案例一：基于2种迁移学习方法做鲜花分类并比较

1.案例目的

• 掌握将预训练的神经网络作为特征提取器的迁移学习方法；

• 掌握微调的迁移学习方法；

• 掌握逻辑回归模型；

• 掌握预训练的VGG16模型使用；

2.案例内容

本案例使用两种方式的迁移学习技术实现鲜花分类，一种是使用预训练的无头部的VGG16模型作为特征提取器提取图像特征，然后基于提取的特征建立逻辑回归模型做分类。另一种是基于预训练的VGG16网络模型使用微调的迁移学习方式训练鲜花分类模型。最后，对使用两种方法得到的模型进行对比。

3.案例知识点

• 预训练的神经网络作为特征提取器的迁移学习方法；

• 微调的迁移学习方法；

• 逻辑回归模型；

• 训练模型；

• 评估模型。

4.案例时长

共4学时，具体安排如下：

• 将预训练模型作为特征提取器的鲜花分类（1.5学时）

• 微调预训练网络的鲜花分类（2学时）

• 对比两种迁移学习方法得到的分类模型（0.5）

5.案例实验环境

**软件环境：**

• Python 3.7.4

• Tensorflow 2.3.0

• Keras 2.4.3

**开发环境与工具：**

• Spyder

6.案例分析

迁移学习作为机器学习的一个重要分支，侧重于将已经学习过的知识迁移应用于新的问题中。使用迁移学习技术可以使模型得到更高的准确率，也可以缩短模型的训练时间。迁移学习可以应用于图像分类、目标检测、图像分割等多个计算机视觉领域。

本案例分为以下两个主要部分：

1） 将预训练模型作为特征提取器的鲜花分类

使用预训练的无头部的VGG16模型作为特征提取器提取图像特征，然后基于提取的特征建立逻辑回归模型做分类。

2） 微调预训练网络的鲜花分类

基于预训练的VGG16网络模型使用微调的迁移学习方式训练鲜花分类模型。

3） 对比两种迁移学习方法得到的分类模型

在同样的实验环境下，比较两种迁移学习方式训练模型的速度及模型分类性能。

7.案例实验过程

7.1开发准备

在桌面上新建一个目录，名字叫flowers-Classification。本案例中所有的数据集和代码都放在该目录下。

7.1.1 获取数据

获取数据之前，在flowers-Classification目录下新建一个目录，命名为dataset。获取到的数据将存放在此目录下。

本案例需要1个数据集，是17种花的图片。

点击右侧菜单栏中的【资源下载】按钮，复制提供的URL前缀，如https://xxx.eec-cn.com/。

将URL前缀和数据集地址拼接到一起，如https://xxx.eec-cn.com/dataSet/systemLib/a6489b2ca7da43bdac1d56e1ae9b5a2b.zip下载数据集。将下载后的数据集解压，得到flowers17目录。

将flowers17目录直接复制到刚刚创建的flowers-Classification/dataset目录下。

7.1.2 打开Spyder软件

在桌面上双击Spyder图标打开Spyder，并把工作目录切换至flowers-Classification目录下。

7.1.3 新建代码文件

确保Spyder的工作目录为flowers-Classification。使用Spyder新建6个Python文件，分别命名为check\_hdf5.py、extract\_features.py、hdf5datasetwriter.py、train\_model.py、finetune\_flowers17.py、 inspect\_model.py。

7.1.4 新建图像输出目录

在flowers-Classification目录下新建一个目录，命名为output。生成的.hdf5文件和分类结果图会保存在该目录下。

7.1.5 预训练文件

本案例需要1个预训练的VGG16网络模型，为 vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5。

点击右侧菜单栏中的【资源下载】按钮，复制提供的URL前缀，如https://xxx.eec-cn.com/。

将URL前缀和数据集地址拼接到一起，如https://xxx.eec-cn.com/dataSet/systemLib/e93be86b973a4db7afee05f60727738e.h5下载模型，下载后得到文件vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5。

将vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5复制到/root/.keras/models目录下。

7.1.6 辅助文件

本实验需要多个辅助文件，这些文件存放在computervision目录下。

点击右侧菜单栏中的【资源下载】按钮，复制提供的URL前缀，如https://xxx.eec-cn.com/。

将URL前缀和数据集地址拼接到一起，如https://xxx.eec-cn.com/dataSet/systemLib/095f12c3e81e46cdba9a7af84ad0960a.zip下载，下载后解压，得到computervision目录。

我们直接将computervision目录完整的复制到flowers-Classification目录下。

至此，整个案例的目录结构如图1所示。

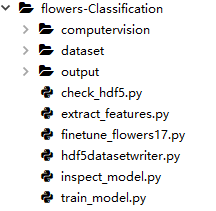


图1 案例目录结构

7.1.7 了解数据

本案例用的鲜花数据集共包含17种鲜花类型，分别为bluebell（蓝铃）、buttercup（毛茛）、coltsfoot（秋葵）、cowslip（牛蒡）、crocus（番红花）、daffodil（水仙花）、daisy（雏菊）、dandelion（蒲公英）、fritillary（贝母）、iris（鸢尾花）、lilyvalley（百合）、pansy（三色堇）、snowdrop（雪莲）、sunflower（向日葵）、tigerlily（虎耳草）、tulip（郁金香）、windflower（风花），每个类别都有80张图像，分别放在对应的目录下，如图2所示。

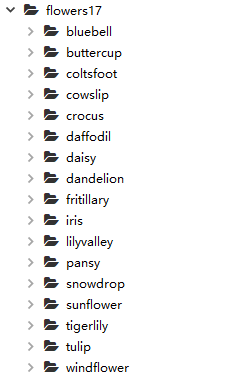


图2 flowers17数据集样式

以bluebell为例，展示部分图像如图3所示，其余类别类似。



图3 部分bluebell图像

7.2将预训练模型作为特征提取器的鲜花分类

该模块主要实现的是使用预训练的无头部的VGG16模型作为特征提取器提取图像特征，然后基于提取的特征建立逻辑回归模型进行鲜花分类。

7.2.1 定义hdf5文件

打开hdf5datasetwriter.py。正如模块名字所提示的那样，该模块定义了如何向hdf5添加数据。此文件不需要重点掌握，直接提供代码。

【代码7-2-1】hdf5datasetwriter.py

# import the necessary packages

import h5py

import os

class HDF5DatasetWriter:

def \_\_init\_\_(self, dims, outputPath, dataKey="images",bufSize=1000):

# check to see if the output path exists, and if so, raise

# an exception

if os.path.exists(outputPath):

raise ValueError("The supplied `outputPath` already "

"exists and cannot be overwritten. Manually delete "

"the file before continuing.", outputPath)

# open the HDF5 database for writing and create two datasets:

# one to store the images/features and another to store the

# class labels

self.db = h5py.File(outputPath, "w")

self.data = self.db.create\_dataset(dataKey, dims,dtype="float")

self.labels = self.db.create\_dataset("labels", (dims[0],),dtype="int")

# store the buffer size, then initialize the buffer itself

# along with the index into the datasets

self.bufSize = bufSize

self.buffer = {"data": [], "labels": []}

self.idx = 0

def add(self, rows, labels):

# add the rows and labels to the buffer

self.buffer["data"].extend(rows)

self.buffer["labels"].extend(labels)

# check to see if the buffer needs to be flushed to disk

if len(self.buffer["data"]) >= self.bufSize:

self.flush()

def flush(self):

# write the buffers to disk then reset the buffer

i = self.idx + len(self.buffer["data"])

self.data[self.idx:i] = self.buffer["data"]

self.labels[self.idx:i] = self.buffer["labels"]

self.idx = i

self.buffer = {"data": [], "labels": []}

def storeClassLabels(self, classLabels):

# create a dataset to store the actual class label names,

# then store the class labels

dt = h5py.special\_dtype(vlen=str) # `vlen=unicode` for Py2.7

labelSet = self.db.create\_dataset("label\_names",(len(classLabels),), dtype=dt)

labelSet[:] = classLabels

def close(self):

