb2
06 from tensorflow_serving.apis import prediction_service_pb2_grpc
07
08 def client_gRPC(data):
09     channel = grpc.insecure_channel('127.0.0.1:9000') #建立一个通道
10     #连接远端服务器
11     stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
12
13     #初始化请求
14     request = predict_pb2.PredictRequest()
15     request.model_spec.name = 'md' #指定模型名称
16     request.model_spec.signature_name = "my_signature" #指定模型签名
17     request.inputs['input_x'].CopyFrom(tf.contrib.util.make_tensor_proto
18 (data))
19     #开始调用远端服务，执行预测任务
20     start_time = time.time()
21     result = stub.Predict(request)
22
23     #输出预测时间
24     print("cost time: {}".format(time.time()-start_time))
25
26     #解析结果并返回
27     result_dict = {}
28     for key in result.outputs:
29         tensor_proto = result.outputs[key]
30         nd_array = tf.contrib.util.make_ndarray(tensor_proto)
31         result_dict[key] = nd_array
32
33     return result_dict
34
35 def main():
36     a = 4.2 #传入单个数值
37     result= client_gRPC(a)
38     print("-----单个数值预测结果-----")
39     print(list(result['output']))
40
41     #传入多个数值
42     data = np.asarray([4.2,4.0],dtype = np.float32)
43     result= client_gRPC(data)
44     print("-----多个数值预测结果-----")
45     print(list(result['output']))
46
47 #主模块运行函数
48 if __name__ == '__main__':
49     main()

```

代码中，分别传入了一个和多个数值到远端服务进行计算。在代码运行之前，需要按照13.2.4小节的内容在服务端启动gRPC服务。具体命令如下：

```
tensorflow_model_server --port=9000 --model_base_path=/home1/test/tfservingmodel1
--model_name=md --rest_api_port=8500
```

为了方便起见，直接将gRPC与REST API两个服务同时启动，分别监听9000端口与8500端口。

将代码运行后，输出如下内容：

```
花费时间: 0.18953657150268555
-----单个数值预测结果-----
[8.396306]
花费时间: 0.16954421997070312
-----多个数值预测结果-----
[8.396306, 7.9942493]
```

在输出结果中，第3行是单个数值的预测结果，最后1行是多个数值的预测结果。

实例中连接的是本机IP（见代码第26行）。在实际使用过程中，将代码第26行的本机IP地址127.0.0.1换成指定的目标服务器IP地址即可。

13.2.6 用HTTP/REST API访问远程TF_Serving服务

Web接口无疑是当今应用最广泛的接口之一。将模型提供的服务封装为以URL方式访问的形式，可以兼容更多的终端，适用于更多的场景。

使用HTTP/REST API时，要通过POST方式请求一个URL，并带上JSON数据来完成。具体的说明如下。

1. URL说明

URL地址可以分为3部分：目的IP和端口、固定的路径（/v1/models）、模型名称（md）与预测方法（predict）。其中，模型名称与预测方法需要与模型文件中的名称与预测方法严格匹配。例如：

```
http://localhost:8500/v1/models/md:predict
```

其中，localhost:8500是第1部分；v1/models是第2部分（是固定不变的）；md:predict是第3部分（md为模型名称、predict为模型的预测任务）。

如果是分类任务或回归任务，则第3部分的内容要写成“md:classify”或“md:regress”。



提示：

如果同时部署多个版本，则在使用时还需要指定版本，即在第3部分的模型名称前加上版本信息。例如：

```
http://localhost:8500/v1/models/md/versions/123456:predict
```

其中，versions/123456为版本信息，表示用123456版本的模型进行预测。

2. POST 请求中的 JSON 数据格式

在 POST 请求中的 JSON 数据需要按照模型的具体任务（分类、回归、预测）所对应的格式来构建。

(1) 对于分类和回归任务，构建的格式是一样的，具体如下：

```
{ "signature_name": 签名字符串, "context": { "公共字段名": 值或列表 }, "examples": [ { "字段名": 值或列表 } ] }
```

具体解释如下：

- `signature_name` 是模型中的签名。当服务端的模型使用默认签名时，可以不填。
- `examples` 里面可以包括多个{}，每个{}代表一个具体要预测输入样本。每个{}内部也可以有多个字段，代表输入。当多个样本具有相同的输入值时，可以将其单独提出来放到 `context` 里面。
- `context` 是可选项，代表从 `examples` 中提取出来的具有相同值的公共输入字段，可以有多个。

(2) 对于预测任务，构建的格式如下：

```
{"signature_name": 签名字符串, "instances": 值或列表, "inputs": 值或列表}
```

具体解释如下：

- `signature_name` 是模型中的签名。如果服务端模型使用的是默认签名，则可以不填。
- `instances` 是输入的样本字段。如果只有一个输入列，则直接填值。如果有多个输入列，则可以用 JSON 格式继续扩展填充内容。预测的结果将以行的形式来显示。
- `inputs` 也是输入的样本字段。与 `instances` 不同的是，用 `inputs` 预测的结果将以列的形式显示。



提示：

在使用时，`inputs` 与 `instances` 不可同时使用。

3. POST 返回中的 JSON 数据的格式

不同的任务返回的 JSON 格式是不同的，具体如下：

(1) 分类任务的返回格式描述。

```
{ "result": [ [ <label1>, <score1> ], [ <label2>, <score2> ], ... ], ... ] }
```

在返回的 JSON 格式中，`label` 是分类结果，`score` 该分类的概率结果。

(2) 回归任务的返回格式描述。

```
{ "result": [ <value1>, <value2>, <value3>, ... ] }
```

在返回的 JSON 格式中，每个 `value` 都是回归任务的返回值。这些 `value` 的顺序是按照输入样本的顺序进行排列。

(3) 预测任务的返回格式描述。

预测任务的返回结果有两种。

① 如果按照行的方式请求，则返回结果如下：

```
{ "predictions": 值或列表}
```

② 如果按照列的方式请求，则返回结果如下：

```
{"outputs": 值或列表}
```

4. 在Linux通过CURL访问服务

在了解完HTTP/REST API的具体使用细节后，便可以开始构建请求数据了。

可用CURL命令来模拟一个URL请求。CURL是一个利用URL语法在命令行下工作的文件传输工具，在Web开发中应用广泛，常用于接口间的测试与对接。具体操作如下：

(1) 启动服务器。

还是采用13.2.5小节的启动命令。这里不再详述。具体命令如下：

```
tensorflow_model_server --port=9000 --model_base_path=/home1/test/tfservingmodel/ --model_name=md --rest_api_port=8500
```

(2) 输入CURL命令。

在Linux命令行下直接输入以下命令：

```
curl -d '{"instances": [1.0,2.0,5.0],"signature_name":"my_signature"}' -X POST http://localhost:8500/v1/models/md:predict
```

命令中的参数解读如下。

- **-d**: 具体的数据内容。它是JSON格式的数据，具体见“2. POST请求中JSON数据的格式”。
- **-X POST**: 以POST方式发送请求。后面跟的是URL连接。

(3) 执行CURL命令。

该命令执行后，可以看到如下输出：

```
{
  "predictions": [1.96339, 3.97367, 10.0045]
}
```

从结果中可以看出，返回结果也是JSON格式的数据。其中的内容为模型计算后的结果。

5. 在Windows中通过CURL访问服务

CURL工具也支持Windows版本。只不过在Windows中，需要对JSON格式的字符做转义。

以一个列结果输出的例子为演示，具体输入如下：

```
curl -d "{\"inputs\": [2.0,3.0],\"signature_name\":\"my_signature\"}" -X POST http://服务器的ip地址:8500/v1/models/md:predict
```

在参数-d之后的JSON数据中，每个双引号都进行了转义。同时将输入关键字instances换成了inputs，使其以列的形式返回。命令执行后，输出如下结果：

```
{
  "outputs": [
```

```

3.97367,
5.98396
]
}
}
```

13.2.7 扩展：关于 TF_Serving 的更多例子

前文实现一个极为简单的例子，意在讲解 TF_Serving 模块的完整用法。在以下网站中，还有更多使用的例子：

https://github.com/tensorflow/serving/tree/master/tensorflow_serving/example

13.3 实例 68：在安卓手机上识别男女

在 5.2 节介绍过通过微调模型识别男女的例子。本节继续使用该数据集。

现将模型部署在安卓系统上，调用手机的摄像头来识别人物的性别。

实例描述

有一组照片，分为男人和女人。将其作为数据集，用来微调一个 ImgNet 上训练好的成熟模型，使该模型能够识别人物的性别。并将其部署到安卓手机上进行应用。

本节使用的数据集与 5.2 节的一致。微调部分也在第 5 章有详细介绍。这里将把分辨男女的模型部署到安卓手机中。

13.3.1 准备工程代码

TensorFlow 在提供 lite 模块的同时，也提供了一个非常好的教学工程。该工程将在训练脚本及安卓、苹果系统上的 App 项目一起打包实现，以方便用户学习。本实例也使用该工程的代码。该代码的下载链接如下：

<https://github.com/googlegooglesamples/tensorflow-for-poets-2>

将该工程下载并解压缩后，再将 5.2 节的数据集（data 目录）复制到该目录下，完成整体工程的部署。目录结构如图 13-2 所示。

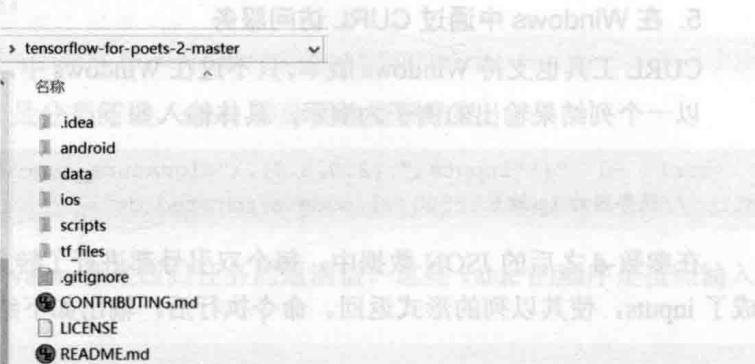


图 13-2 TF-Lite 模块 demo 项目的目录结构

其中的目录描述如下。

- idea: 开发工具自动生成的隐藏文件夹, 可以忽略。
- android: 存放安卓端 App 的工程代码。
- data: 存放男女图片的数据集。
- ios: 存放苹果端 App 的工程代码。
- scripts: 存放再训练模型相关的工具脚本。
- tf_lites: 存放准备使用的 lite 模型文件。

下面将用 scripts 目录下的脚本微调模型, 用 android 目录下的工程加载模型。在 android 目录下有两个文件夹 tflite 与 tfmobile, 分别代表两个工程。前者是在安卓系统中加载 lite 格式的模型; 后者是在安卓系统中加载冻结图格式的模型。

13.3.2 微调预训练模型

预训练模型使用的是 scripts 目录中的源代码文件 “retrain.py”, 该文件的使用方法与 5.5 节类似。在该文件中的代码第 1143 行及以下, 可以看到该文件运行时所需要的具体参数。例如:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--image_dir',
        type=str,
        default='',
        help='Path to folders of labeled images.')
    ...
    parser.add_argument(...)
```

在上面代码中可以看到, 每个参数都有具体的解释 (在 help 参数中) 和默认值 (在 default 参数中)。读者可以自行查看。

1. 介绍 retrain 的参数及选取模型的方法

其中需要重点关注的参数有两个。

(1) --final_tensor_name: 指定模型最终输出的张量名称, 在转化模型时会用到。默认为 final_result。

(2) --architecture: 在微调过程中, 指定所选择的预训练模型。所支持的模型可以在以下链接中找到:

<https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html>

本实例中使用的模型是 MobileNet_1.0_224。

2. 微调模型

在“开始”菜单的“运行”框中运行 cmd 命令, 来到命令行模式。通过 cd 命令进入当前代码所在的路径下, 然后直接用以下命令微调模型:

```
python      scripts/retrain.py      --image_dir=data\train      --random_crop=10
--random_scale=10      --random_brightness=10      --architecture=MobileNet_1.0_224
--learning_rate=0.001      --how_many_training_steps=100
--output_graph=tf_files/retrained_graph.pb
--output_labels=tf_files/retrained_labels.txt
```

上面的命令含义是：选择 MobileNet_1.0_224 预训练模型，用数据增强方法训练 data\train 下的数据集，训练的次数为 100 次，生成的模型文件是 tf_files/retrained_graph.pb，标签文件是 tf_files/retrained_labels.txt。

系统运行时，会默认去网上下载 MobileNet_1.0_224 预训练模型，并放到本地盘符的根目录 tmp 下。例如，作者的本地代码在 G 盘，MobileNet_1.0_224 模型就会下载到 G:\tmp\imagenet 下。如果是 Linux 系统，则模型被直接下载到/tmp 下。



提示：

如果由于网络原因无法下载该模型，则可以使用本书配套资源中的模型文件。直接将 tmp 文件夹解压缩出来，放到盘符的根目录下即可。

3. 获得微调后的模型

运行微调模型的命令后，会输出如下信息：

(1) 标签信息，如图 13-3 所示。

```
INFO:tensorflow:Looking for images in 'man'
INFO:tensorflow:Looking for images in 'woman'
```

图 13-3 微调模型的输出标签信息

(2) 数据处理信息，如图 13-4 所示。

```
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\019489.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\019489.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\004938.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\016566.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\004905.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\010432.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\005549.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\008335.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\012536.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\002600.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\009405.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\man\007552.jpg_MobileNet_1.0_224.txt
INFO:tensorflow:Creating bottleneck at /tmp/bottleneck\woman\007943.jpg_MobileNet_1.0_224.txt
```

图 13-4 微调模型后输出数据处理信息

(3) 训练信息，如图 13-5 所示。

```
INFO:tensorflow:2018-06-07 18:06:35.996925: Step 0: Train accuracy = 53.1%
INFO:tensorflow:2018-06-07 18:06:35.996925: Step 0: Cross entropy = 0.544592
INFO:tensorflow:2018-06-07 18:06:36.919422: Step 0: Validation accuracy = 42.0% (N=64)
INFO:tensorflow:2018-06-07 18:06:42.169503: Step 10: Train accuracy = 84.4%
INFO:tensorflow:2018-06-07 18:06:43.169503: Step 10: Cross entropy = 0.399055
INFO:tensorflow:2018-06-07 18:06:43.243436: Step 10: Validation accuracy = 89.1% (N=64)
INFO:tensorflow:2018-06-07 18:06:49.356117: Step 20: Train accuracy = 89.1%
INFO:tensorflow:2018-06-07 18:06:49.356117: Step 20: Cross entropy = 0.374813
INFO:tensorflow:2018-06-07 18:06:49.431879: Step 20: Validation accuracy = 90.6% (N=64)
```

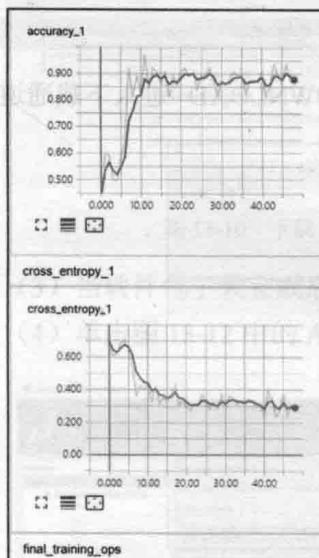
图 13-5 微调模型后输出训练信息

(4) 训练结束后，在tf_files文件夹中生成模型文件和标签文件，如图13-6所示。

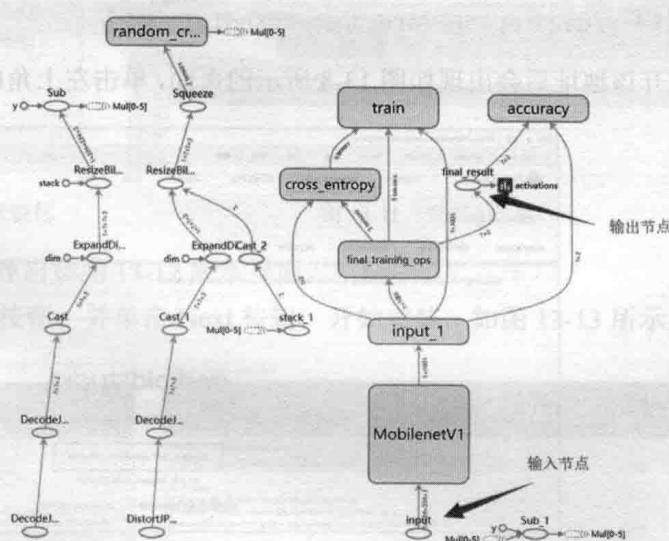
名称		修改日期	类型	大小
桌面	retrained_graph.pb	2018/6/7 17:44	PB文件	16,723 KB
下载	retrained_labels.txt	2018/6/7 17:44	文本文档	1 KB