# check to see if there are any other entries in the buffer

# that need to be flushed to disk

if len(self.buffer["data"]) > 0:

self.flush()

# close the dataset

self.db.close()

代码不需要执行，从鲜花数据集提取图像特征时会调用该模块。

7.2.2 从鲜花数据集提取图像特征

接下来向extract\_features.py文件中添加代码，该模块的作用是从鲜花数据集中提取图像特征，并将特征保存在hdf5文件中。

7.2.2.1 导入库

把所有用到的库都导入。

【代码7-2-2-1】extract\_features.py

# 导入库

from keras.applications import VGG16

from keras.applications import imagenet\_utils

from keras.preprocessing.image import img\_to\_array

from keras.preprocessing.image import load\_img

from sklearn.preprocessing import LabelEncoder

from hdf5datasetwriter import HDF5DatasetWriter

from imutils import paths

import numpy as np

import progressbar

import random

import os

执行代码块。

7.2.2.2 设置全局常量

这些常量都是的一些指定参数，BATCH\_SIZE是处理图像时批的大小，BUFFER\_SIZE是生成.hdf5文件时存储缓冲区的大小。

【代码7-2-2-2】extract\_features.py

# 全局常量

BATCH\_SIZE = 32

BUFFER\_SIZE = 1000

执行代码块。

7.2.2.3 设置全局变量

这些变量用于指定数据源的路径、.hdf5文件保存的路径。设置为全局变量比较方便使用。

【代码7-2-2-3】extract\_features.py

# 全局变量

dataset\_path = 'dataset/flowers17'

output\_path = 'output/flowers17\_features.hdf5'

执行代码块。

7.2.2.4 打乱图像顺序

打乱图片顺序。

【代码7-2-2-4】extract\_features.py

# 获取图像名字，并打乱顺序

print("[INFO] 导入图像...")

imagePaths = list(paths.list\_images(dataset\_path))

random.shuffle(imagePaths)

执行代码块。

7.2.2.5 提取图像的标签

在该模块，我们使用LabelEncoder将标签分配一个0-n\_classes-1之间的编码，将各种标签分配一个可数的连续编号，因为flowers17数据集有17个鲜花种类，因此，标签将分配0-16之间的编码。

【代码7-2-2-5】extract\_features.py

# 提取图像的标签

labels = [p.split(os.path.sep)[-2] for p in imagePaths]

le = LabelEncoder()

labels = le.fit\_transform(labels)

执行代码块。

此时建议切换到Spyder的Variable explorer面板，查看变量信息。

执行labels = [p.split(os.path.sep)[-2] for p in imagePaths]，通过分割符划分字符串，并获得指定字符串，即可得到每张图片的类别信息。1360张图片的类别信息如图4所示。

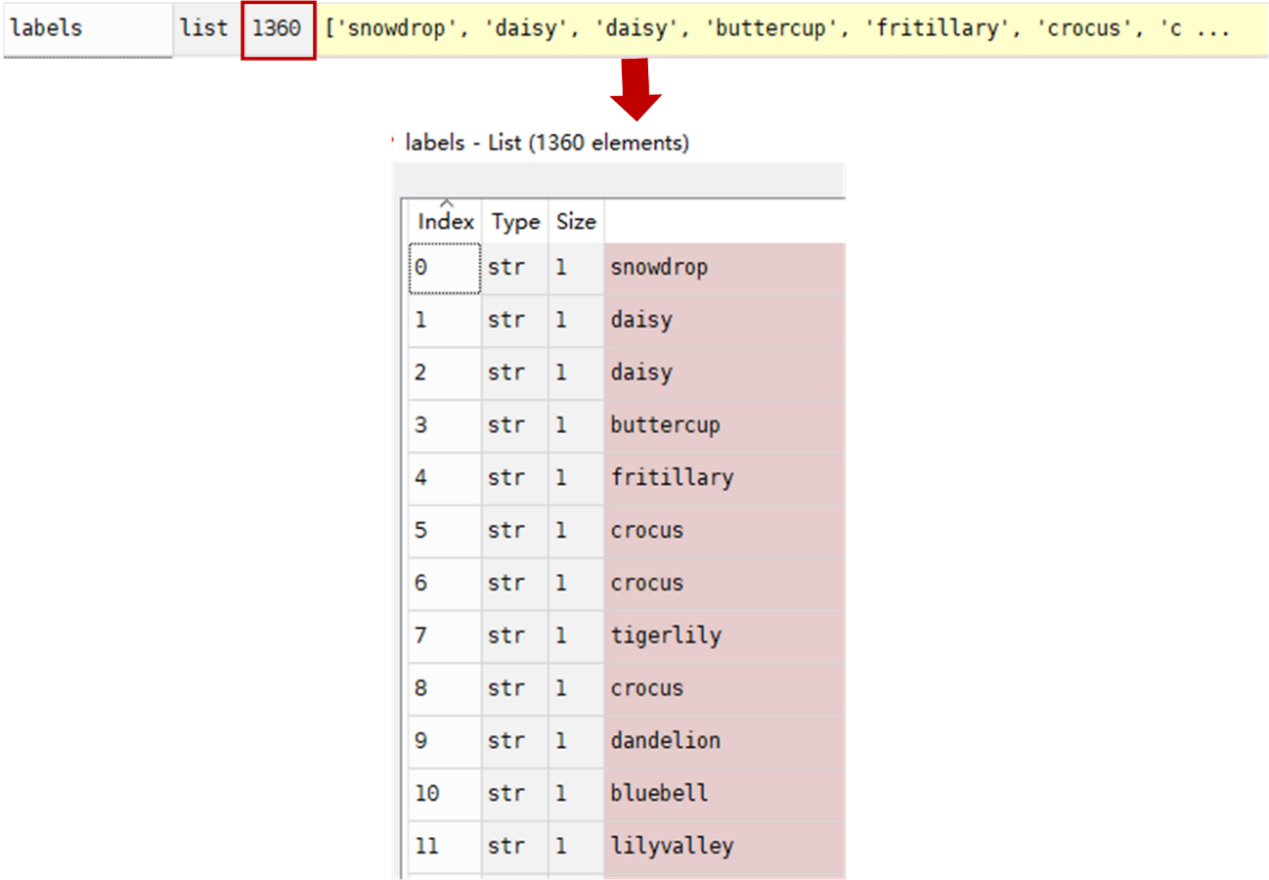


图4 labels变量的值

由图4可以看到，我们得到的标签是字符串，非数字化的，因此，需要用LabelEncoder对标签进行编号，1360张图像的labels变量信息如图5所示。



图5 标签编码后labels变量的值

7.2.2.6 加载VGG16网络

本案例使用的是在Imagenet数据集上预训练的且不包含3层全连接层的VGG16网络提取图像特征，如果本地不存在该模型，那么会自动从网络上下载。

【代码7-2-2-6】extract\_features.py

# 加载 VGG16 网络

print("[INFO] 加载 VGG16 网络...")

model = VGG16(weights="imagenet", include\_top=False) # VGG16是去掉头部的（去掉FC层）

执行代码块。

7.2.2.7 初始化HDF5类

该模块的作用是初始化HDF5类，在output目录下会生成.hdf5初始文件。

【代码7-2-2-7】extract\_features.py

# 初始化HDF5类

dataset = HDF5DatasetWriter((len(imagePaths), 512 \* 7 \* 7),output\_path, dataKey="features", bufSize=BUFFER\_SIZE)

执行代码块。

7.2.2.8 写入标签

该模块的目的是将标签信息写入.hdf5文件.