图13-6 微调后的结果

(5) 在训练过程中，生成的日志信息存放在tmp\retrain_logs\train目录下。（作者本地路径是G:\tmp\retrain_logs\train）。用TensorBoard工具进行查看，过程图和结构图如图13-7所示。



(a) 模型的训练过程图



(b) 模型的结构图

图13-7 微调后的模型日志

图13-7(a)显示了模型训练过程中准确率和损失值的变化情况。图13-7(b)中显示了模型的内部结构。在模型结构图的最下方可以找到输入节点input；在模型结构图的最上方第2行中间可以找到输出节点final_result。



提示：

还可以用scripts文件夹下的脚本对训练好的冻结图文件进行二次瘦身。例如：

(1) 删去输入、输出中不用的节点，并将预处理过程的归一化操作与卷积操作合并。这样减少了模型的运算次数，提升了模型的整体运算速度。

```
python -m tensorflow.python.tools.optimize_for_inference --input=tf_files/retrained_graph.pb --output=tf_files/optimized_graph.pb --input_names="input" --output_names="final_result"
```

(2) 通过压缩权重的方式量化模型，使模型变得更小。

```
python scripts/quantize_graph.py --input=tf_files/optimized_graph.pb --output=tf_files/
rounded_graph.pb --output_node_names=final_result --mode=weights_rounded
```

13.3.3 搭建安卓开发环境

模型准备好之后，就可以将其安装到安卓系统上了。

通过本节的操作，先将安卓的开发环境搭建起来。

1. 下载安卓开发工具

安卓开发工具的下载地址如下：

<https://developer.android.com/studio/>

打开该地址后会出现如图 13-8 所示的页面，单击左上角的“DOWNLOAD”进入下载通道。



图 13-8 下载安卓开发工具



提示：

在打开软件时，可能会由于网络原因无法访问官网进行更新。可以通过设置代理来完成更新。具体方法可以自行在百度或谷歌里进行搜索。

安装好 Android Studio 之后，双击该程序将其打开。刚打开 Android Studio 之后系统会自动更新软件包。等待片刻，待其更新之后将弹出如图 13-9 所示界面。



图 13-9 安卓开发工具启动界面

2. 打开工程代码，并编译程序

(1) 在图 13-9 中选择第二项（打开一个存在的程序），然后选中 tensorflow-for-poets-2-master\android\tflite 目录，这时系统会下载 gradle 包装器，如图 13-10 所示。

(2) 当 gradle 加载完成后，会出现如图 13-11 所示工作区界面，系统会自动编译该项目。如出现问题，则单击 Error 后面的链接，系统会自动下载缺失的软件包（图 13-11 中的箭头处）。



图 13-10 下载 gradle 安装包

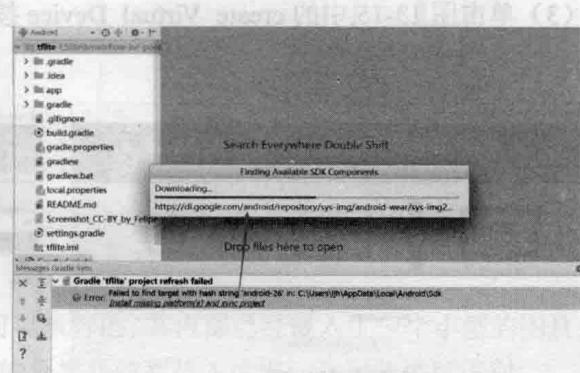


图 13-11 安卓工作区

(3) 当软件包下载完成后，会弹出如图 13-12 所示界面。

(4) 单击图 13-12 中的 Accept 按钮，并单击 Next 按钮，开始安装，如图 13-13 所示。

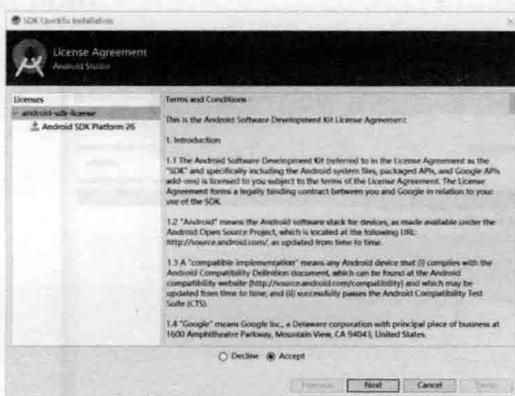


图 13-12 缺失的安卓包下载完毕

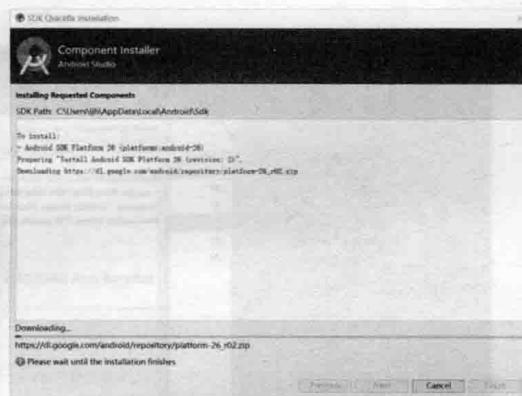


图 13-13 正在安装

(5) 安装好之后，系统会再次自动编译程序。如果再次遇到缺失软件包的错误，则接着按照图 13-11 进行操作。直到编译完成，没有任何错误为止。

3. 创建虚拟设备

在开发安卓软件时，除需要 APP 代码外，还需要有移动设备的测试环境。

- (1) 用安卓开发工具自带的模拟环境来创建一个模拟的移动设备，以进行测试。
- (2) 待编译好后，单击图 13-14 中画圈的按钮来创建一个虚拟设备。

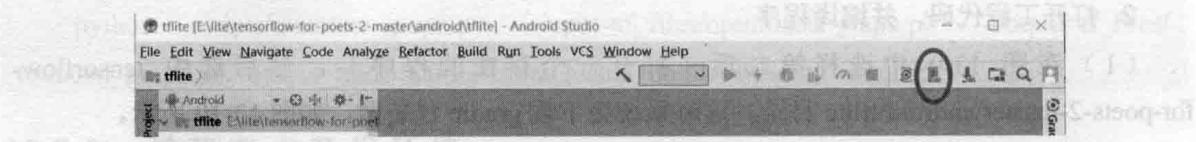


图 13-14 创建虚拟设备

(3) 单击图 13-15 中的 Create Virtual Device 按钮, 进入创建虚拟设备页面, 如图 13-16 所示。



图 13-15 创建虚拟设备

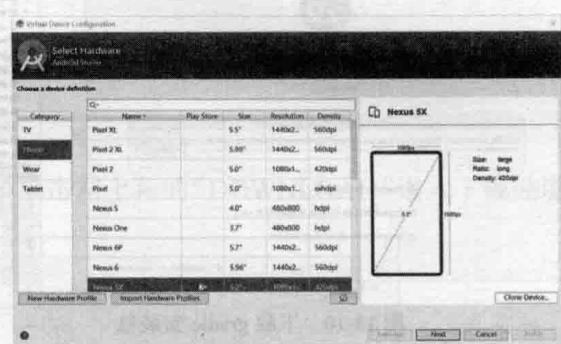


图 13-16 创建虚拟设备界面

(4) 在图 13-16 中选择指定的手机型号, 单击 Next 按钮, 进入如图 13-17 所示的页面。

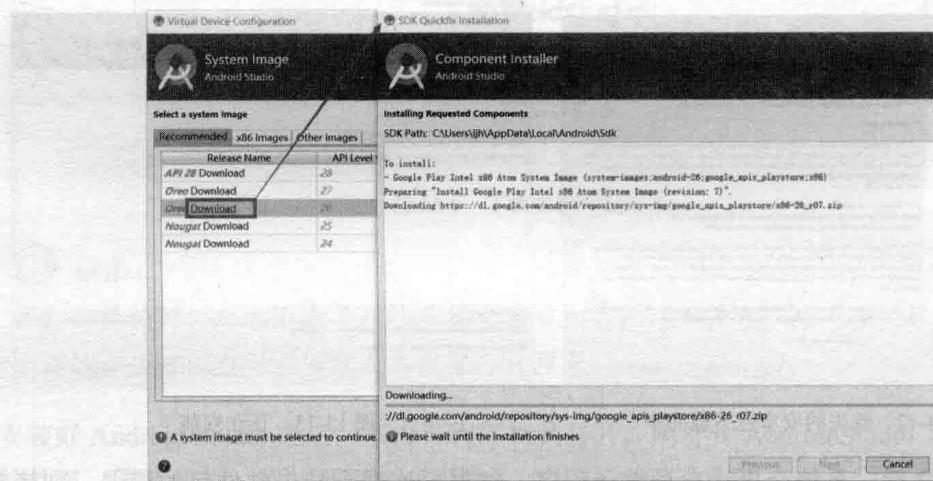


图 13-17 指定 API 版本

(5) 在图 13-17 中选择左侧指定版本的 API 软件包, 然后进入下载页面。当下载完成后, 会出现如图 13-18 所示的界面。

(6) 单击 Show Advanced Settings 按钮, 进入高级设置界面, 如图 13-19 所示。

(7) 将前后摄像头都设置成本机的摄像头 Webcam0 (前提保证本机电脑上已经安装了摄像头设备), 如图 13-19 所示。最终单击图 13-18 中的 Finish 按钮, 完成虚拟设备的创建。

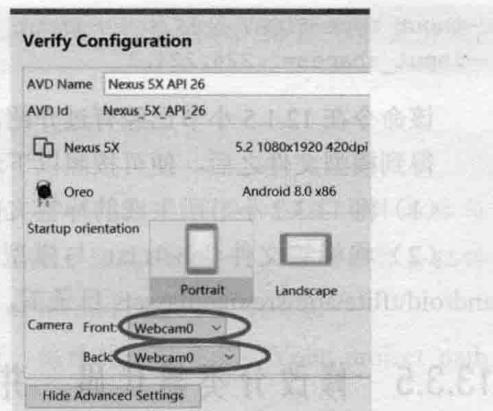
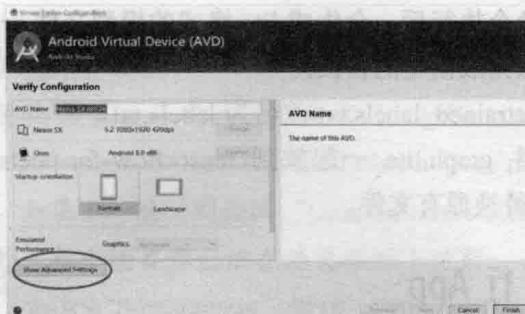


图 13-18 设置虚拟设备

图 13-19 设置虚拟设备的摄像头

4. 测试工程代码

工程代码 tensorflow-for-poets-2 本身是可以执行的，其内部已经嵌入了一个小型的图片分类器模型。可以在该工程中用虚拟设备将内部的分类器模型载入进来，并用其进行识别。

按照图 13-20 中的箭头顺序依次单击，以两步程序和启动虚拟设备。最终单击 OK 按钮，完成虚拟设备的选择。

选择好虚拟设备之后，系统会启动一个手机程序，并调用本机摄像头。手机上的画面如图 13-21 所示。

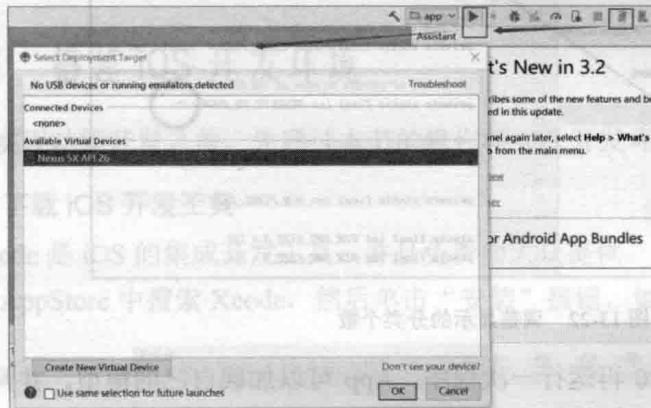


图 13-20 同步与启动设备



图 13-21 运行示例程序

在图 13-21 的下方显示了 4 行信息。第 1 行是输入模型的采样帧率，后面 3 行是模型所识别出的结果中概率最大的 3 个类别。

13.3.4 制作 lite 模型文件

将 13.3.2 小节训练好的 retrained_graph.pb 文件放到装有 TensorFlow 的 Linux 机器上（如果使用 TensorFlow 1.13.1 版本，则可以直接在 Windows 系统下运行）。用以下命令将其转化为 lite 文件：

```
toco --graph_def_file=./retrained_graph.pb --input_format=TENSORFLOW_GRAPHDEF
--output_format=TFLITE --output_file=graph.lite --inference_type=FLOAT
```

```
--input_type=FLOAT          --input_arrays=input          --output_arrays=final_result
--input_shapes=1,224,224,3
```

该命令在 12.1.5 小节已经有过介绍。命令执行后，会生成 lite 格式的模型文件 graph-lite。

得到模型文件之后，便可按照以下步骤完成模型的替换：

- (1) 将 13.3.2 小节所生成的标签文件 retrained_labels.txt 改名为 labels.txt。
- (2) 将标签文件 labels.txt 与模型文件 graph-lite 一起放到 tensorflow-for-poets-2-master\android\tflite\app\src\main\assets 目录下，并替换原有文件。

13.3.5 修改分类器代码，并运行 App

本实例中使用的模型与实际工程中的模型参数比较接近，所以只需要改动显示的分类个数即可。

如图 13-22 所示，打开 ImageClassifier.java 文件，将其中 RESULTS_TO_SHOW 的值设为 2。原因是：在识别男女模型中标签只有两个；而在 App 中，页面设置了显示 3 个分类结果的概率。所以，需要将其改成 2。

另外，如果使用的模型的输入尺寸不是 224，也可以在改代码中进行修改。

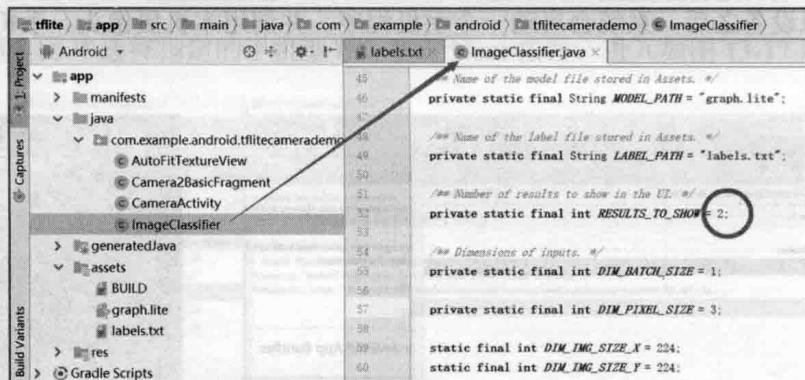


图 13-22 调整显示的分类个数

一切准备好之后，按照图 13-20 再运行一次程序。App 可以加载自己的模型，并判别出男女。显示的结果如图 13-23 所示。



图 13-23 App 运行结果

如图 13-23 所示，在图片旁边显示了 3 行信息。同样，第 1 行是帧率，第 2、3 行是分类的结果。



提示：

在替换模型过程中，一定要让 `ImageClassifier.java` 中的指定模型和标签文件名称与 `tensorflow-for-poets-2-master\android\tflite\app\src\main\assets` 目录下的一致。这是很容易犯错的地方。如果不一致，则会报“Uninitialized classifier or invalid context.” 错误。

另外，整个项目都应该在英文路径下进行。否则，编译时会报类似“Your project path contains non-ASCII characters” 错误。

13.4 实例 69：在 iPhone 手机上识别男女并进行活体检测

在 13.3.4 小节制作好的 lite 模型基础之上，实现一个活体检测程序。

实例描述

在 iOS 上实现一个活体检测程序。

要求：在进行活体检测之前，能够识别出人物性别，并根据性别显示问候语。

本实例可以分为两部分功能：第 1 部分是性别识别，第 2 部分是活体检测。

13.4.1 搭建 iOS 开发环境

在实现功能开发之前，先通过本节的操作将 iOS 开发环境搭建起来。

1. 下载 iOS 开发工具

Xcode 是 iOS 的集成开发工具，并且免费向大众提供。可以通过 AppStore 下载它。

在 AppStore 中搜索 Xcode，然后单击“安装”按钮，如图 13-24 所示。

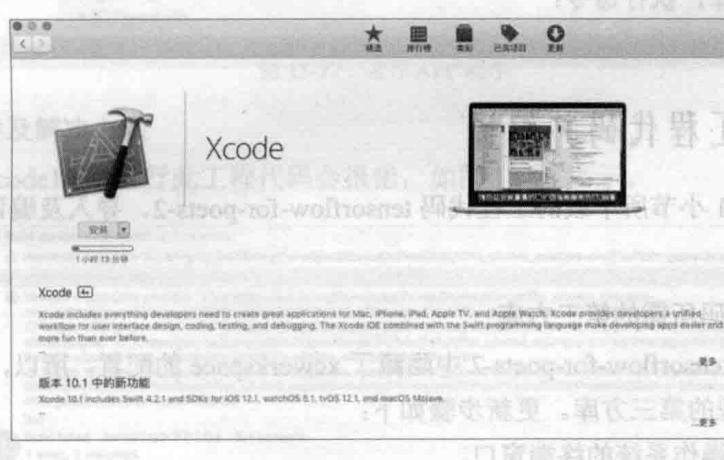


图 13-24 Xcode 的安装界面

2. 安装 CocoaPods

CocoaPods 是一个负责管理 iOS 项目中第三方开源库的工具。

CocoaPods 能让我们、统一地管理第三方开源库，从而节省设置和更新第三方开源库的时间。具体安装方法如下：

(1) 安装 CocoaPods 需要用到 Ruby。虽然 Mac 系统自带 Ruby，但是需要将其更新到最新版本。更新方法是，在命令行模式下输入以下命令：

```
sudo gem update --system
```

(2) 更换 Ruby 的软件源。有时会因为网络原因无法访问到 Ruby 的软件源“rubygems.org”，所以需要将“rubygems.org”地址更换为更容易访问的地址，即把 Ruby 的软件源切换至 [ruby-china](https://gems.ruby-china.com/)。执行命令：

```
gem sources --add https://gems.ruby-china.com/
gem sources --remove https://rubygems.org/
```

(3) 检查源路径是否替换成功。执行命令：

```
gem sources -l
```

该命令执行完后，可以看到 Ruby 的软件源已经更新，如图 13-25 所示。

```
ganyuedeMacBook-Air:~ ganyue$ gem sources -l
*** CURRENT SOURCES ***

https://gems.ruby-china.com/
ganyuedeMacBook-Air:~ ganyue$
```

图 13-25 Ruby 软件源已经更新

(4) 安装 CocoaPods，执行命令：

```
sudo gem install cocoapods
```

(5) 安装本地库，执行命令：

```
pod setup
```

13.4.2 部署工程代码并编译

下面使用 13.3.1 小节所下载的工程代码 tensorflow-for-poets-2。导入及编译该工程中的 iOS 代码的步骤如下：

1. 更新工程代码所需的第三方库

因为工程代码 tensorflow-for-poets-2 中隐藏了.xcworkspace 的配置，所以，在运行前需要用 CocoaPods 更新管理的第三方库。更新步骤如下：

(1) 打开 Mac 操作系统的终端窗口。

(2) 输入“cd”，并且按空格键。

(3) 将工程目录下的文件夹拖入终端窗口，按 Enter 键。

(4) 输入“pod update”指令来更新第三方库。

完整的流程如图 13-26 所示。

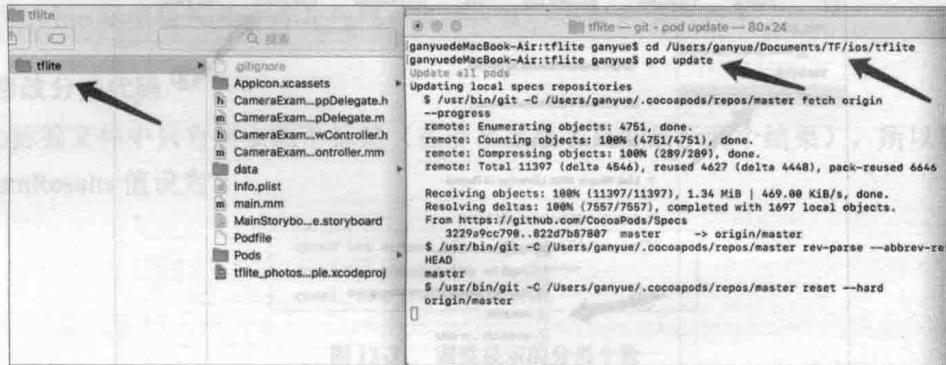


图 13-26 更新代码第三方库

2. 打开工程代码，并编译程序

完成更新之后，在项目目录下会生成一个.xcworkspace 文件。双击该文件打开 Xcode 工具。在 Xcode 工具中选择需要运行的模拟器（见图 13-27 中标注 1 部分），并单击“运行”按钮（见图 13-27 中标注 2 部分）在模拟器中启动应用程序，如图 13-27 所示。

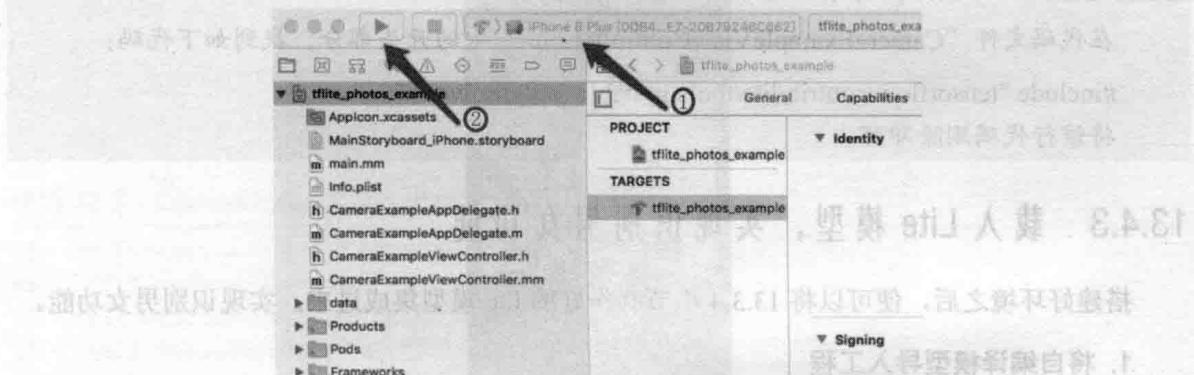


图 13-27 运行 APP 程序

3. 常见错误及解决办法

在最新的 Xcode10 中运行此工程代码会报错，如图 13-28 所示。



图 13-28 错误异常

解决方法是：单击 TARGETS 下的 tflite_photos-example，然后单击 Build Phases，将 Copy Bundle Resources 下的 Info.plist 文件删掉，如图 13-29 所示。

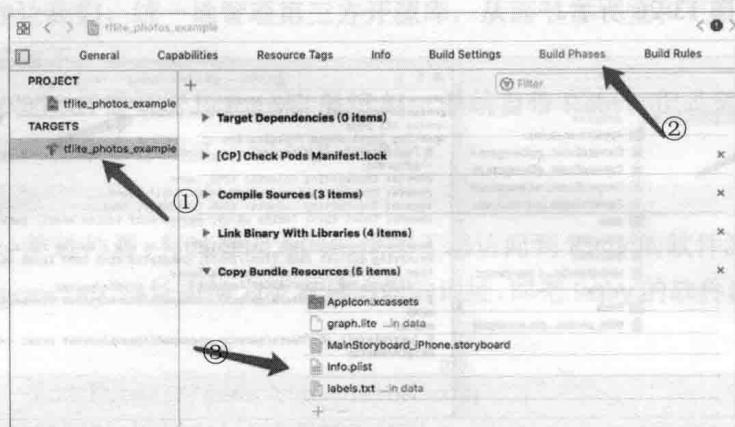


图 13-29 解决错误方法



提示：

在打开工程项目时，需要双击的是.xcworkspace 文件，而不是.xcproject 文件。

另外，如果在运行过程中，如果因为找不到 tensorflow/contrib/lite/tools/mutable_op_resolver.h 文件而报错，则可以使用以下方式来解决：

在代码文件“CameraExampleViewController.mm”中的开头部分，找到如下代码：

```
#include "tensorflow/contrib/lite/tools/mutable_op_resolver.h"
```

将该行代码删除即可。

13.4.3 载入 Lite 模型，实现识别男女功能

搭建好环境之后，便可以将 13.3.4 小节制作好的 lite 模型集成进来，实现识别男女功能。

1. 将自编译模型导入工程

将 13.3.4 小节制作好的 lite 模型和 13.3.2 小节中生成的标签文件，拖到工程代码 tensorflow-for-poets-2-master/ios/tflite/data 目录下，并替换原有文件，如图 13-30 左侧的箭头部分。

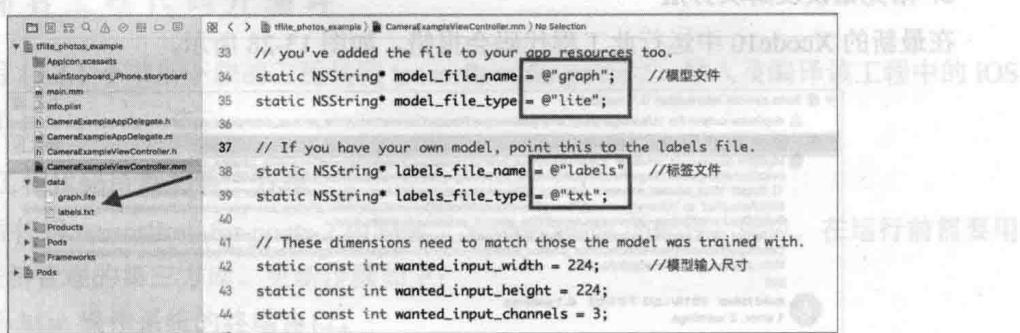


图 13-30 替换文件

**提示：**

在替换过程中，需确保文件名与代码中配置的一致。在运行App过程中，一旦发生不一致的情况，则找不到文件，导致App进程崩溃。

另外，lite模型输入尺寸应与代码中保持一致，否则影响识别率。

2. 修改分类代码

因为标签文件中只有男女两个标签（在屏幕上最多只能显示两个结果），所以将图13-31中的kNumResults值设为2。

```
const int output_size = (int)labels.size();
const int kNumResults = 2; ←
const float kThreshold = 0.1f;
```

图13-31 调整显示的分类个数

3. 运行程序，查看效果

这一环节是在模拟器上实现的。事先将图片保存至模拟器相册，然后从模拟器相册中获取图片来进行人物性别识别。

模拟器运行之后，显示的结果如图13-32所示。



图13-32 在iPhone 8上App的运行结果

13.4.4 代码实现：调用摄像头并采集视频流

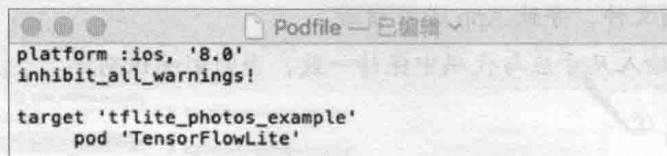
因为活体检测功能需要用到摄像头，所以需要在原来工程代码中添加摄像头功能。具体操作如下：

1. 增加GoogleMobileVision库

活体检测主要是通过计算人脸特征点的位置变化来判断被检测人是否完成了指定的行为动作。该功能是借助谷歌训练好的人脸特征API来实现的。该API为GoogleMobileVision。将其

引入到工程中的操作如下：

- (1) 双击打开工程代码 tensorflow-for-poets-2-master/ios/tflite 下的 Podfile，如图 13-33 所示。

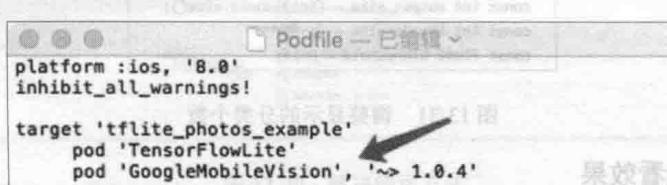


```
platform :ios, '8.0'
inhibit_all_warnings!

target 'tflite_photos_example'
pod 'TensorFlowLite'
```

图 13-33 Podfile 文件

- (2) 增加“pod 'GoogleMobileVision'”，如图 13-34 所示。



```
platform :ios, '8.0'
inhibit_all_warnings!

target 'tflite_photos_example'
pod 'TensorFlowLite'
pod 'GoogleMobileVision', '~> 1.0.4'
```

图 13-34 增加 GoogleMobileVision 后的 Podfile 文件

- (3) 按照 13.4.2 小节的“1. 更新工程代码所需第三方库”中的内容更新第三方库。

2. 自定义相机

- (1) 进入工程中，在左侧工程目录下右击文件，在弹出的菜单中选择“New File”命令，如图 13-35 所示。

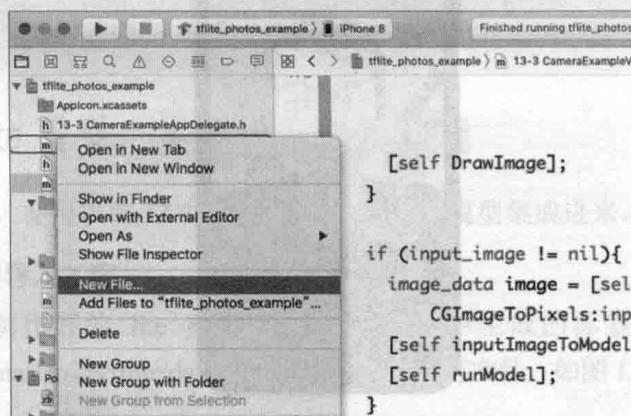


图 13-35 新建文件

- (2) 弹出如图 13-36 所示界面，在其中选择需要创建的平台。这里选择 iOS，然后选择 Source 下的 Cocoa Touch Class。

- (3) 进入 Choose options for your new file 界面，在“Class:”文本框中输入要创建文件的名字，在“Subclass of.”文本框中输入继承的父类名称，在“Language:”文本框中选择 Objective-C，如图 13-37 所示。