【代码7-2-2-8】extract\_features.py

# 写入标签

dataset.storeClassLabels(le.classes\_)

执行代码块。

7.2.2.9 初始化进度条

初始化进度条，以便后续更直观的看到提取特征的进度。

【代码7-2-2-9】extract\_features.py

# 初始化进度条

widgets = ["Extracting Features: ", progressbar.Percentage(), " ",progressbar.Bar(), " ", progressbar.ETA()]

pbar = progressbar.ProgressBar(maxval=len(imagePaths),widgets=widgets).start()

执行代码块。

初始化进度条的样式如图6所示。



图6 初始化进度条

7.2.2.10提取图像特征

该部分的目的是使用预训练好的VGG16的无头部模型提取图像特征，具体步骤如下：

每次循环遍历一批图片及其标签。

【代码7-2-2-10】extract\_features.py

# 循环遍历每一批图片

for i in np.arange(0, len(imagePaths), BATCH\_SIZE):

batchPaths = imagePaths[i:i + BATCH\_SIZE]

batchLabels = labels[i:i + BATCH\_SIZE]

batchImages = []

执行代码块。

每批图片的标签和路径信息分别存储在batchPaths和batchLabels变量中，如图7所示。这里需要说明一点，前面设置的BATCH\_SIZE是32，所以一批图片应该有32张，但是因为一共有1360张图片，1360/32不能整除，所以最后一个BATCH\_SIZE只有16张图片。

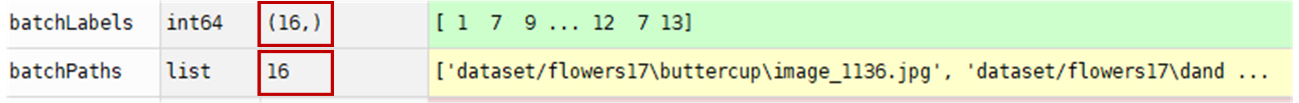


图7 一批图片的batchPaths和batchLabels变量

循环遍历这批图片中的每一张图片，因为VGG16模型需要的输入大小为224x224，所以，需确保每张图像的大小为224x224。

【代码7-2-2-10】extract\_features.py

# 循环遍历这批图片中的每一张图片

for (j, imagePath) in enumerate(batchPaths):

# 确保输入尺寸是224x224，这是VGG16的输入图像尺寸

image = load\_img(imagePath, target\_size=(224, 224))

image = img\_to\_array(image)

执行代码块。

每张图像的像素信息存储在image变量中，如图8所示。



图8 图像的像素信息存储在image变量中

使用Keras自带的图像预处理函数preprocess\_input()，执行mean subtraction运算。

【代码7-2-2-10】extract\_features.py

# 使用图像预处理函数preprocess\_input()，执行mean subtraction运算

image = np.expand\_dims(image, axis=0)

image = imagenet\_utils.preprocess\_input(image)

执行代码块。

预处理后的图像信息变化，如图9所示。



图9 预处理后的image变量

将预处理完的单张图片加入到batchImages变量中。

【代码7-2-2-10】extract\_features.py

# 把单张图片加入到这一批中

batchImages.append(image)

执行代码块。

将这批图片传入到网络中，提取图像特征，并保存在features变量中。

【代码7-2-2-10】extract\_features.py

# 把这批图片传入到网络中

batchImages = np.vstack(batchImages)

features = model.predict(batchImages, batch\_size=BATCH\_SIZE)

执行代码块。

VGG16的输入为224x224-3的图像，经过VGG16的13层卷积层以5个池化层后，输出其中一张图像的特征图为7x7-512。features变量如图10所示。



图10 提取特征后的features变量

平铺特征，将每张图像的特征平铺为一行，25088列，这是一批图像中的其中一张，所以是一行。最终执行整个for循环的时候，一批图像中有多少张，就有多少行。

【代码7-2-2-10】extract\_features.py

# 存储平铺的特征

features = features.reshape((features.shape[0], 512 \* 7 \* 7))

执行代码块。

平铺特征后的features变量如图11所示。



图11 平铺特征后的features变量

把特征和标签写入.hdf5文件中。

【代码7-2-2-10】extract\_features.py

# 把特征和标签加入到 HDF5 中

dataset.add(features, batchLabels)

pbar.update(i)

关闭数据集和停止进度条。

【代码7-2-2-10】extract\_features.py

# 关闭数据集

dataset.close()

# 停止进度条

pbar.finish()

注意，请同学们重新选中上述7.2.2.10节中的全部代码，并执行代码块。

至此，图像特征提取完成，flowers17\_features.hdf5文件完成，可以看到控制台输出特征提取过程进度条如图12所示。提取图像特征的过程大约2分钟。

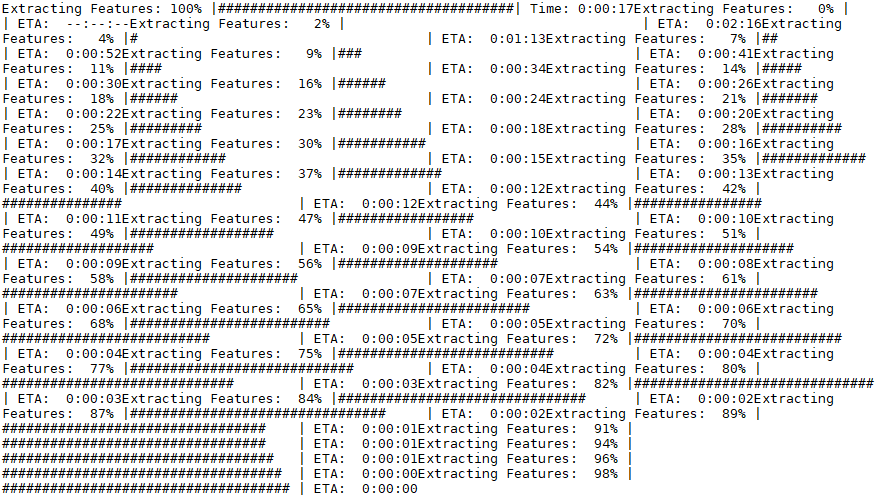


图12 控制台输出特征提取过程进度条

执行extract\_features.py文件时，执行过程中若出现ValueError: ('The supplied `outputPath` already exists and cannot be overwritten. Manually delete the file before continuing.', 'output/flowers17\_features.hdf5')这个错误时，需要手动是删除output目录下生成的flowers17\_features.hdf5文件。

7.2.3 查看HDF5文件

接下来向check\_hdf5.py文件中添加代码，该模块的作用是查看生成的hdf5文件，以更清晰的了解该文件。

【代码7-2-3】check\_hdf5.py

import h5py

p = 'output/flowers17\_features.hdf5'

db = h5py.File(p)

print(list(db.keys())) # ['features', 'label\_names', 'labels']

执行完毕后，如图13所示，可以看到hdf5文件中存储了features、label\_names、labels的信息。



图13 hdf5文件中的关键存储信息

【代码7-2-3】check\_hdf5.py

print(db["features"].shape) # (1360, 25088) (25088=512 \* 7 \* 7)

print(db["labels"].shape) # (1360,)

print(db["label\_names"].shape) # (17,)

执行代码块。

执行代码后，控制台输出如图14所示的信息，features的大小为(1360,25088)，其中(25088=512\*7\*7)；labels变量的大小为(1360,)，1360对应flowers17数据集中的所有图像，label\_names的大小为(17,)，17是flowers17数据集中有17种类别的鲜花。



图14 hdf5文件中变量的shape

7.2.4 基于提取的特征建立逻辑回归模型做分类

接下来向train\_model.py文件中添加代码，该模块的作用基于提取的特征建立逻辑回归模型做分类，并评估模型的性能。

7.2.4.1 导入库

把所有用到的库都导入。

【代码7-2-4-1】train\_model.py

# 导入包

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

import h5py

import time

import math

执行代码块。

7.2.4.2 设置全局变量

此参数用于指定.hdf5文件路径。

【代码7-2-4-2】train\_model.py

# 全局变量

feature\_path = 'output/flowers17\_features.hdf5'

执行代码块。

7.2.4.3 拆分数据集

将数据集拆分成训练集和测试集，拆分比例是3：1。

【代码7-2-4-3】train\_model.py

# 75%数据做训练集，25%数据做测试集

db = h5py.File(feature\_path, "r")

i = int(db["labels"].shape[0] \* 0.75)

执行代码块。

此时训练集有1020条记录，测试集有340条记录。

7.2.4.4 训练模型

基于保存图像特征的.hdf5文件，训练逻辑回归模型，并记录训练开始和结束时间，最后计算模型训练的用时。

【代码7-2-4-4】train\_model.py

# 训练模型

print("[INFO] 训练模型...")

start\_time = time.time()

model = LogisticRegression()

model.fit(db["features"][:i], db["labels"][:i])

end\_time = time.time()

print('Used %d minutes' % math.ceil((end\_time - start\_time)/60))

执行代码块。

模型训练很快，用时大约在1分钟左右。

7.2.4.5 评估模型

训练完成后，我们在测试集上评估模型性能。

【代码7-2-4-5】train\_model.py

# 评估模型

print("[INFO] evaluating...")

preds = model.predict(db["features"][i:])

print(classification\_report(db["labels"][i:], preds,target\_names=db["label\_names"]))

# 关闭数据集

db.close()

执行代码块。

性能评估结果如图15所示：

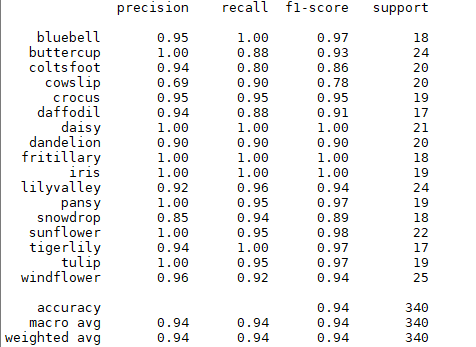


图15 分类测试结果

从图15结果可以看出，在测试集的340张图像上，每个类别的precision、recall、f1-sorce的值大部分都在0.8以上，accuracy、macro avg（加权平均）、weighted avg（加权平均）都为0.94，说明分类性能已经非常不错了。

7.3微调预训练网络的鲜花分类

该模块的主要作用是基于预训练的VGG16网络模型使用微调的迁移学习方式训练鲜花分类模型。

7.3.1 VGG16结构

因为本案例基于VGG16的分类模型，VGG16包含13层卷积层和3层全连接层，我们采用在Imagenet数据集上训练好的模型，但我们可以输出其结构看一下，帮助理解VGG16的网络结构。我们直接向inspect\_model.py文件中添加代码。

7.3.1.1 导入库

把所需的库都导入。

【代码7-3-1-1】inspect\_model.py

# 导入库

from keras.applications import VGG16

执行代码块。

7.3.1.2 加载VGG16网络

头部指的是网络结构中的全连接层，代码中若INCLUDE\_TOP =True，则表示下载的模型中包含全连接层，若INCLUDE\_TOP = False，则表示下载的模型中不包含全连接层，因为本案例需要重新训练全连接层，因此选用False。

【代码7-3-1-2】inspect\_model.py

INCLUDE\_TOP = False # if True, FC layers are included; if False, FC layers are excluded.

# 加载VGG16网络（如果VGG16不存在，那么自动从网上下载）（16的含义是：13个卷积层+3个全连接层）

print("[INFO] 加载VGG16网络...")

model = VGG16(weights="imagenet", include\_top = INCLUDE\_TOP)

print("[INFO] 显示VGG16的层...")

print("[INFO] 显示输入层：")

print(model.input) # 输入层

# 遍历显示VGG16的层

print("[INFO] 显示隐藏层：")

for (i, layer) in enumerate(model.layers):

print("[INFO] {}\t{}".format(i, layer)) # 隐藏层

print("[INFO] 显示输出层：")

print(model.output) # 输出层

执行代码块。

当INCLUDE\_TOP = False时，如图16所示，可以看到模型中仅包含卷积层和池化层。

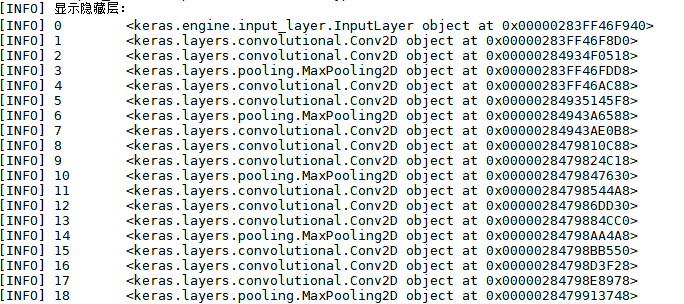


图16 无头部VGG16模型

为了更完整的了解VGG16的网络结构，我们将INCLUDE\_TOP = False改为INCLUDE\_TOP = True，输出结果如图17所示。输入数据大小为224 x 224，RGB图像通道数为3。隐藏层为所有的卷积层、池化层、全连接层。输出层采用softmax进行分类，该模型是在Imagenet数据集上训练得到的，Imagenet数据集包含1000个类别。切记，本案例需要全连接层的模型，即INCLUDE\_TOP = False。