图 13-36 选择要创建的平台

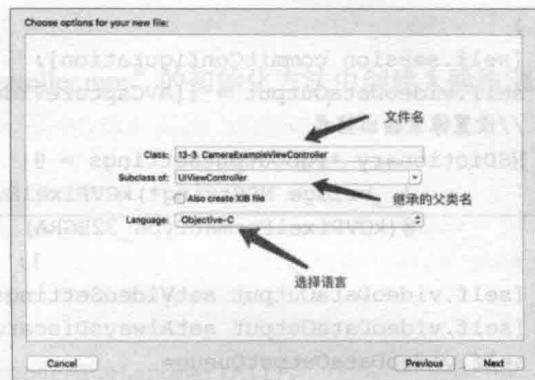


图 13-37 选择文件类型

(4) 在创建的代码文件“13-3 CameraExampleViewController.mm”中声明自定义相机变量。具体代码如下：

代码 13-3 CameraExampleViewController

```
01 //AVCaptureSession 对象来执行输入设备和输出设备之间的数据传递
02 @property(nonatomic,strong)AVCaptureSession *session;
03 //视频输出流
04 @property(nonatomic,strong)AVCaptureVideoDataOutput *videoDataOutput;
05 //预览图层
06 @property(nonatomic,strong)AVCaptureVideoPreviewLayer *previewLayer;
07 //显示方向
08 @property(nonatomic,assign)UIDeviceOrientation lastDeviceOrientation;
```

(5) 添加自定义相机的初始化变量。具体代码如下：

代码 13-3 CameraExampleViewController (续)

```
09 self.session = [[AVCaptureSession alloc] init];
10 //设置 session 显示分辨率
11 self.session.sessionPreset = AVCaptureSessionPresetMedium;
12 [self.session beginConfiguration];
13 NSArray *oldInputs = [self.session inputs];
14 //移除 AVCaptureSession 对象中原有的输入设备
15 for (AVCaptureInput *oldInput in oldInputs) {
16     [self.session removeInput:oldInput];
17 }
18 //设置摄像头方向
19 AVCaptureDevicePosition desiredPosition =
20 AVCaptureDevicePositionFront;
21 AVCaptureDeviceInput *input = [self cameraForPosition:desiredPosition];
22 //添加输入设备
23 if (!input) {
24     for (AVCaptureInput *oldInput in oldInputs) {
25         [self.session addInput:oldInput];
26     }
27 } else {
28     [self.session addInput:input];
```

```

29 }
30 [self.session commitConfiguration];
31 self.videoDataOutput = [[AVCaptureVideoDataOutput alloc] init];
32 //设置像素输出格式
33 NSDictionary *rgbOutputSettings = @{
34     (_bridge NSString*)kCVPixelBufferPixelFormatTypeKey:
35     @(kCVPixelFormatType_32BGRA)
36     };
37 [self.videoDataOutput setVideoSettings:rgbOutputSettings];
38 [self.videoDataOutput setAlwaysDiscardsLateVideoFrames:YES];
39 self.videoDataOutputQueue=
40 dispatch_queue_create("VideoDataOutputQueue",DISPATCH_QUEUE_SERIAL);
41 [self.videoDataOutput setSampleBufferDelegate:self
queue:self.videoDataOutputQueue];
42 //添加输出设备
43 [self.session addOutput:self.videoDataOutput];
44 //相机拍摄预览图层
45 self.previewLayer =
46 [[AVCaptureVideoPreviewLayer alloc] initWithSession:self.session];
47 [self.previewLayer setBackgroundColor:[UIColor clearColor] CGColor]];
48 [self.previewLayer setVideoGravity:AVLayerVideoGravityResizeAspectFill];
49 self.overlayView = [[UIView alloc] initWithFrame:self.view.bounds];
50 self.overlayView.backgroundColor = [UIColor darkGrayColor];
51 [self.view addSubview:self.overlayView];
52 CALayer *overlayViewLayer = [self.overlayView layer];
53 [overlayViewLayer setMasksToBounds:YES];
54 [self.previewLayer setFrame:[overlayViewLayer bounds]];
55 [overlayViewLayer addSublayer:self.previewLayer];

```

(6) 添加自定义相机的代理方法。具体代码如下：

代码 13-3 CameraExampleViewController（续）

```

55 #pragma mark - AVCaptureVideoDataOutputSampleBufferDelegate
56 -(void)captureOutput:(AVCaptureOutput*)captureOutput
57 didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
58 fromConnection:(AVCaptureConnection *)connection{
59 //将 CMSampleBuffer 转换为 UIImage
60 UIImage *image = [GMVUtility sampleBufferTo32RGBA:sampleBuffer];
61 }

```

在上面代码中，用 `AVCaptureVideoDataOutputSampleBufferDelegate` 代理方法获取实时的 `image`。在获取到 `image` 之后，将其分为两个分支：一个用于识别男女性别，另一个用于活体检测。

13.4.5 代码实现：提取人脸特征

本小节用 `GoogleMobileVision` 接口获取人脸关键点，进行人脸特征提取。

1. 创建人脸检测器

在代码文件“13-3 CameraExampleViewController.mm”的初始化方法中创建人脸检测器。具体代码如下：

代码 13-3 CameraExampleViewController（续）

```

60 //配置检测器
61 NSDictionary *options = @{
62     GMVDetectorFaceMinSize : @(0.1),
63     GMVDetectorFaceTrackingEnabled : @(YES),
64     GMVDetectorFaceLandmarkType : @(GMVDetectorFaceLandmarkAll),
65     GMVDetectorFaceClassificationType : @(GMVDetectorFaceClassificationAll),
66     GMVDetectorFaceMode : @(GMVDetectorFaceFastMode)
67 };
68 //创建并返回已配置的检测器
69 self.faceDetector = [GMVDetector detectorOfType:GMVDetectorTypeFace
options:options];

```

2. 获取人脸

在代码文件“13-3 CameraExampleViewController.mm”的相机代理方法中，调用GoogleMobileVision框架的GMVDetector检测功能，获取屏幕上所有的人脸。具体代码如下：

代码 13-3 CameraExampleViewController（续）

```

70 UIImage *image = [GMVUtility sampleBufferTo32RGBA:sampleBuffer];
71 //建立图像方向
72 UIDeviceOrientation deviceOrientation = [[UIDevicecurrentDevice] orientation];
73 GMVImageOrientation orientation = [GMVUtility
imageOrientationFromOrientation:deviceOrientation
withCaptureDevicePosition:AVCaptureDevicePositionFront
defaultDeviceOrientation:self.lastKnownDeviceOrientation];
74 //定义图像显示方向，用于指定面部特征检测
75 NSDictionary *options = @{@"GMVDetectorImageOrientation" : @(orientation)};
76 //使用GMVDetector检测功能
77 NSArray<GMVFaceFeature*>*faces = [self.faceDetector featuresInImage:image
options:options];
78 CMFormatDescriptionRef fdesc = CMSampleBufferGetFormatDescription(sampleBuffer);
79 CGRect clap = CMVideoFormatDescriptionGetCleanAperture(fdesc, false);
80 //计算比例因子和偏移量以正确显示特征
81 CGSize parentFrameSize = self.previewLayer.frame.size;
82 CGFloat cameraRatio = clap.size.height / clap.size.width;
83 CGFloat viewRatio = parentFrameSize.width / parentFrameSize.height;
84 CGFloat xScale = 1;
85 CGFloat yScale = 1;
86 CGRect videoBox = CGRectMakeZero;
87 //判断视频预览尺寸与相机捕获视频帧尺寸
88 if (viewRatio > cameraRatio) {
89     videoBox.size.width = parentFrameSize.height * clap.size.width /
clap.size.height;
90     videoBox.size.height = parentFrameSize.height;

```

```

91     videoBox.origin.x = (parentFrameSize.width-videoBox.size.width) / 2;
92     videoBox.origin.y = (videoBox.size.height-parentFrameSize.height) / 2;
93     xScale = videoBox.size.width / clap.size.width;
94     yScale = videoBox.size.height / clap.size.height;
95 } else {
96     videoBox.size.width = parentFrameSize.width;
97     videoBox.size.height = clap.size.width * (parentFrameSize.width /
clap.size.height);
98     videoBox.origin.x = (videoBox.size.width-parentFrameSize.width) / 2;
99     videoBox.origin.y = (parentFrameSize.height-videoBox.size.height) / 2;
100    xScale = videoBox.size.width / clap.size.height;
101    yScale = videoBox.size.height / clap.size.width;
102 }
103 dispatch_sync(dispatch_get_main_queue(), ^{
104     //移除之前添加的功能视图
105     for (UIView *featureView in self.overlayView.subviews) {
106         [featureView removeFromSuperview];
107     }
108     for (GMVFaceFeature *face in faces) {
109         //所有的 face
110         .....
111     }

```

13.4.6 活体检测算法介绍

通过获取人脸的 GMVFaceFeature 对象可以得到五官参数，从而实现微笑检测、向左转、向右转、抬头、低头、张嘴等功能。

代码第 77 行，会返回一个 GMVFaceFeature 对象。该对象包含人脸的具体信息。其中所包括的字段及含义如下。

- smilingProbability：用于检测微笑，该字段是 CGFloat 类型，取值范围为 0~1。微笑尺度越大，则 smilingProbability 字段越大。
- noseBasePosition：检测图像在视图坐标系中的鼻子坐标。
- leftCheekPosition：检测图像在视图坐标系中的左脸颊坐标。
- rightCheekPosition：检测图像在视图坐标系中的右脸颊坐标。
- mouthPosition：检测图像在视图坐标系中的嘴角坐标。
- bottomMouthPosition：检测图像在视图坐标系中的下唇中心坐标。
- leftEyePosition：检测图像在视图坐标系中的左眼坐标。

在活体检测的行为算法中，只有微笑行为可以直接用 smilingProbability 进行判断。其他的行为需要多个字段联合判断，具体代码如下。

- 左转、右转：通过 noseBasePosition、leftCheekPosition、rightCheekPosition 三点之间的间距进行判断。
- 抬头：通过 noseBasePosition、leftEyePosition 两点之间的间距进行判断。
- 低头：通过 noseBasePosition、rightCheekPosition 两点之间的间距进行判断。

- 162 • 张嘴：通过 mouthPosition、bottomMouthPosition 两点之间的间距进行判断。
163

13.4.7 代码实现：实现活体检测算法

164 在了解原理之后，就可以编写代码实现人脸检测算法。具体如下：
165

1. 识别左转、右转行为

166 左转、右转的识别行为算法是通过鼻子与左、右脸颊 x 坐标的间距之差来判断的。如果左边间距比右边间距大 20 以上，即为左转；反之则为右转。具体代码如下：
167

代码 13-3 CameraExampleViewController（续）

```
112 //鼻子的坐标  
113 CGPoint nosePoint = [weakSelf scaledPoint:face.noseBasePosition xScale:xScale  
114 yScale:yScale offset:videoBox.origin];  
115 //左脸颊的坐标  
116 CGPoint leftCheekPoint = [weakSelf scaledPoint:face.leftCheekPosition  
117 xScale:xScale yScale:yScale offset:videoBox.origin];  
118 //右脸颊的坐标  
119 CGPoint rightCheekPoint = [weakSelf scaledPoint:face.rightCheekPosition  
120 xScale:xScale yScale:yScale offset:videoBox.origin];  
121 //鼻子与右脸颊之间的距离  
122 CGFloat leftRightFloat1 = rightCheekPoint.x - nosePoint.x;  
123 //鼻子与左脸颊之间的距离  
124 CGFloat leftRightFloat2 = nosePoint.x - leftCheekPoint.x;  
125 if (leftRightFloat2 - leftRightFloat1 > 20) {  
126 //左转  
127 } else if (leftRightFloat1 - leftRightFloat2 > 20) {  
128 //右转  
129 } else{  
130 //没有转动，或者转动幅度小  
131 }
```

2. 识别抬头、低头行为

132 通过计算鼻子和左眼的 y 坐标之差是否小于 24，来判断是否为抬头的行为。如果鼻子与右脸颊的 y 坐标之差大于 0，则为低头行为。具体代码如下：
133

代码 13-3 CameraExampleViewController（续）

```
129 //鼻子的坐标  
130 CGPoint nosePoint = [weakSelf scaledPoint:face.noseBasePosition xScale:xScale  
131 yScale:yScale offset:videoBox.origin];  
132 //左眼的坐标  
133 CGPoint leftEyePoint = [weakSelf scaledPoint:face.leftEyePosition xScale:xScale  
134 yScale:yScale offset:videoBox.origin];  
135 //右脸颊的坐标  
136 CGFloat if(nosePoint.y - leftEyePoint.y < 24){  
137 //抬头  
138 }
```

```

137 }else if(nosePoint.y - rightCheekPoint.y > 0) {
138 //低头
139 }

```

3. 识别张嘴行为

通过计算上唇中心 y 坐标与下唇中心 y 坐标之差是否大于 18，来判定是否为张嘴的行为。具体代码如下：

代码 13-3 CameraExampleViewController（续）

```

140 //下唇中心的坐标
141 CGPoint bottomMouthPoint = [weakSelf scaledPoint:face.bottomMouthPosition
142 xScale:xScale yScale:yScale offset:videoBox.origin];
142 //上唇中心的坐标
143 CGPoint mouthPoint = [weakSelf scaledPoint:face.mouthPosition xScale:xScale
144 yScale:yScale offset:videoBox.origin];
144 if(bottomMouthPoint.y - mouthPoint.y > 18){
145 //张嘴
146 .....
147 }

```

4. 识别微笑行为

微笑行为可直接通过 `face.smilingProbability` 属性判断出来。具体代码如下：

代码 13-3 CameraExampleViewController（续）

```

148 //微笑判断，0.3 是经过验证后的经验值
149 if (face.smilingProbability > 0.3) {
150 //微笑
151 .....
152 }

```

13.4.8 代码实现：完成整体功能并运行程序

将男女识别算法与所有的活体检测算法结合起来，完成完整流程。并在其中添加问候语。具体代码如下：

1. 实现完整流程

代码 13-3 CameraExampleViewController（续）

```

153 for (GMVFaceFeature *face in faces) {
154     CGRect faceRect = [weakSelf scaledRect:face.bounds xScale:xScale yScale:yScale
155     offset:videoBox.origin];
155     //判断是否在指定的尺寸里
156     if (CGRectContainsRect(weakSelf.bgView.frame, faceRect)) {
157         //如果 index 为 1，则表示微笑行为
158         if(index == 1{
159             if(face.smilingProbability > 0.3){
160                 }
161             //如果 index 为 2，则表示左转、右转行为

```

```

162     }else if(index == 2){
163         //鼻子的坐标
164         CGPoint nosePoint = [weakSelf scaledPoint:face.noseBasePosition
165             xScale:xScale yScale:yScale offset:videoBox.origin];
166         //左脸颊的坐标
167         CGPoint leftCheekPoint = [weakSelf scaledPoint:face.leftCheekPosition
168             xScale:xScale yScale:yScale offset:videoBox.origin];
169         //右脸颊的坐标
170         CGPoint rightCheekPoint = [weakSelf scaledPoint:face.rightCheekPosition
171             xScale:xScale yScale:yScale offset:videoBox.origin];
172         //鼻子与右脸颊之间的距离
173         CGFloat leftRightFloat1 = rightCheekPoint.x - nosePoint.x;
174         //鼻子与左脸颊之间的距离
175         CGFloat leftRightFloat2 = nosePoint.x - leftCheekPoint.x;
176         if (leftRightFloat2 - leftRightFloat1 > 20) {
177             //左转
178         }else if (leftRightFloat1 - leftRightFloat2 > 20) {
179             //右转
180         }
181         //如果 index 为 3，则表示张嘴行为
182     }else if(index == 3){
183         //下唇中心的坐标
184         CGPoint bottomMouthPoint = [weakSelf scaledPoint:face.
185             bottomMouthPosition xScale:xScale yScale:yScale offset:videoBox.origin];
186         //上唇中心的坐标
187         CGPoint mouthPoint = [weakSelf scaledPoint:face.mouthPosition
188             xScale:xScale yScale:yScale offset:videoBox.origin];
189         if(bottomMouthPoint.y - mouthPoint.y > 18){
190             //张嘴
191         }
192         //如果 index 为 4，则表示抬头、低头行为
193     }else if(index == 4){
194         //鼻子的坐标
195         CGPoint nosePoint = [weakSelf scaledPoint:face.noseBasePosition
196             xScale:xScale yScale:yScale offset:videoBox.origin];
197         //左眼的坐标
198         CGPoint leftEyePoint = [weakSelf scaledPoint:face.leftEyePosition
199             xScale:xScale yScale:yScale offset:videoBox.origin];
200         //右脸颊的坐标
201         CGPoint rightCheekPoint = [weakSelf scaledPoint:face.rightCheekPosition
202             xScale:xScale yScale:yScale offset:videoBox.origin];
203         if(nosePoint.y - leftEyePoint.y < 24){
204             //抬头
205         }else if(nosePoint.y - rightCheekPoint.y > 0){
206             //低头
207         }
208     }
209 }
210 }
```