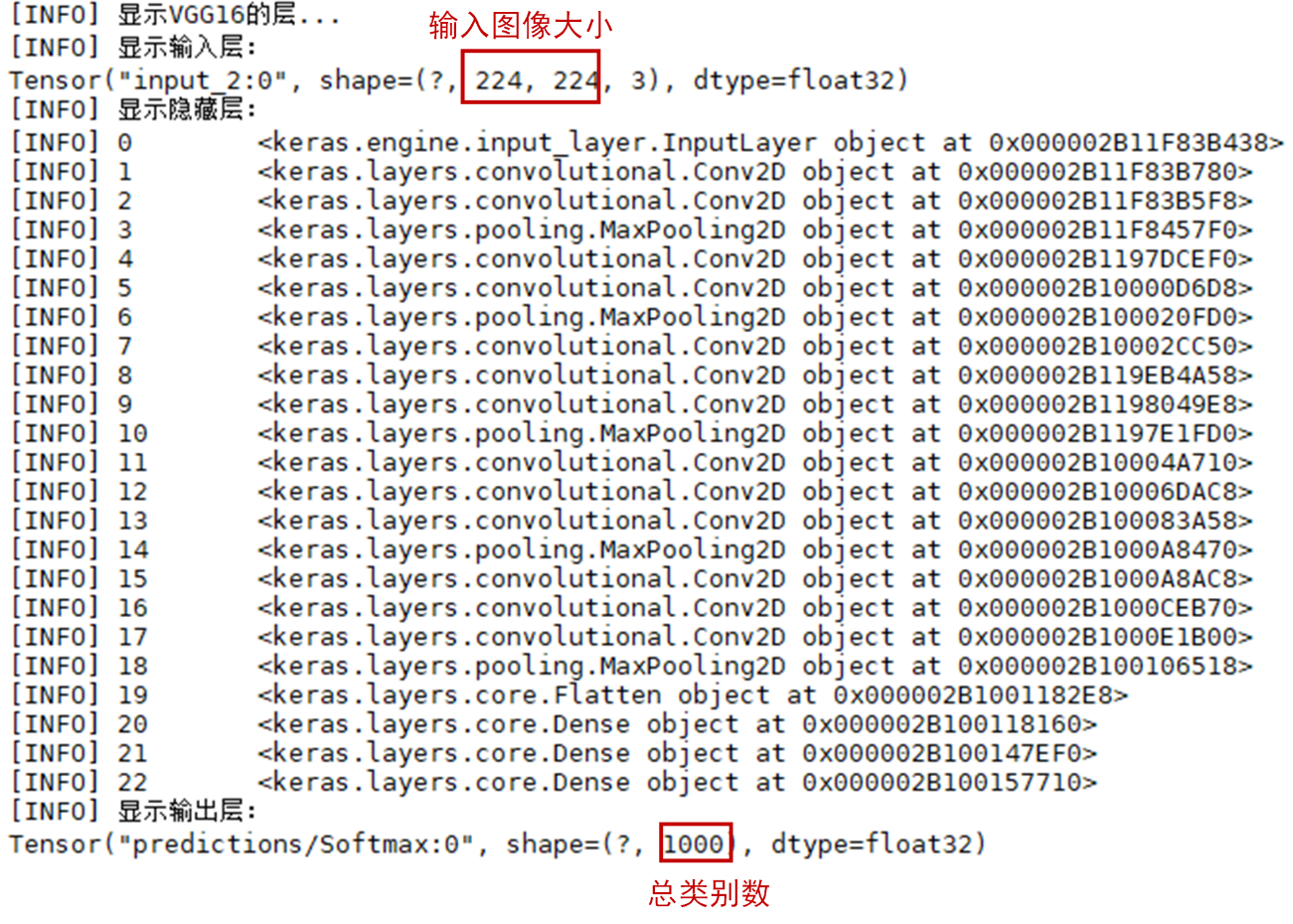


图17 控制台输出VGG16网络模型

VGG16总共有16层，13个卷积层和3个全连接层，第一次经过64个卷积核的两次卷积后，采用一次pooling，第二次经过两次128个卷积核卷积后，再采用pooling，再重复两次三个512个卷积核卷积后，再pooling，最后经过三次全连接。附上官方的VGG16网络结构图，如图18所示。

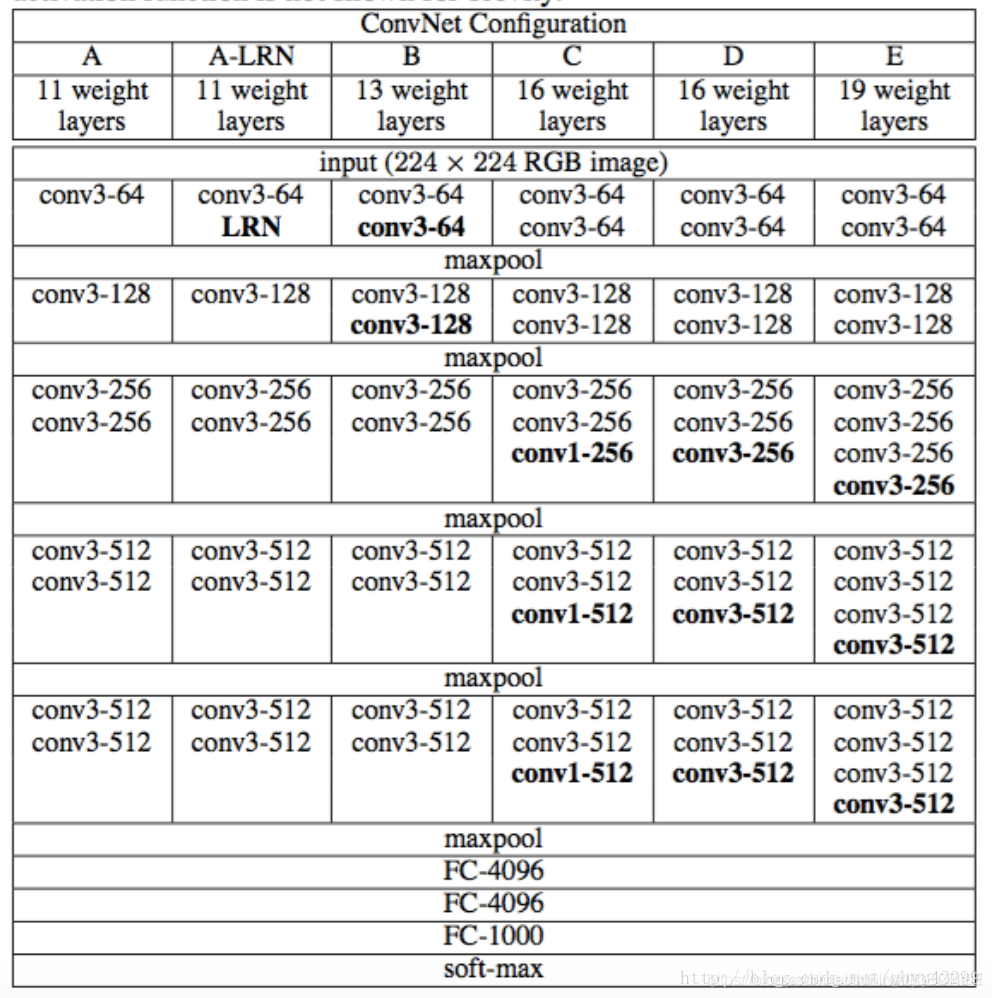


图18 官方的VGG16网络结构图

7.3.2 训练鲜花分类模型

接下来向finetune\_flowers17.py文件中添加代码，该模块的作用是完成基于VGG16网络模型使用微调的迁移学习方式训练鲜花分类模型，并进行验证及评估对比各模型的性能。

7.3.2.1 导入库

把所有用到的库都导入。

【代码7-3-2-1】finetune\_flowers17.py

# 导入库

from sklearn.preprocessing import LabelBinarizer

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report

from computervision.callbacks import TrainingMonitor

from computervision.preprocessing import ImageToArrayPreprocessor

from computervision.preprocessing import AspectAwarePreprocessor

from computervision.datasets import SimpleDatasetLoader

from computervision.nn.conv import FCHeadNet

from keras.preprocessing.image import ImageDataGenerator

from keras.optimizers import RMSprop

from keras.optimizers import SGD

from keras.applications import VGG16

from keras.layers import Input

from keras.models import Model

from imutils import paths

import numpy as np

import os

执行代码块。

7.3.2.2 设置模型超参数

这些超参数都是训练模型的一些指定参数，像一批加载几张图片，学习率是多少。如果模型性能不好或者出现其他问题，我们需要调整这些参数。所以把这些参数统一放在一起作为常量。

EPOCHS参数是影响模型训练时间最主要的因素。本案例中（使用CPU训练，不用GPU训练）有两个关于EPOCH的参数，WARM\_UP\_EPOCHS=25是只训练新加入的FC层25个EPOCH。FINE\_TUNING\_EPOCHS = 100是设置微调网络模型100个EPOCH。可以更改EPOCHS参数的值，增大或减少，但是在本案例中若训练 EPOCH的数量数不足，模型会欠拟合。

【代码7-3-2-2】finetune\_flowers17.py

# 模型超参数

WARM\_UP\_EPOCHS = 25

FINE\_TUNING\_EPOCHS = 100

BATCH\_SIZE = 32

LR = 0.001

执行代码块。

7.3.2.3 设置全局变量

这些变量用于指定数据源的路径、loss和accuracy曲线保存的路径。设置为全局变量比较方便使用。

【代码7-3-2-3】finetune\_flowers17.py

# 全局变量

dataset\_path = 'dataset/flowers17'

warm\_up\_plot = 'output/flowers17\_warmup.png'

fine\_tuning\_plot = 'output/flowers17\_finetuning.png'

执行代码块。

7.3.2.4 图像增强

这是一个非常重要的步骤。这里并不是人为的处理图片，而是在模型训练过程中程序自动对图片做适当的处理，如垂直翻转、旋转、缩放等操作。整个过程叫数据增强(data augmentation)。

【代码7-3-2-4】finetune\_flowers17.py

# 图像增强

aug = ImageDataGenerator(rotation\_range=30, width\_shift\_range=0.1,

height\_shift\_range=0.1, shear\_range=0.2, zoom\_range=0.2,

horizontal\_flip=True, fill\_mode="nearest")

执行代码块。

7.3.2.5 读取图像数据集

从指定路径读取图像数据集。并通过处理图片路径，获得该数据集中所有鲜花的类别。

【代码7-3-2-5】finetune\_flowers17.py

# 读取图像数据集

print("[INFO] 加载图片...")