2. 添加问候语

在代码文件 CameraExampleViewController.mm 中添加下列代码，实现问候语的显示功能。具体代码如下：

代码 13-3 CameraExampleViewController (续)

```

203 //遍历获取到的所有结果
204 for (const auto& item : newValues) {
205     std::string label = item.second;
206     const float value = item.first;
207     if (value > 0.5) {
208         NSString *nsLabel = [NSString stringWithFormat:@"%@", label.c_str()];
209         encoding:[NSString defaultCStringEncoding]];
210         NSString *textString;
211         if ([nsLabel isEqualToString:@"man"]) {
212             textString = @"先生你好";
213         }else{
214             textString = @"女士你好";
215         }
216         //创建 UILabel 显示对应的问候语
217         .....
218     }

```

3. 运行程序并显示效果

将苹果手机通过 USB 接口连接到电脑上。先选择真机，然后单击“运行”按钮进行程序同步，如图 13-38 所示。



图 13-38 选择真机调试

在手机上打开 App 即可运行程序。当手机屏幕显示绿色边框时，表示正在检测。手机屏幕离人脸 50cm 为最佳距离。以检测微笑、张嘴的行为为例，程序运行结果如图 13-39、图 13-40 所示。



图 13-39 微笑检测



图 13-40 张嘴检测

图 13-39 表示程序识别出微笑行为，图 13-40 表示程序识别出张嘴行为。



提示：

在 iOS 9 之后的操作系统中，使用相机功能需要在项目 Info.plist 文件中增加了“Privacy - Camera Usage Description”权限提示，否则会报出异常。

13.5 实例 70：在树莓派上搭建一个目标检测器

深度学习的出现，让人们看到人工智能在现实世界中创造出了巨大的机会。但深度学习往往需要巨大的计算能力，有时我们身边没有强大的服务器或 NVIDIA 的 GPU 加速平台，而只有一个 ARM CPU，那我们如何将深度学习部署到 ARM CPU 上呢？本节将利用树莓派和 TensorFlow 开发一个 CNN 目标检测器。

实例描述

在树莓派上安装一个目标检测器模型，让其能够通过摄像头完成目标检测功能。

本实例使用的树莓派型号是 Raspberry Pi 3 Model B。该型号是目前市面上常见的一款树莓派主板，在各大主流电商平台都可以买到。其外观如图 13-41 所示。

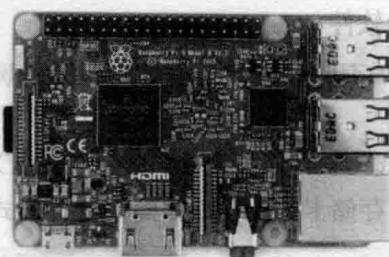


图 13-41 树莓派 Raspberry Pi 3 Model B 的外观

13.5.1 安装树莓派系统

当准备好树莓派主板之后，需要为其安装操作系统。具体步骤如下。

1. 准备硬件

在安装之前，除需要树莓派外，还需要准备一张 TF 卡，用于存放系统程序。卡的速度直接影响树莓派的运行速度。这里推荐使用型号为 class10、容量在 8GB 以上的 TF 卡。同时还得配备一个读卡器，如图 13-42 所示。



图 13-42 读卡器（左）和 TF 卡（右）

2. 下载树莓派系统的镜像文件

树莓派使用的是 raspbian 系统。该镜像文件的下载地址如下：

<http://www.raspberrypi.org/downloads/>

访问该网站后会找到镜像文件的下载链接，如图 13-43 所示。

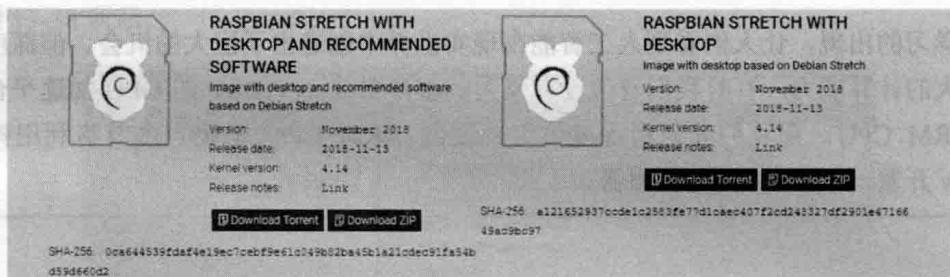


图 13-43 树莓派 3b 镜像下载

这里选择的是右边的 RSPBIAN STRETCH WITH DESKTOP 版本。如果出现下载缓慢或者超时，可以使用第三方下载工具来进行下载。

3. 下载安装镜像文件的工具软件

可以在 Windows 7 或 Windows 10 下用 SDFormatter 和 Raspberry Pi 3 Model B 软件将系统安装到树莓派上。

- **SDFormatter:** 是一款格式化软件，符合 SD 协会（SDA）创建的 SD 文件系统规范。可以将 SD 存储卡、SDHC 存储卡、SDXC 存储卡进行格式化。下载链接如下：

https://www.sdcard.org/downloads/formatter_4/

- **Win32 Disk Imager:** 是一款将原生镜像写入移动设备的软件。它可以将 SD/CF 卡或其

他 USB sticks 上的镜像系统写入移动设备，或者从移动设备中备份镜像。下载链接：

<https://sourceforge.net/projects/win32diskimager/files/Archive/win32diskimager-v0.9-binary.zip/download>

4. 安装步骤

当硬件和软件都准备好后，便可以按以下步骤进行安装。

(1) 把下载的 raspbian 系统解压缩成 IMG 格式(注意：如果镜像保存的路径中有中文字符，则在镜像烧录时可能发生错误)。

(2) 将 TF 卡插入读卡器，连上电脑，用 SDFormatter 软件对 SD 卡进行格式化，Drive 是 SD 卡盘符，如图 13-44 所示。

(3) 格式化完成后，用 Win32 Disk Imager 进行镜像烧录。如图 13-45 所示，打开镜像所在路径并选择 SD 卡盘符，单击 Write 按钮，如果出现对话框，则选择“yes”进行安装。



图 13-44 格式化 SD 卡

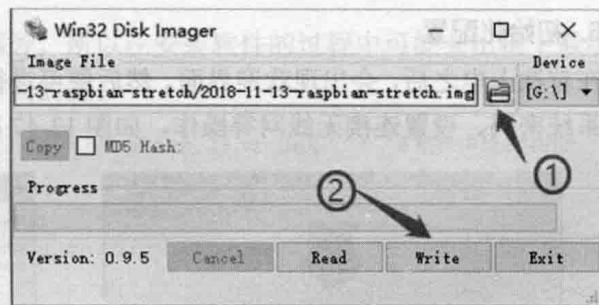


图 13-45 烧录镜像

(4) 安装结束后如弹出完成对话框，则说明安装完成了。如果不成功，则尝试关闭防火墙之类的软件，重新插入 TF 卡进行安装。



提示：

在 Windows 系统中会看到 TF 卡只有 58MB 或更小。这是正常现象，因为 Linux 中的分区在 Windows 中是看不到的。

5. 连接摄像头

将烧录好镜像的 TF 卡插入树莓派背面的 TF 卡槽中，准备一个 USB micro 接头适配器 (5v 2A) 给树莓派供电，一根 HDMI 线（或 HDMI 转 VGA 线）连接树莓派和显示器，如图 13-46 所示。

在选取 USB 摄像头时，推荐使用的硬件配置为：支持 USB 2.0 接口、分辨率在 640×480 pixel 以上的摄像头。本实例所采用的摄像头，具体参数如下。

- 型号：罗技 C170。
- USB 接口类型：2.0。
- 捕获图像分辨率：1027 pixel × 768 pixel。

树莓派的 RSBPIAN 系统原生支持 UVC 设备驱动，可直接使用 USB 摄像头。

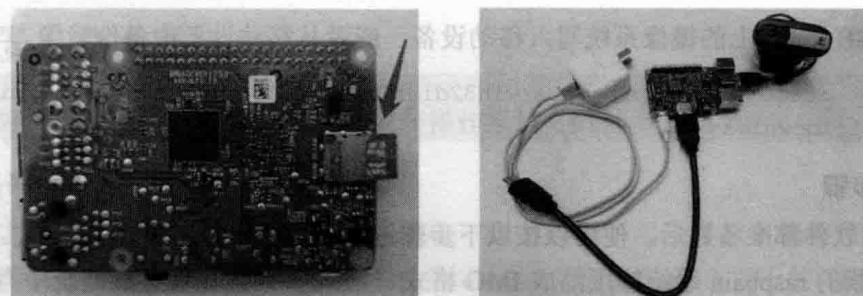


图 13-46 连接树莓派 3B 与摄像头

**提示：**

UVC 设备的全称为 USB video class 或 USB video device class。它是 Microsoft 与另外几家设备厂商联合推出的为 USB 视频捕获设备定义的协议标准。

6. 初始化配置

在首次上电之后，会出现欢迎界面。然后便可进行初始配置操作：设置国家、语言、时钟、设置系统密码、设置连接无线网等操作，如图 13-47 所示。

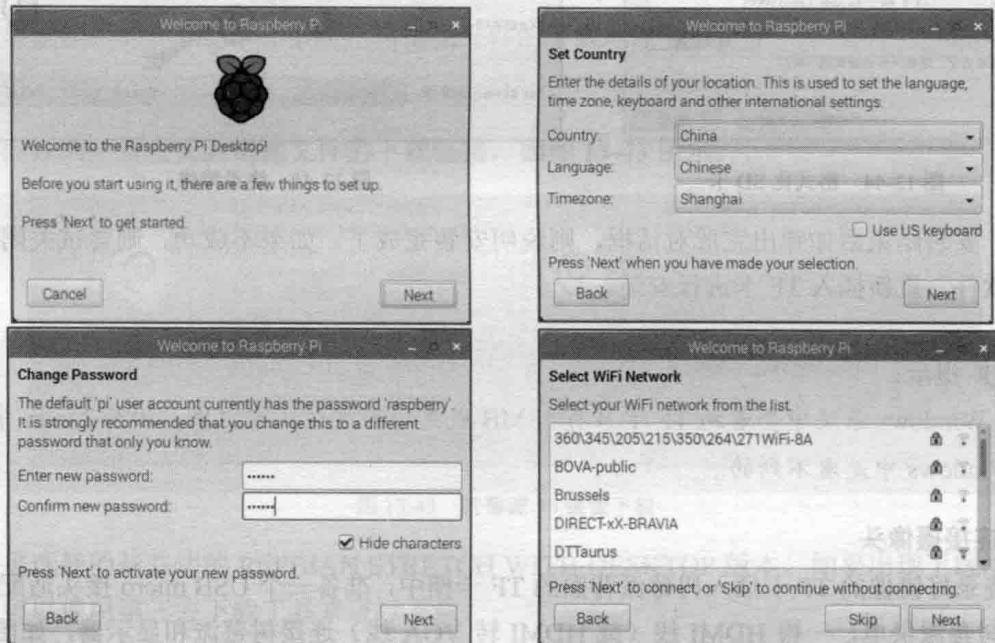


图 13-47 上电初始化配置

在基本的配置完成后，还需要开启树莓派的 CAMERA、SSH、VNC 功能。具体操作如图 13-48 所示。

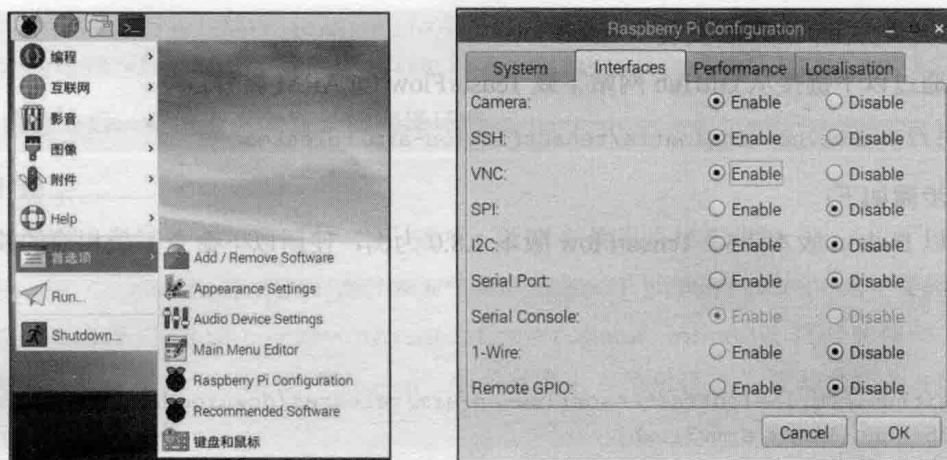


图 13-48 开启 CAMERA、SSH、VNC 功能

7. 更换软件源

由于树莓派使用的软件下载源来自国外，所以在安装软件的过程中可能会出现下载缓慢或者超时的问题。下面将软件源更换为中科大的源。

```

sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak      #先将源进行备份
sudo vi /etc/apt/sources.list                            #更改为以下内容
#使用清华镜像
deb http://mirrors.tuna.tsinghua.edu.cn/raspbian/raspbian/ stretch main contrib
non-free rpi
deb-src http://mirrors.tuna.tsinghua.edu.cn/raspbian/raspbian/ stretch main
contrib non-free rpi
#使用 neusoft 镜像
deb http://mirrors.neusoft.edu.cn/raspbian/raspbian/ stretch main contrib non-free
rpi
deb-src http://mirrors.neusoft.edu.cn/raspbian/raspbian/ stretch main contrib
non-free rpi
#使用 ustc 镜像
deb http://mirrors.ustc.edu.cn/raspbian/raspbian/ stretch main contrib non-free rpi
deb-src http://mirrors.ustc.edu.cn/raspbian/raspbian/ stretch main contrib non-free
rpi

```

修改完之后，将其保存。执行下面的命令进行更新：

```
sudo apt-get update
```

13.5.2 在树莓派上安装TensorFlow

树莓派上安装TensorFlow的方式有多种，可以安装现成的软件包，也可以从源码编译。最方便的方法是直接用 pip 命令进行安装。不过，由于嵌入式设备的定制化配置更加灵活，所以很多情况下无法找到与其匹配的软件包。源码编译的安装方式更为通用。本小节将介绍 3 种安装方式：用 pip 安装、用源码编译安装、通过 Docker 交叉编译源码进行安装。

1. 在树莓派上，用 pip 安装 TensorFlow

可以通过以下链接从 GitHub 网站下载 TensorFlow for ARM 软件：

```
https://github.com/lhelontra/tensorflow-on-arm/releases
```

具体步骤如下：

(1) 以 Python 版本 3.5、TensorFlow 版本 1.8.0 为例，使用以下命令下载相应的软件包：

```
mkdir tf
cd tf
wget
https://github.com/lhelontra/tensorflow-on-arm/releases/download/v1.8.0/tensorflow-1.8.0-cp35-none-linux\_armv7l.whl
```

(2) 安装 TensorFlow 1.8.0 版本，具体命令如下：

```
sudo pip3 install /home/pi/tf/tensorflow-1.8.0-cp35-none-linux_armv7l.whl
sudo pip3 install /home/pi/tf/tensorflow-1.8.0-cp35-none-linux_armv7l.whl -i
https://pypi.tuna.tsinghua.edu.cn/simple
```

在安装过程中，如果出现校验和不同导致的失败，则可以多试几次。如果出现下载超时，则可以选用清华大学的 pypi 镜像站。

(3) 更新软件源，并安装 TensorFlow 的其他依赖项，具体命令如下：

```
sudo apt-get update
sudo apt-get install libatlas-base-dev
sudo pip3 install pillow lxml jupyter matplotlib cython
sudo apt-get install python-tk
```

2. 在树莓派上编译源码，并安装 TensorFlow

(1) 用以下命令下载 TensorFlow 源码：

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

该命令使用的是 GIT 工具，该工具的下载方法见 8.1.7 小节。

(2) 用以下命令编译 TensorFlow 并安装：

```
cd ~/tensorflow
tensorflow/contrib/makefile/download_dependencies.sh      # 安装依赖项
sudo apt-get install -y autoconf automake libtool gcc-4.8 g++-4.8
cd tensorflow/contrib/makefile/downloads/protobuf/        # 编译安装 protobuf
./autogen.sh
./configure
make
sudo make install
sudo ldconfig                                         # 刷新动态链接库缓存
cd ../../..
export HOST_NSYNC_LIB=`tensorflow/contrib/makefile/compile_nsync.sh` 
export TARGET_NSYNC_LIB="$HOST_NSYNC_LIB"
```

(3) 针对树莓派 3 Model B 型号，对编译项进行优化。具体命令如下：

```
make -f tensorflow/contrib/makefile/Makefile HOST_OS=PI TARGET=PI OPTFLAGS="-Os
-mfpu=neon-vfpv4 -funsafe-math-optimizations -ftree-vectorize" CXX=g++-4.8
```

执行该命令后，系统便进入较长的编译环节。

提示：

(1) 编译时要加上“CXX=g++-4.8”选项，否则会使用系统默认的“g++-4.9”，这样可能会出现一些“`_atomic_compare_exchange`”和“`malloc(): memory corruption`”等错误。

(2) 如果出现“`virtual memory exhausted: Cannot allocate memory`”这样的错误，则需要分配更大的虚拟内存。可以通过“`free -m`”命令查看内存使用情况。增大虚拟内存的具体操作如下：

```
cd /var
sudo swapoff /var/swap
sudo dd if=/dev/zero of=swap bs=1M count=2048      # 创建 2G 虚拟内存
sudo mkswap /var/swap
sudo swapon /var/swap
free -m                                         # 查看内存情况
total        used         free        shared    buff/cache   available
Mem:       927          332          505           9          88        536
Swap:      2047          130         1917
```

(3) 另外，编译时可以加上“-j2”选项，以提高编译速度。

(4) 编译完成之后安装 libjpeg：

```
sudo apt-get install -y libjpeg-dev
```

(5) 下载模型。

```
curl https://storage.googleapis.com/download.tensorflow.org/models/inception_
dec_2015_stripped.zip -o /tmp/inception_dec_2015_stripped.zip
unzip /tmp/inception_dec_2015_stripped.zip -d tensorflow/contrib/pi_examples/
label_image/
data/
```

(6) 编译例子程序。

```
cd ~/tensorflow
make -f tensorflow/contrib/pi_examples/label_image/Makefile
tensorflow/contrib/pi_examples/label_image/gen/bin/label_image # 尝试用默认的 Grace
Hopper 图像进行图像标注
I tensorflow/contrib/pi_examples/label_image/label_image.cc:384] Running model
succeeded!
I tensorflow/contrib/pi_examples/label_image/label_image.cc:284] military uniform
(866):
0.624293
```

```
I tensorflow/contrib/pi_examples/label_image/label_image.cc:284] suit (794):  
0.0473981  
I tensorflow/contrib/pi_examples/label_image/label_image.cc:284] academic gown  
(896):  
0.0280926  
I tensorflow/contrib/pi_examples/label_image/label_image.cc:284] bolo tie (940):  
0.0156956  
I tensorflow/contrib/pi_examples/label_image/label_image.cc:284] bearskin (849):  
0.0143348
```

3. 在树莓派上，通过 Docker 交叉编译源码的方式安装 TensorFlow

用 Docker 交叉编译源码的方式在树莓派上安装 TensorFlow 最为常用。这种方式利用 Docker 容器在性能强大的主机上虚拟出树莓派环境，并在该环境下进行编译 Tensorflow 的源码。这样可以大大提升编译的效率。这种方式也被叫作交叉编译。具体操作如下：

(1) 在 Ubuntu 16.04 LTS 64 中安装 Docker。

```
$ sudo apt-get install -y apt-transport-https ca-certificates curl  
software-properties-common  
#安装以上包，以使apt可以通过HTTPS使用存储库(repository)  
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - #  
添加Docker官方的GPG密钥  
$ sudo add-apt-repository "deb https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" #设置  
stable存储库  
$ sudo apt-get update #更新一下apt包索引  
$ sudo apt-get install -y docker-ce #安装最新版本的Docker CE  
$ sudo systemctl start docker #启动Docker服务  
$ sudo docker run hello-world #查看是否启动成功
```

上面命令执行之后，如果看到如图 13-49 所示的输出信息，则表示 Docker 软件已安装成功。

```
steve@steve-vm:~$ sudo docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

图 13-49 Docker 启动界面

(2) 用 GIT 工具下载 TensorFlow 源代码。具体如下：

```
git clone https://github.com/tensorflow/tensorflow.git  
cd tensorflow
```

(3) 用以下命令交叉编译 TensorFlow 源代码：

```
CI_DOCKER_EXTRA_PARAMS="-e CI_BUILD_PYTHON=python3 -e \
CROSSTOOL_PYTHON_INCLUDE_PATH=/usr/include/python3.5m" \
tensorflow/tools/ci_build/ci_build.sh PI-PYTHON3 \
tensorflow/tools/ci_build/pi/build_raspberry_pi.sh
```

该命令执行完后（约30分钟左右），会在output-artifacts目录下找到一个安装包文件“tensorflow-version-cp35-none-linux_armv7l.whl”。

(4) 将文件“tensorflow-version-cp35-none-linux_armv7l.whl”复制到树莓派中，并通过pip命令进行安装。具体命令如下：

```
pip3 install tensorflow-version-cp35-none-linux_armv7l.whl
```

13.5.3 编译并安装Protobuf

Protobuf是一种平台无关、语言无关、可扩展且轻便高效的序列化数据结构的协议，可以用于网络通信和数据存储。它独立于语言，独立于平台。目标检测API会用到处理Protobuf协议的软件包。这个包的名字是Protobuf。具体安装如下：

(1) 用以下命令安装一些依赖项：

```
sudo apt-get install autoconf automake libtool curl
```

(2) 用以下命令编译并安装Protobuf：

```
wget
https://github.com/google/protobuf/releases/download/v3.5.1/protobuf-all-3.5.1.
tar.gz
tar -zvxf protobuf-all-3.5.1.tar.gz
cd protobuf-3.5.1
./configure          #配置、编译并安装
make
sudo make install
cd python           #编译Python版protobuf
export LD_LIBRARY_PATH=../src/.libs
python3 setup.py build --cpp_implementation
python3 setup.py test --cpp_implementation
sudo python3 setup.py install --cpp_implementation
export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp
export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION_VERSION=3
sudo ldconfig
```

(3) 通过如下命令验证protobuf的安装情况：

```
protoc
```

该命令运行完，将会出现如下信息，则表示protobuf已经安装成功。

```
Usage: protoc [OPTION] PROTO_FILES
Parse PROTO_FILES and generate output based on the options given:
-IPATH, --proto_path=PATH  Specify the directory in which to search for
                           imports. May be specified multiple times;
```

```
directories will be searched in order. If not given, the current working directory is used.
```

13.5.4 安装 OpenCV

安装 OpenCV 的命令相对简单。具体如下：

(1) 安装依赖项，命令如下：

```
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install qt4-dev-tools
```

(2) 安装 OpenCV，命令如下：

```
pip3 install opencv-python
```

13.5.5 下载目标检测模型 SSDLite

所有的软件包安装完毕之后，便开始下载目标检测模型，这里使用 SSDLite 模型。该模型属于 TensorFlow 中 Object Detection 模块的一部分（见《深度学习之 TensorFlow——入门、原理与进阶实战》11.7 节）。在使用之前需要下载含有 Object Detection API 的 models 模块的源码。具体步骤如下：

(1) 在命令行中，用 GIT 工具下载 models 代码。具体命令如下：

```
mkdir tf-ws
cd tf-ws
git clone --recurse-submodules https://github.com/tensorflow/models.git
```

(2) 设置环境变量。具体命令如下：

```
sudo nano ~/.bashrc
export
PYTHONPATH=$PYTHONPATH:/home/pi/tf-ws/models/research:/home/pi/tf-ws/models/research/slim #在 ~/.bashrc 末尾增加
```

(3) 将 *.proto 文件转化为 *_pb2.py 文件：

```
cd /home/pi/tf-ws/models/research
protoc object_detection/protos/*.proto --python_out=.
```

(4) 下载/ssdlite_mobilenet_v2 模型文件：

```
cd /home/pi/tensorflow1/models/research/object_detection
wget
http://download.tensorflow.org/models/object_detection/ssdlite_mobilenet_v2_coco_2018_05_09.tar.gz
tar -xzvf ssdlite_mobilenet_v2_coco_2018_05_09.tar.gz
```

13.5.6 代码实现：用SSDLite模型进行目标检测

创建代码文件“13-4 Object_detection_usbcam.py”，并在其中添加调用代码，具体代码如下：

代码 13-4 Object_detection_usbcam

```

01 import os                                     #导入软件包
02 import cv2
03 import numpy as np
04 import tensorflow as tf
05 import sys
06 sys.path.append('..')                         #添加系统路径
07 from utils import label_map_util             #导入工具包
08 from utils import visualization_utils as vis_util
09 #设置摄像头分辨率
10 IM_WIDTH = 640                               #用较小的分辨率，可以得到较快的检测帧率
11 IM_HEIGHT = 480
12
13 MODEL_NAME = 'ssdlite_mobilenet_v2_coco_2018_05_09'#使用的模型名字
14
15 CWD_PATH = os.getcwd()                      #获取当前工作目录的路径
16 #获取 detect model 文件的路径
17 PATH_TO_CKPT = os.path.join(CWD_PATH,MODEL_NAME,'frozen_inference_graph.pb')
18
19 #获取 label map 文件路径
20 PATH_TO_LABELS = os.path.join(CWD_PATH,'data','mscoco_label_map.pbtxt')
21
22 NUM_CLASSES = 90                            #定义目标种类的数量
23
24 label_map = label_map_util.load_labelmap(PATH_TO_LABELS) #载入标签
25 categories = label_map_util.convert_label_map_to_categories(label_map,
   max_num_classes=NUM_CLASSES, use_display_name=True)
26 category_index = label_map_util.create_category_index(categories)
27
28 detection_graph = tf.Graph()                 #将模型加载到内存中
29 with detection_graph.as_default():
30     od_graph_def = tf.GraphDef()
31     with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
32         serialized_graph = fid.read()
33         od_graph_def.ParseFromString(serialized_graph)
34         tf.import_graph_def(od_graph_def, name='')
35     sess = tf.Session(graph=detection_graph)
36
37 #定义输入
38 image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
39

```