imagePaths = list(paths.list\_images(dataset\_path))

classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]

classNames = [str(x) for x in np.unique(classNames)]

执行代码块。该代码块每执行一句都查看变量信息。

执行imagePaths = list(paths.list\_images(dataset\_path))，获得数据集中1360张图像的路径信息，如图19所示。

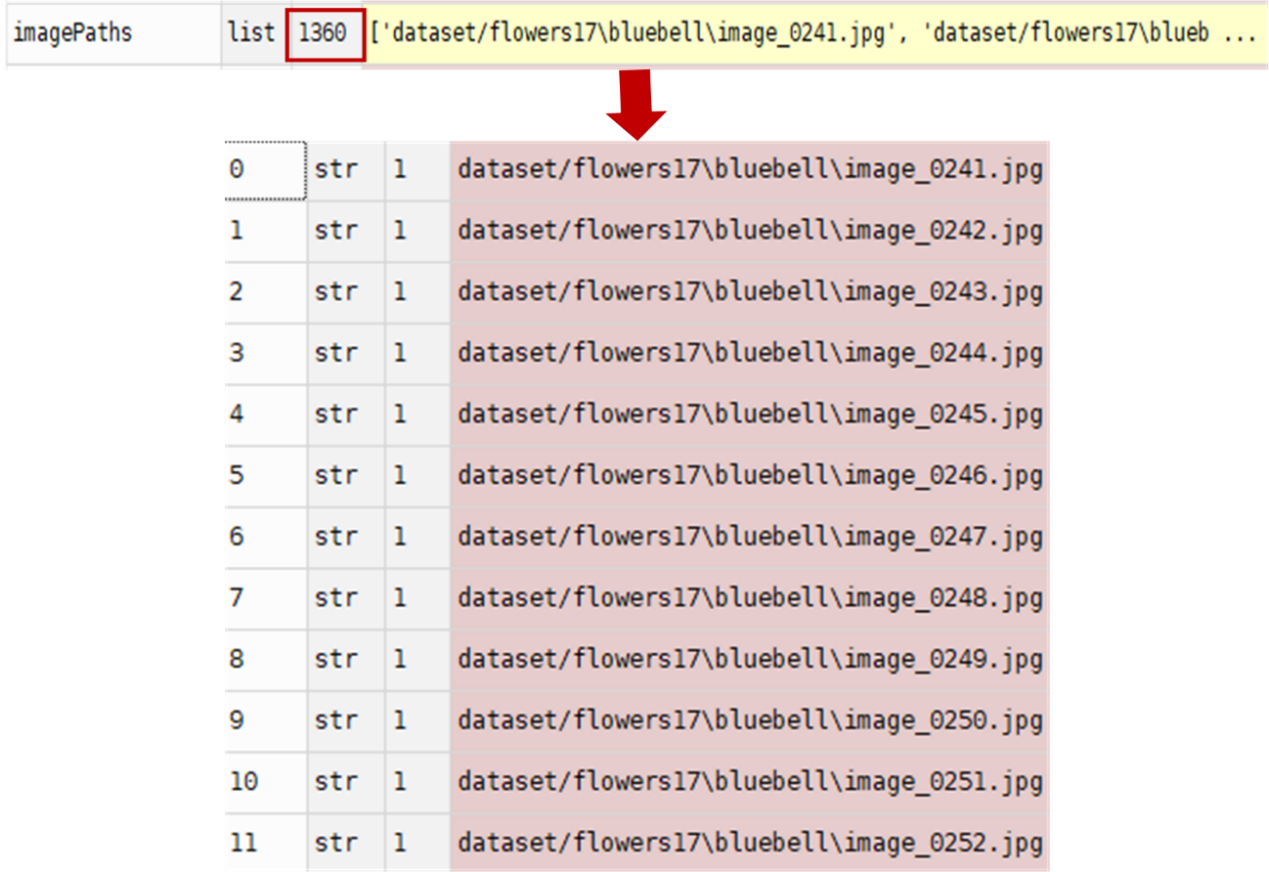


图19 1360张图片的路径列表

执行classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]，通过分割符划分字符串，并获得指定字符串，即可得到每张图片的类别信息。1360张图片的类别信息如图20所示。

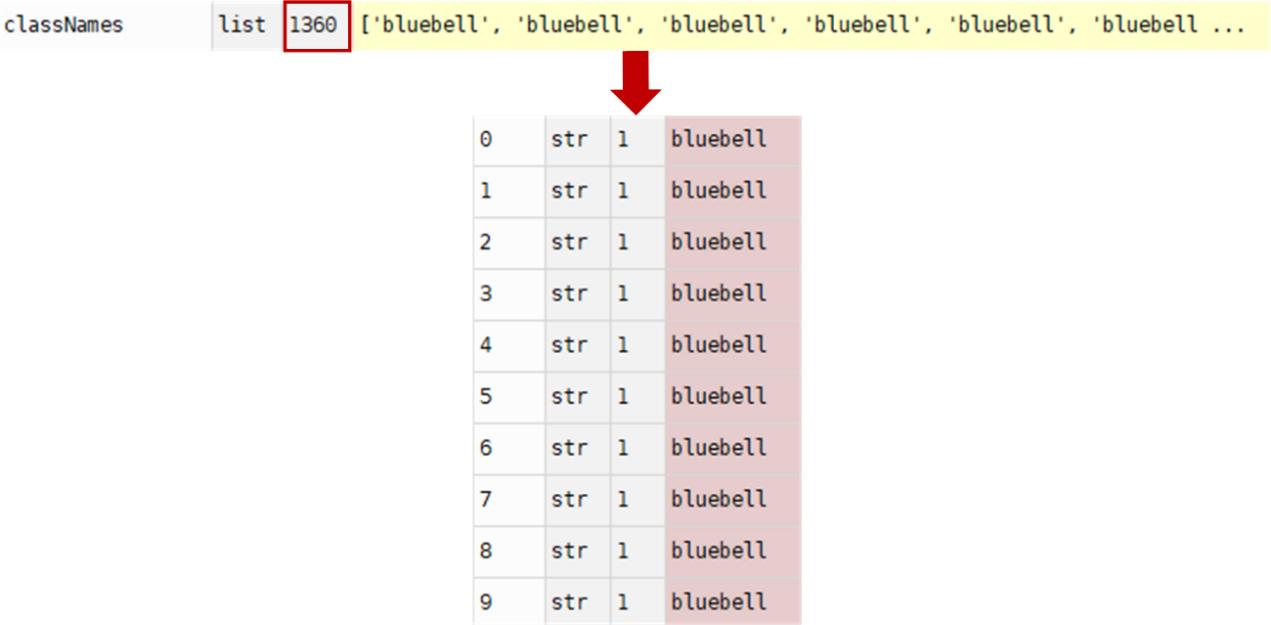


图20 1360张图片的类别

执行classNames = [str(x) for x in np.unique(classNames)]，通过np.unique函数去除上述获得的1360张图片类别中的重复元素，得到flower17数据集中鲜花的类别，共17个类别，如图21所示。