```

40 # 定义输出：检测框、种类的分值
41 detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
42 detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
43 detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')
44
45 # 获得目标种类的数量
46 num_detections = detection_graph.get_tensor_by_name('num_detections:0')
47
48 frame_rate_calc = 1 # 初始化帧率
49 freq = cv2.getTickFrequency()
50 font = cv2.FONT_HERSHEY_SIMPLEX
51
52 camera = cv2.VideoCapture(0) # 初始化 USB 摄像头
53 ret = camera.set(3, IM_WIDTH)
54 ret = camera.set(4, IM_HEIGHT)
55
56 while(True):
57     t1 = cv2.getTickCount()
58     ret, frame = camera.read() # 读取一张图片，并扩展维度成：[1, None, None, 3]
59     frame_expanded = np.expand_dims(frame, axis=0)
60
61     (boxes, scores, classes, num) = sess.run(# 运行模型
62         [detection_boxes, detection_scores, detection_classes, num_detections],
63         feed_dict={image_tensor: frame_expanded})
64
65     vis_util.visualize_boxes_and_labels_on_image_array( # 显示检测的结果
66         frame,
67         np.squeeze(boxes),
68         np.squeeze(classes).astype(np.int32),
69         np.squeeze(scores),
70         category_index,
71         use_normalized_coordinates=True,
72         line_thickness=8,
73         min_score_thresh=0.85)
74     # 显示帧率
75     cv2.putText(frame,"FPS: " + ("%.2f" % frame_rate_calc),(30,50),font,1,(255,255,0),2,cv2.LINE_AA)
76     cv2.imshow('Object detector', frame) # 显示图像
77
78     t2 = cv2.getTickCount()
79     time1 = (t2-t1)/freq
80     frame_rate_calc = 1/time1
81
82     if cv2.waitKey(1) == ord('q'): # 按 q 键退出
83         break
84 camera.release()
85 cv2.destroyAllWindows()

```

执行程序，显示结果如图 13-50 所示。该类识别模型（例如 YOLO、SSD）或是分离背景和背景的检测（例如 RGB-D 检测）。

其中，检测模型的精度和分离模型的精度相似度匹配。

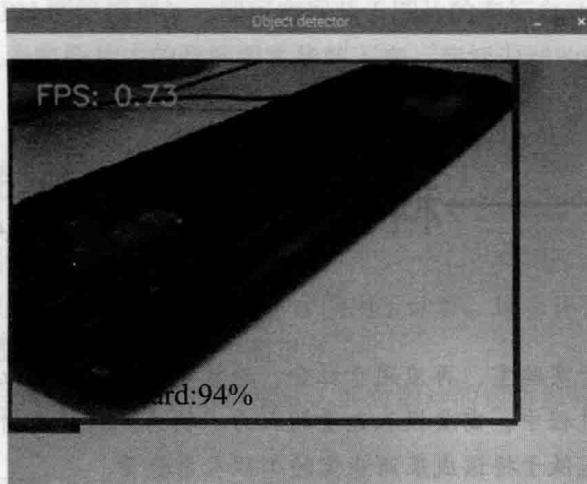


图 13-50 检测结果

如图 13-50 所示，图片上显示了帧率(FPS)为 0.73。图片中用矩形框标注了所识别的键盘，并在矩形框的左下角显示出 keyboard:94%。这表示，模型认为矩形框中的物体有 94% 的可能性是键盘。

在处理图片上，做了如下一些工作：

- 用数据增强方法对图片做对比度、膨胀、随机剪切等。
- 在编码部分，使用变分自动编码器。
- 事先将所有图中的字母图片进行特征提取，并保存在磁盘上。

长海苏市：「闻斗者来人由长海县来市，中唐以待商事市。长海的鼠标真其自来个一暴发，做件好事。」长海之属志商的增加堂表哥等，各士商请者甚集。长海县令作书告诫，不得有日逐相争的无妄之灾。已故提取的特征。」长海来时本市去假道，后公私相求，相拥，只消个夜半时分，长海县令便用各种理由去辞。」善归还桥孔时，去势，罪责出外。

14.1.5 用卷积神经网络识别物体 通过前面的分析，我们已经知道，卷积神经网络在识别物体方面，特别是识别静态图像时，比传统的分类器要好很多。

卷积神经网络最大的代价是需要对样本进行标注处理。因此其真实的应用范围最广泛的应用领域中会得到更好的表现。

人工标注样本的步骤如下：

- (1) 对于每张图片，根据标注的类别，将其标注为 1，否则标注为 0。
- (2) 用带有 `SwitchableNorm` 的一批类别的卷积网络和 `S` 来由最近邻算出每个样本 i 的特征的计算。

第 14 章

商业实例——科技源于生活，用于生活

好的科研成果诞生于实验室，再应用于社会，造福于人类。而商业化则是科研成果流到社会的重要途径。在这一过程中，需要投入大量的人力、物力。一般来讲，一个科研成果所需要的科研人员数量，会远远低于将该成果商业化的工程人员数量。

而人工智能当今的人才分布现状是，工程人员基数远远小于科研人员。这一缺口将是未来人才培养的驱动力。

本章将通过几个实际的实例，介绍一下人工智能在商业化中要经历的一些过程。其中包括做事的思路、遇到的问题及解决方案。

14.1 实例 71：将特征匹配技术应用在商标识别领域

本节将通过一个商标识别实例，讲解图片特征匹配任务的实现，以及该问题的处理细节和解决思路。

14.1.1 项目背景

这是一个来自某商标局的需求。在申请商标过程中，审核过程是由人来操作的。审核通过的商标将收到法律保护，不允许市面上有与其一样甚至相似的商标图案出现。一旦出现这种情况，则商标所属的公司可以通过法律手段来追责。

科技的发展，使得这种情况得以改善。在审核过程中，可以通过技术手段从成千上万个商标中发现和预申请商标类似的商标。这样可以从源头上控制商标冲突事件的发生。

同样，这类需求还可以泛化成为根据商品图片智能识别出该商品的所属品牌。本节以这种泛化后的任务需求来讲解实现过程。

14.1.2 技术方案

商标识别属于神经网络中的相似度匹配任务，与人脸识别的解决方案类似。具体做法是：

- (1) 将每个商标的特征提取出来。
- (2) 计算所有样本的特征之间的相似度。
- (3) 将商标按照相似度由大到小的顺序显示出来。

在完成核心算法之后，可以在前端使用目标识别模型（例如 YOLO、SSD）或是分离前景和背景的模型（例如 RPN 网络模型），即可实现基于图片的商标自动识别功能。

其中，前端模型负责将图片中的商标图案裁剪下来，商标识别的核心算法负责商标特征的相似度匹配。

14.1.3 预处理图片——统一尺寸

收集来的商标图案来自不同场景，大小不一。

必须将图标样本的尺寸调整成统一大小，才能用于计算。这个环节有两种方法：

- 通过 `resize` 函数将图片缩放到指定的尺寸。
- 保存高宽比，按照最大的边长进行缩放，并对图片的短边部分补 0（见 8.7 节的处理方式）。

这里建议用 `resize` 函数直接对图片尺寸进行调整。

14.1.4 用自编码网络加夹角余弦实现商标识别

用自编码网络压缩图片的特征，然后依次计算每两个图片特征的夹角余弦（见《深度学习之 TensorFlow：入门、原理与进阶实战》一书的 9.7.4 节）来实现商标的识别。这种方案在样本处理方面比较省劲，直接使用全量样本进行学习即可。

在处理细节上，做了如下一些工作：

- 将所有图片都归一化为 0~1 之间的浮点数。
- 用数据增强方法对图片做对比度、翻转、随机剪辑操作。
- 在自编码部分，使用变分自编码网络。
- 事先将所有库中的样本图片做好特征提取，并保存。在查找时，直接通过夹角余弦进行计算比对。

经过实验后发现该方案是可行的。唯一的弊端是：模型训练过程太长，收敛太慢。而且这种没有目标指导的无监督训练所提取的特征，并不能与人眼的相似度判定标准完全吻合。

为了进一步提升效果，采用有监督的方式进行训练，详见 14.1.5 小节。

14.1.5 用卷积网络加 triplet-loss 提升特征提取效果

监督式训练最大的代价是需要对样本进行标注处理。然而，人工标注后的样本在模型训练中会得到更好的表现。

人工标注样本的步骤如下：

- (1) 对现有的样本进行分类。将相同品牌的图标放在一起，如图 14-1 所示。
- (2) 用带有 SwitchableNor 归一化算法的卷积网络和 Swish 激活函数搭建网络模型，完成特征的计算。

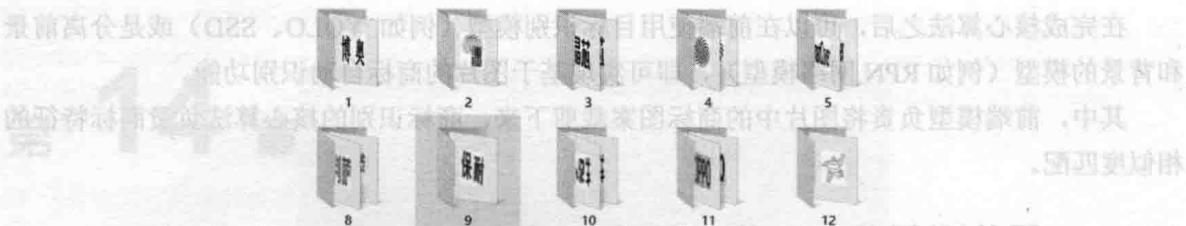


图 14-1 将样本分类存放

**提示：**

SwitchableNorm 归一化算法的介绍见 10.1.6 小节。激活函数 Swish 的介绍请参见《深度学习之 TensorFlow——入门、原理与进阶实战》一书 6.2.4 小节。

在损失值部分使用到损失函数 triplet-loss。在每次特征提取时，同步输入与该样本相同类别和非同类别的两个样本。利用监督学习，让该样本特征与同类的样本特征间的差异越来越小，与非同类样本特征间的差异越来越大，如图 14-2 所示。

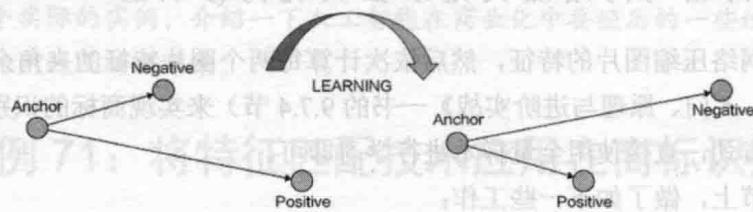


图 14-2 triplet-loss 损失函数

在图 14-2 中，Anchor 代表输入样本，Positive 与 Negative 分别代表了同步输入的同类样本与非同类样本。通过监督学习，让 Anchor 经过网络计算之后的特征与 Positive 的特征更近，与 Negative 特征更远。

提示：由于篇幅有限，这里不再详细介绍 triplet-loss 的代码细节。为读者推荐一个比较好的例子实现，链接如下：

<https://github.com/omoindrot/tensorflow-triplet-loss>

该代码用矩阵方式计算 triplet-loss，效率较高。读者可以自行研究。

利用改动之后的网络，可以解决 14.1.4 小节特征指向不明确的问题。

14.1.6 进一步的优化空间

该问题还可以理解成为一个图片搜索问题。在特征提取方面上，还可以有提升空间。例如，用特征学习效果比较好的胶囊网络代替卷积网络，或用带有对抗网络的自编码网络 AEGAN。

14.2 实例 72：用 RNN 抓取蠕虫病毒

本节将介绍一个应用在网络安全领域的 AI 实例。

14.2.1 项目背景

该项目源于本书作者在 2017 年发表的一篇文章。它的主要技术是用 RNN 发现恶意域名的特征。该项目背景在文章里已有详细介绍。具体链接如下：

https://github.com/jinhong0427/domain_malicious_detection

该文章介绍了项目背景，还介绍了编写 TensorFlow 代码的工程化静态图框架、训练及设计模型的一些心得，并提供了除模型以外的流程化代码。

本节属于该文章在应用层面的延伸——将文章中的技术用于实际的环境当中。

14.2.2 判断是否恶意域名不能只靠域名

所谓“没有不好的技术，只有不好的应用”。该项目是用 RNN 来拟合已有的恶意域名特征，从而发现与该特征一样的域名。其本质是域名间的特征匹配，并不是真正意义上的识别恶意域名。

之所以会有识别恶意域名的效果，是因为被匹配的域名都属于恶意域名。但是，被匹配的域名不代表全部的恶意域名。

如果将其错误的理解成“该模型能够发现恶意域名”并将其使用，那必然会效果很差。这里便解释了——为什么有些读者直接将其用于发现恶意域名得到的效果并不理想。

14.2.3 如何识别恶意域名

识别恶意域名的本质是识别恶意网站。最终还需要根据网站的信息内容来定性。判别恶意网站的特征有很多种，包括：内容、流量、IP、域名。所有该网站所带的信息都可以理解为该网站的特征。单纯从域名并不能完全识别。域名只是反映了恶意网站的一部分特征而已。所以，通过一个点来判定一条线，这本身就是伪命题。

在真正进行恶意域名识别时，通过域名特征来匹配恶意域名只是其中的一种技术手段。必须综合其他特征的识别，才能最终判定被检测域名是否是恶意的。

在使用时，可以将“根据域名特征来匹配恶意域名”技术用于对海量域名数据进行初步识别。并在模型报出的结果中找出正常的域名，将其补充到训练集的正向样本中，继续训练模型。另外，也需要将收集到的恶意域名补充到训练集的负向样本中。

在整个过程中，“根据域名特征来匹配恶意域名”这一技术的价值在于，能够从海量的域名里，找到值得关注的、具有恶意域名特征的域名。该技术相当于一张过滤网，从域名的角度来对海量域名进行一次过滤，从而大大缩小需要检测的域名范围。

同时，在对过滤后的域名内容进行检测时，会得到真实的标签信息，为过滤模型提供了更多可靠的训练样本。将这些样本补充到过滤模型的数据集里，又可以实现过滤模型再训练的自我升级，使模型越用越精确。

在真实项目中，该模型在系统中的架构如图 14-7 所示。

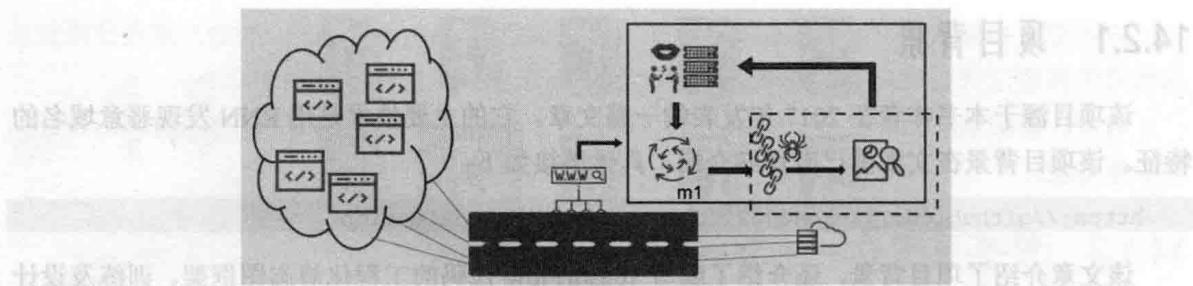


图 14-7 恶意域名检测系统架构图

在图 14-7 中， $m1$ 模型使用了“根据域名特征来匹配恶意域名”这项技术，被该模型筛查过的域名会被放入爬虫系统里进行二次判断。爬虫系统按照指定的域名列表爬取网站首页的内容，并将爬到的内容通过一个神经网络分类器模型来判别其是否为恶意网站。判别得到的结果会被作为 $m1$ 模型的训练数据集，以支持 $m1$ 模型的在线训练、自我更新。

14.3 实例 73：迎宾机器人的技术关注点——体验优先

如果要做一个迎宾机器人，你会怎么做？这里介绍一个来自真实项目的解决方案，希望读者通过该实例的学习，能够触类旁通，对于类似的项目可以有的放矢。

14.3.1 迎宾机器人的产品背景

制作一个迎宾机器人，将其放置在酒店、商场、机场等公共场所，为用户提供基本业务的咨询服务，提高用户体验。

迎宾机器人是一个将多种技术融于一体的科技产品。在实现这种产品之前，分析清楚自身的能力边界尤为重要。想清楚自己能做什么、不能做什么、哪些可以自己做、哪些需要对接外部的成熟模块。

下面以一套实际的商业产品为例，来介绍研发过程中的思路、方案，以及所遇到的问题和处理方法。

14.3.2 迎宾机器人的实现方案

由于商业规则的限制，这里不会透露任何技术以外的其他细节，会将该实例的技术思想泛化到行业技术视角来讲解。

整个产品可以拆分为机器人本体、外观、语音识别系统、内部的 AI 对话系统，以及行为交互几个功能。每个功能的实现方案如下：

1. 机器人本体及外观的技术实现方案

首先要对自己公司的主营业务方向做出明确的定位——公司是否是一个专业制作仿人机器人的公司。如果当前乃至未来没有这方面的规划，则应优先从外部寻找成熟的机器人本体，并根据使用场景，选择不同型号及外观的机器人本体。

2. 语音识别系统的实现方案

迎宾机器人的语音识别系统是一个非常大的挑战——它要求响应速度快并且容错率高（需要兼容更多的地方口音）。在某些特殊场景了，还需要它具有噪音分离、静音检测之类的声音预处理功能。

声音预处理部分相对比较独立。如果使用环境较安静，则可以直接使用现成的开源算法，做简单的去噪增强，并配合静音检测实现断句功能。如果使用环境吵架，则还得依靠集成专用硬件来解决。

语音识别过程一般做法是：将常用语与非常用语分成两个模型进行训练。

- 常用语模型内置在机器人本体，用于快速响应。
- 非常用语模型用于云端的精细化识别。

所有的输入都会先通过常用语模型识别一遍。常用语模型背后的控制算法决定是选择该识别结果，还是使用云端服务进行精确识别。为了加强用户体验，一般在使用云端识别时，都会同步让机器人发出“嗯、哦……”之类的语气词，以优化用户的等待体验。

控制语音识别结果的算法，是计算常用语的识别结果与常用语之间的加权语义相似度，并根据设定好的相似度阈值来决定是否匹配成功。如果匹配不到常用语，则将语音转发至云端。

其中的加权部分，来源于对话过程中的句子索引。因为在正常对话中，前几句出现常用语的概率比较高，所以用该索引计算出的衰减权重可以使匹配效果更好。

3. 智能对话系统的实现方案