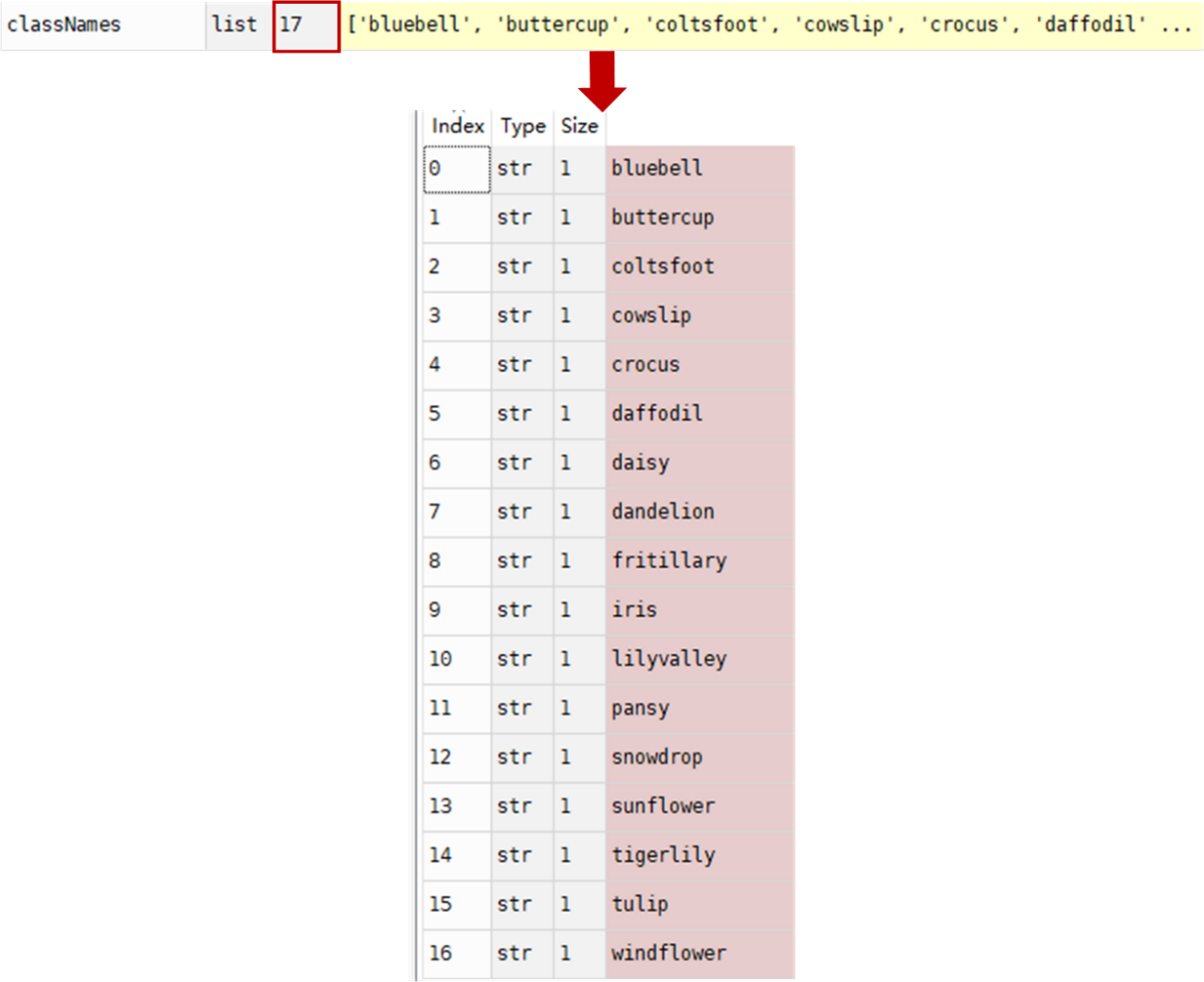


图21 flower17数据集中鲜花的类别

7.3.2.6 实例化图像处理类

由于VGG16的输入大小为224224，所以我们需要将所有图像处理为224224的大小，此处我们不能直接通过resize直接改变图像的大小，否则会造成图像变形，影响模型精度，我们通过AspectAwarePreprocessor这个辅助类来获得所需大小。

【代码7-3-2-6】finetune\_flowers17.py

# 实例化图像处理类

aap = AspectAwarePreprocessor(224, 224)

iap = ImageToArrayPreprocessor()

执行代码块。

7.3.2.7 处理图像数据

通过SimpleDatasetLoader辅助类，处理图像和标签（类别），作为特征向量，然后将图片数据标准化。

【代码7-3-2-7】finetune\_flowers17.py

# 处理图像数据

sdl = SimpleDatasetLoader(preprocessors=[aap, iap])

(data, labels) = sdl.load(imagePaths, verbose=500)