智能对话系统的特点比较明显，因为使用的场景较为固定，都与具体业务相关，所以处理语音的内容不至于太过发散，机器人的知识储备内容相对可控。

该系统的最大的挑战是，用户的输入内容不可控，即无法控制用户的输入边界。遇到“不按套路出牌”的用户，会使机器人对话的体验直接拉低到0分。而这种“不按套路出牌”的用户在现实生活中经常出现。再者，用户的问题不会完全按照程序设置的方式输入，有些问题甚至还用到上下文相关的代词。这对于目前还不是完全成熟的AI算法是致命的。机器人会以一些不知所云或答非所问的内容来回答用户。这在真实场景中是绝对不允许发生的。

在这一环节中，并没有直接使用AI聊天机器人，而是用AI算法来关联“用户问题的语义”与“语料库中的标准问题语义”。

用AI算法直接回答用户的问题是很有挑战的，但是通过AI算法来猜测用户的意愿，并返回给用户做二次确认，是可以实现的。在实现细节上，可以利用语言的技巧，让用户自己对AI的结果再做一遍人工核对。例如：某用户问“哪里能让我洗个手？”AI算法可以根据语义匹配找到最相近的问题，并反问“您是想问洗手间在哪吗？”一旦用户说了“是的”，则系统便直接调出“洗手间在哪”对应的标准答案。

在实际应用中，这种算法达到了预期的效果。它的关键在于：将AI算法用在输入规范化环节，而不是投入大量的人力、财力去研发一套类似于人类客服的聊天机器人算法。

4. 人工接口的实现方案

业界所有的成熟迎宾机器人系统都有一个人工接口。通过该接口，后台的真人客服可以借

助迎宾机器人的外表来为用户服务。一方面可以给用户提供近乎完美的科技体验；另一方面这部分数据会用作语聊样本，不断提升算法精度。

在这种场景中，后台的客服只需对机器匹配出的问题进行二次确认，并直接选择对应的答案。只有在找不到该问题答案的情况下，才会用人工解答。这大大提升了服务效率。

纯人工成本太高，纯智能体验不好。在体验优先的需求之下，最佳的方案一定是人工与智能相结合。

5. 行为交互系统的实现方案

行为交互是指，机器人在完成本职工作的同时，表现出与人类相近的肢体行为。这会让机器像人一样对话。带有行为交互的机器人能够大大提升体验感。

一般需要做的几个重要功能包括：看着对话人的眼睛、将头转向对话人和与对话人保持一定的对话距离等。具体实现如下：

- 看着对话人的眼睛功能，通过前置摄像头配合人脸检测技术可以实现。找到摄像图像中人眼的位置，再调整对应的头部角度即可实现。
- 将头转向对话人功能，需要在头顶安装一圈 8 个方向的麦克风接收器，通过声音能量来计算旋转的角度。对于旋转较大的角度，有的还需要转身功能与其配合。这需要机器人本体的功能支撑。
- 与对话人保持一定的距离功能，这个一般由红外摄像头配合普通摄像头的人脸检测技术一起实现。红外摄像头主要负责测距，人脸检测技术负责目标。这一功能要控制机器人的移动范围。在公开场所下，通用的做法是将机器人限定在可控范围内，因为可移动机器人的商用化并不成熟（主要是在避障环节）。由于成本限制，机器人身上不可能搭载太多的激光雷达。而低成本的雷达，又有受限于发射材料的限制、覆盖盲区等问题，效果很不理想。

14.3.3 迎宾机器人的同类产品

人机交互类机器人，可以各种形态出现在我们的生活。以屏幕为主的软体机器人，商用价值往往较高。例如各种公共场所的电子广告屏幕、某些饭店的触摸屏点菜系统、银行理财推荐广告屏、移动营业厅的自助服务一体机等。如在原有的系统基础上将其拟人化，对客户来说都将是很好的体验。

在环境相对简单的场景中，移动机器人是有商用价值的。例如夜间的巡检机器人，可以按照指定的寻路轨迹搭载摄像头，并配合 YOLO 之类的目标识别算法，实现动态巡逻和及时报警功能。

随着人工智能的到来，越来越多的机器人产品将会陆续进入我们的生活。而机器人研发产业，也有巨大的商业空间。

14.4 实例 74：基于摄像头的路边停车场项目

路边停车场是指在道路两边的收费停车位。它没有固定的出口和进口，一般都是通过人工来收费和管理。这类停车场受效率低、高成本、监管难、记录数据缺失、容易漏收费等问题困扰。本节介绍一个低成本的解决方案。

14.4.1 项目背景

用科技手段来管理的路边停车场，一般有3种技术手段：地磁感应器、车桩识别器、路侧摄像头。相对来讲，地磁感应器与车桩识别器的普及度比较高。

但在某种场景中，路侧摄像头方法也有其不可取代的地位。下面从需求角度介绍路侧摄像头方式的适用场景。

1. 项目需求

来自香港客户的需求：随着城市的不断繁荣发展，一些具有特殊意义的老城街道，具有马路窄、停车位少、车辆多的特点。路边停车位建设是非常需要的，但是传统的管理模式必须要有大量的人工来维护，而地磁等高科技手段对空间的占用需求比较高。渴望得到一种可以减小人工又不会占用太多空间的解决方案。

来自北方寒冷区域的需求：路边停车场无法使用场地停车中的车牌识别技术来减少人工。目前可行的方案只能是地磁。但是由于北方寒冷的气候加上冰雪的覆盖，大大影响了地磁停车技术的灵敏度。再者，对设备的维护也需要更昂贵的费用。急迫需要使用视觉或其他技术手段实现停车管理方案。

2. 路侧摄像头方案

路侧摄像头方案是在路边的街灯杆或建筑物上安装摄像头，向马路对侧进行拍摄。通过图像识别的算法动态跟踪车位情况，从而实现车位的管理，如图 14-8 所示。

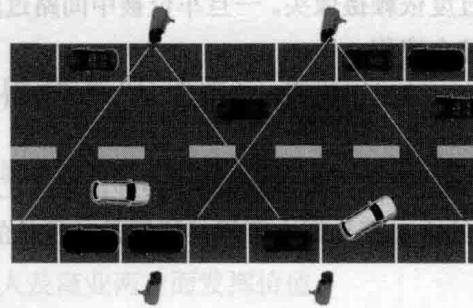


图 14-8 路侧摄像头技术方案

由于该方案是从旁侧拍摄，无法获得车牌号。车牌号的关联还需要靠人工解决。虽然该方案不能完全取代人工，但可以大大提升人工管理的效率。管理员只需要按照“管理员端 App”中的指示对车牌拍照，并上传到云端即可。计时和收费（通过关联账户的方式）等工作全由系

统自动完成。并且该方案从源头上获得了停车数据，避免了监管难、容易漏收费等问题。

用户也更加方便：不再需要与停车场管理员接触，可以即停即走。

14.4.2 技术方案

该方案主要使用的是目标识别算法。按照摄像头所拍摄的车位，在图像上划出需关注的坐标范围。根据车位的使用情况，输出每个车位的“空闲”“已占”两种状态。再根据车位在时间轴上的状态，判断出该车的“入库”“出库”行为。接着便可对车位进行计时、收费等相关流程。现场的应用情况如图 14-19 所示。



图 14-19 路边停车场现场

图 14-19 是路边停车场摄像头采集的图片。在该停车场中，一个摄像头管理 4 个车位。

14.4.3 方案缺陷

该方案的最大特点是成本低廉。平均下来，一个车位的建设成本不足地磁感应器或车桩识别器的 1/10。当然该方案也又不足之处，具体如下：

- 该方案只适用于中小型街道。对于 4 排车道以上的宽马路并不适合。
- 该方案的致命缺陷是过度依赖摄像头。一旦车位被中间路过的大车遮挡，车位的监管便会失效。这将会打乱整个流程。
- 摄像头的位置要求精确。这过于脆弱：当摄像头角度受到外部因素干扰而发生变化时，系统将无法管理车位。

任何实验室中出来的产品原型都不可能是完美的，必须再通过工程化的手段将其商业化后，才可以真正使用。将带有缺陷的方案变为真实可用，这便是工程化的价值。具体做法见 14.4.4 小节。

14.4.4 工程化补救方案

该项目的真正工作量并不是在算法部分，而是主要来自工程化部分。一旦进入市场进行使用，系统必须能够处理任何可能发生的异常。整个产品体验的各个环节都不能放过。这便是工作量的所在之处。具体如下：

- 前端硬件需要通过微处理器对多个摄像头进行分级管理，负责采集、预处理和上传。当然还包括维护、自检、告警、自动化配置等辅助功能。
- 通过后台系统对已经部署的车位、摄像头、微处理器进行统一编号管理。使其支持动态更新配置、维护、调用测试接口、实时发现告警、信息统计等操作。
- 使用大数据平台对终端图片进行统一管理。大数据平台支持快速存取即状态判定工作。
- 由于考虑到车位被遮挡的情况，对车位的判定状态做了复杂的设计。
- 通过算法发现车位被遮挡事件，并根据遮挡时长及时通知管理员，让其前去协调。
- 将停车记录照片（包括起始、结束）同步到用户终端，并支持申请退款功能，用于弥补系统异常给用户带来的损失。
- 在后端系统中，还要有停车场管理员的维护系统，包括管理员轮班制度、绩效指标等。其中的绩效包括：上传车牌的次数、错误率、漏传车牌的次数、上班时长等。
- 客户端的业务也是相当复杂。因为停车事件可以暴露个人行踪，所以要考虑隐私方面的因素。另外，一辆车可以由多个用户使用，一个用户也可以开多辆车。这里的关键是：在维持这种多对多的关系同时，还要通过权限控制实现每个用户的信息独立。

还有一些可能发生的异常现象，也都是由工程化环节所来解决的。例如：由于没有管理员当面收费，用户停完车后不付费或忘记付费也是常见的情况。通过自建征信系统或打通市政征信系统，解决欠费车辆与车主之间的联系。还要对车辆进行套牌检查，以保证欠费车辆的有效性等。

经过以上这些功能的开发，最终才可以实现系统的完整性，实现其商业价值。

14.5 实例 75：智能冰箱产品——硬件成本之痛

随着白色家电日趋普及，其价格变得越来越低，功能变得越来越多。冰箱——这一个改变人们生活方式的家电，已经走进了千家万户。使用者的基数决定了市场价值。虽然非智能领域的白色家电已经进入一片红海，但是，传统产品搭载人工智能将是一片商界蓝海，也是该领域众多厂商高度关注的方向之一。

14.5.1 智能冰箱系列的产品背景

白色家电智能化是人工智能产品在人们日常生活中的主要应用场景。

1. 商业价值

智能冰箱主要可以从个人及商业两方面发挥价值：

- 从个人服务方面，智能冰箱可以让机器了解人类对食物的存储及使用习惯，从而更好地管理饮食。
- 从商业应用方面，智能冰箱（或冰柜）可以做成售卖一体机、冷链终端的超市货架。

2. 技术方案

从技术角度来分析，这类场景都有以下共同的环节。

- 采集：通过摄像头、麦克风之类的输入设备，获取人类的原始行为数据。
- 处理：对原始行为数据做加工识别，变成结构化的可用信息。或者，把机器需要表达的信息转化为人类能够接受的信息方式。
- 分析：根据若干统计、关联、神经网络等算法，从可用信息中得到有价值的信息或数据。
- 呈现：通过网络终端以音频、图像、文字等方式回馈给人类。

在整个环节中，采集与呈现部分属于人机交互环节，需要基础硬件的支撑。而处理部分涉及人工智能技术。分析部分则属于人工智能在数据分析方面的技术。

这里重点讲解基于传统产品的技术改进方案（采集、处理部分）。对于分析环节，可以根据某个单一产品的具体定位、受众人群细分出更多的功能点和业务需求。这里不进行展开。

14.5.2 智能冰箱的技术基础

智能冰箱的工作流程相对简单。甚至稍有产品概念的用户都可以想到。但如果通过技术将其实现，则考验工程能力及集成能力。

1. 智能冰箱的工作流程

智能冰箱的工作流程如下：

冰箱打开门之后，启动监控流程。一旦发现有手伸进来拿东西，则开始拍照，并按照一定算法识别出“手伸进去”和“取出东西”这两个行为。并选出相对优质的图片，传入后端。后端通过 YOLO、SSD 之类的目标识别算法识别出具体的物体，形成有效记录信息。

后端会根据这些有效的信息记录并结合具体的业务场景进行运算，最终再将信息返给用户。

2. 基础硬件

如想将传统的冰箱智能化，需要添加以下基础硬件。

- 摄像相机：负责采集图片。一般的做法是：在冰箱内部安装 2、3 个摄像头。
- 前端逻辑控制器：用于控制摄像头、适配冰箱接口、与云端交互，以及实现整体的业务逻辑。它可以是一个单片机、工控机等设备。
- 神经网络处理器：专用于快速处理前端的采集数据。它是一个独立的计算单元。
- 网络模块：用于与云端交互。

14.5.3 真实的非功能性需求——低成本

白色家电市场的特点之一就是量大。这个特点直接决定了对人工智能技术的硬性要求——低成本。一台的成本降低 1 块钱，1 亿台直接就可以省出 1 亿元的成本。这对任何一个厂家都是不可忽视的问题。

1. 功能性需求决定着硬件的选取

至今为止，智能化技术所依赖的硬件还是比较昂贵的。因为在神经网络里需要进行大量的浮点运算，所以对算力有要求；因为图片的清晰度直接影响目标识别之类的图片算法，所以对拍摄相机有要求；因为用户完成取物品的时间较短，所以对整体的处理能力（包括网络速度）

有要求。尤其是公共售卖机，如果不能与用户在时间上同步，则大大影响客户体验。

这些便是该产品的真正需求点。它依赖人工智能的算法，但要求的不仅仅是精度。智能化需要使用更匹配实际需求的算法，并结合大量的工程化工作，才能够完成。

2. 非功能性需求阻碍了行业的发展

追溯起来，早先智能冰箱方案中，算法部分大多都是基于 `caffe` 框架实现的。当时的解决方案是：在前端用小型的工控机连接 2、3 个高速摄像机（需要采集高清图片，支持算法识别），再搭载英特尔的计算神经棒来实现（如果使用英伟达显卡，则成本将变得更高）。主板要求至少支持 8 线程（因为连接的外设较多，每个外设都需要单独的线程处理）。

整个方案所需的硬件成本已经突破 5000 元人民币，相当于一台中端传统冰箱的价格。而这也还不算开发系统的人工成本和冰箱本身的成本。

智能产品的人工研发成本也相当高。目前需要的是工程化的人才。他们主要的工作是对接并适配模型、标注、训练、测试、剪枝等工作。因为对于企业而言，将精度提高 1% 与将成本压缩 1% 相比，显然后者更有诱惑力。尤其是在开发前端的算法模块中，工程师们一直会尝试将 `YOLO`、`SSD` 之类的目标检测模型进行优化和精简，降低模型的运算需求。而并非我们常见的如何调优参数、增加准确率、提高模型训练收敛度之类的目标。这就是制造出一个智能化产品所需的工作与代价。

通过硬件和软件的投入成本可以看出，为什么市面上带有人工智能概念的家电几乎没有一万元以下的。为了一个看是锦上添花的功能，付出一倍以上的价格。这样的性价比显然不能让主流用户满意。

显然，在当今时代，智能冰箱之类产品的发展，已经被自身的市场价值所阻碍。从这一点看去，人工智能想要更广、更快地普及下，仍需要很长的路要走。

14.5.4 未来的技术趋势及应对策略

不断进步的科技总会给我们带来新的希望。新的技术体系的应用，在改进现有产品窘状的道路上从来没有停滞过，甚至在某些环节上已经能够降低成本。例如：用 `TensorFlow` 的 `lite` 模块对模型进行转化，使神经网络可以运行在“至强”系列主板或树莓派之类的低功耗主板上。这种方案在降低成本的同时，还简化了嵌入工控机的电源适配问题；用模糊图片优化算法配合普通相机，来替代高速相机等。

1. 高科技企业之痛

对于一个研发了几年的成熟产品线来讲，想要将新技术快速应用起来并非易事。如果框架和技术栈已经自成体系，则任何一个涉及框架级别的技术改动都需要付出巨大的代价。然而又有多少公司能大刀阔斧地将已有成果推倒重新再来。新技术的调头困难和旧技术的成本压力，将是这类公司永远的痛。

2. 应对策略

“真正看清科技发展局势，调整自有研发体系与之适应”是高科技产品的存活之本。人工

智能时代技术发展是飞速的。这要求企业的研发团队不仅要有超强的工程能力，还要有与时俱进的学习能力。追踪新技术、调整老框架将会是家常便饭。

在产品研发期间，构建灵活、高效的架构要优先于稳定健壮的架构。尤其是对于中小型企业或是大公司里的小规模独立团队来说，巨大的生存压力使其根本不允许在科研技术上进行过大的投入。凭借自身超强的工程化能力，将课题性技术转化为产品可用的商业技术，是人工智能时代大部分企业的大部分工作，也是小规模智能化企业的生存之道。只有大量的这种工程化人工智能企业崛起，才会实现真正推动人工智能的普及，才标志着人工智能时代的到来。

14.5.2 智能零售的技术需求

本章前半部分主要讨论了如何通过深度学习来提高零售业的效率，而本节将讨论如何通过深度学习来提高零售业的体验。零售业的体验主要体现在商品的陈列、包装、配送、售后服务等方面。商品的陈列是零售业的基础，商品的包装是商品的附加值，售后服务是商品的附加价值。商品的配送是商品的流通环节，售后服务是商品的售后保障。商品的售后服务是商品的附加价值，商品的配送是商品的流通环节，售后服务是商品的售后保障。

商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。

商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。

商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。

14.5.3 真实的非功能性需求——低成本

商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。

商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。商品的陈列是零售业的基础，商品的包装是商品的附加值，商品的配送是商品的流通环节，售后服务是商品的附加价值。

[General Information]

书名=a

SS号=14625781