data = data.astype("float") / 255.0

执行代码块。

1360张图像的像素信息存放在data变量中，如图22所示。

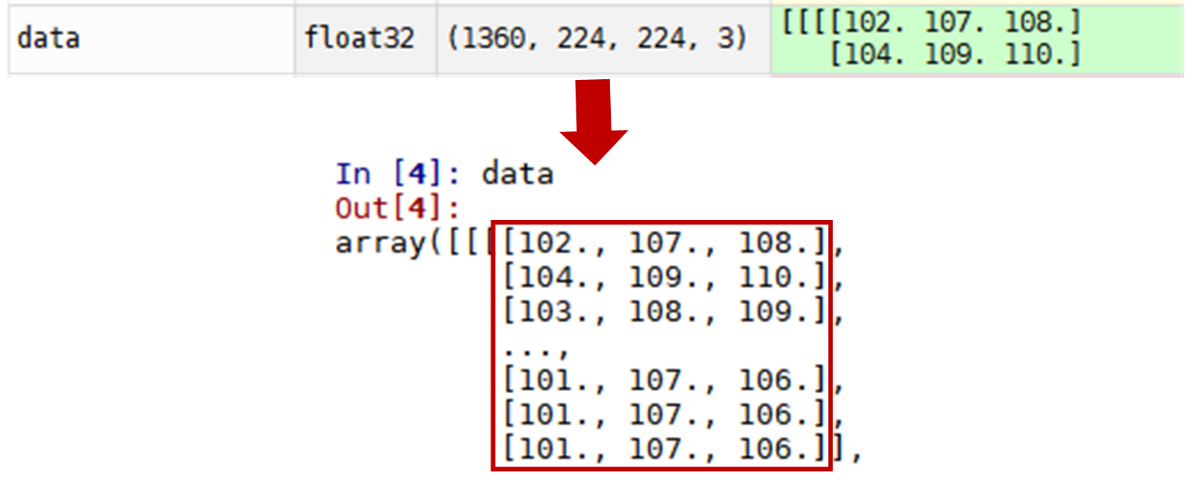


图22 1360张图像的像素信息存放在data变量中

标签信息存放在labels变量中，如图23所示。

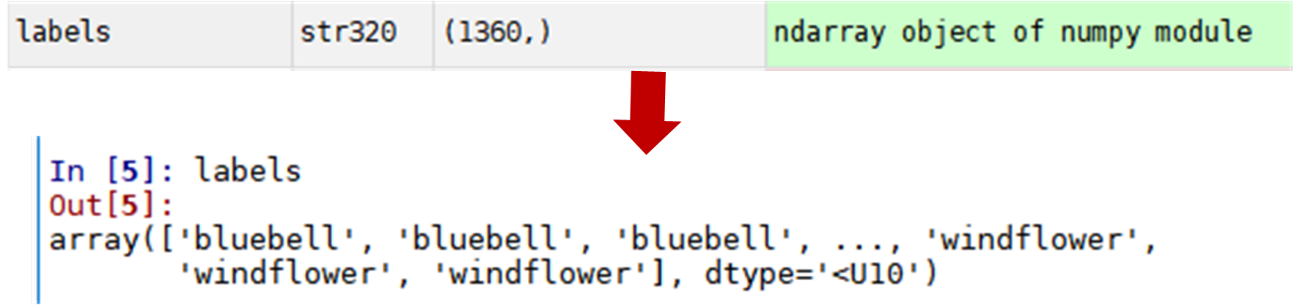


图23 1360张图像的标签信息存储在labels变量中

执行图像数据标准化后，data变量如图24所示，此时，图片像素值在[0,1]之间。

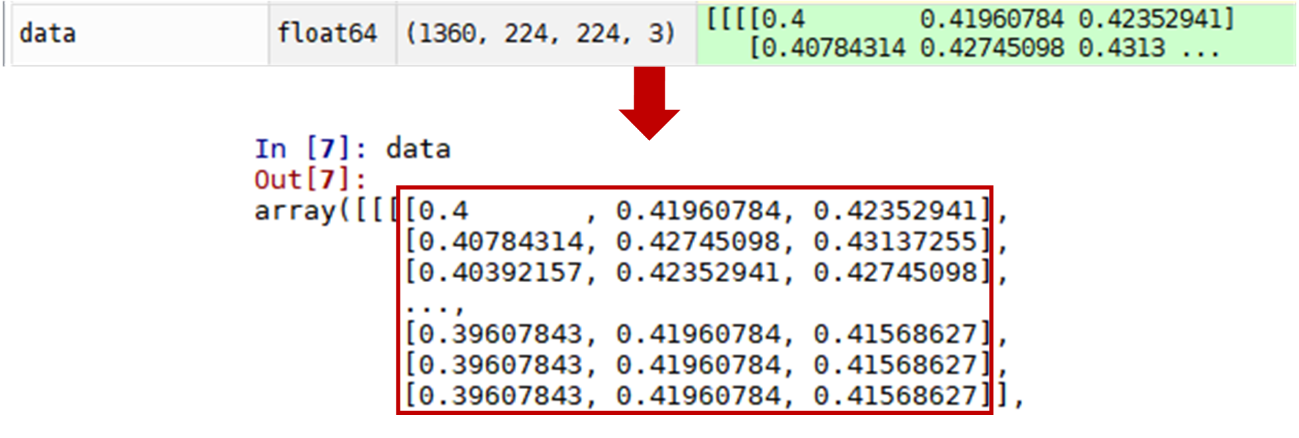


图24 图像标准化后data变量的信息

7.3.2.8 拆分数据集

将数据集拆分成训练集和测试集，拆分比例是3：1。

【代码7-3-2-8】finetune\_flowers17.py

# 拆分数据集

(trainX, testX, trainY, testY) = train\_test\_split(data, labels,test\_size=0.25, random\_state=42)

执行代码块。

testX表示340测试图片，testY表示340张测试图片的类别标签，trainX表示1020张训练图片，trainY表示1020张训练图片的类别标签，如图25所示。

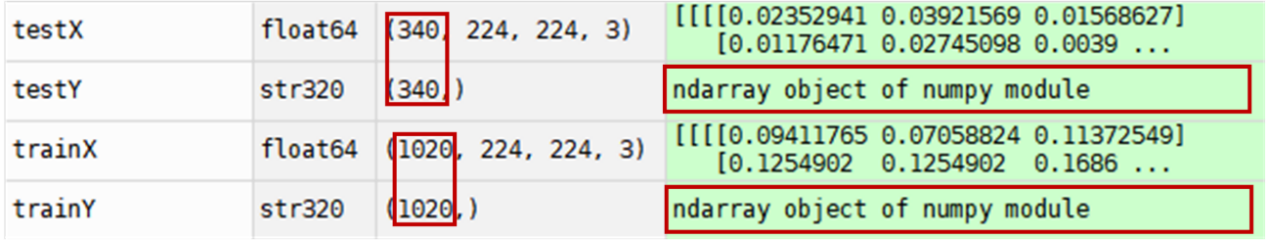


图25 训练集和测试集信息

7.3.2.9 对因变量做独热编码

因为CNN输出层的激活函数用的是softmax，且给出的分类标签当前以字符串表示，所以我们要对非数字化标签进行处理，对这些标签执行独热编码，使每个标签表示为矢量而非整数或是字符串。完成此编码，我们使用scikit-learn 中的LabelBinarizer，初始化LabelBinarizer 调用fit\_transform 在 trainY和testY中找到所有唯一的类标签并将他们转换为独热编码的标签。

【代码7-3-2-9】finetune\_flowers17.py

# 对因变量做独热编码

lb = LabelBinarizer()

trainY = lb.fit\_transform(trainY)

testY = lb.transform(testY)

执行代码块。

进行独热编码后，有几个类别就有几列，如图26所示。

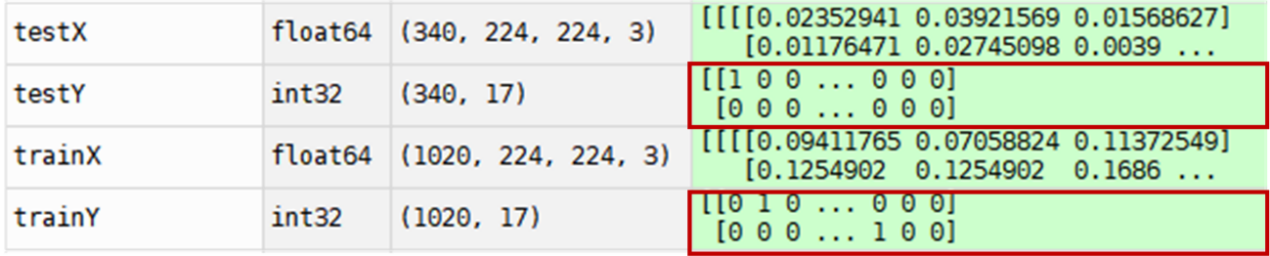


图26 独热编码后的标签

7.3.2.10实例化VGG16网络

本案例使用的是在Imagenet数据集上训练好的VGG16网络，且不包含3层全连接层，如果本地不存在该模型，那么会自动从网络上下载。

【代码7-3-2-10】finetune\_flowers17.py

# 实例化VGG16网络（确保不包含 head FC (Fully Connected) layer）

baseModel = VGG16(weights="imagenet", include\_top=False,input\_tensor=Input(shape=(224, 224, 3))) # 如果VGG16不存在，那么自动从网上下载

执行代码块。

7.3.2.11 实例化新的FC

由于我们使用迁移学习的模型，是不包含全连接层的VGG16网络模型，因此我们需要建立新的全连接层。

【代码7-3-2-11】finetune\_flowers17.py

# 实例化新的FC

headModel = FCHeadNet.build(baseModel, len(classNames), 256)

执行代码块。

7.3.2.12 将FC放到baseModel上

该部分是将新建立的全连接层放到网络主体上面，以组成完整的网络结构。

【代码7-3-2-12】finetune\_flowers17.py

# 把FC放到baseModel的上面

model = Model(inputs=baseModel.input, outputs=headModel)

执行代码块。

7.2.3.13 冻结baseModel的所有层

baseModel部分是已经在Imagenet数据集上训练好的，所以暂时不需要重新训练网络参数，将他们冻结。

【代码7-3-2-13】finetune\_flowers17.py

# 循环遍历baseModel的所有层，并冻结他们

for layer in baseModel.layers:

layer.trainable = False

执行代码块。

7.3.2.14 编译模型

编译模型，优化算法选择RMSprop，loss值选择二元交叉熵，评估指标选择accuracy。

优化算法选择RMSprop，一般需要手动指定learning rate的值，甚至需要不断调整学习率的值。

【代码7-3-2-14】finetune\_flowers17.py

# 编译模型

print("[INFO] 编译图片...")

opt = RMSprop(lr=LR)

model.compile(loss="categorical\_crossentropy", optimizer=opt, metrics=["accuracy"])

执行代码块。

7.3.2.15 实时画出训练头部的loss和accuracy

该部分的作用是实时画出loss和accuracy，每训练一个epoch，都实时更新loss和accuracy的曲线图。

【代码7-3-2-15】finetune\_flowers17.py

# callbacks，实时画出loss和accuracy

callbacks = [TrainingMonitor(warm\_up\_plot)]

执行代码块。

7.3.2.16 训练网络头部

该部分的目的是训练新建立的FC部分，baseModel部分已经冻结。根据代码中设置的超参数WARM\_UP\_EPOCHS，我们需要训练25个epoch。且训练过程中使用了验证集，因此可以实时评估模型性能。

模型训练时调用aug.flow()方法，即使用数据增强技术。所以虽然trainX有1020条数据，但是真正训练模型时的训练集数据远远大于1020条。

【代码7-3-2-16】finetune\_flowers17.py

# 训练网络的头部（其它部分被冻结了）

print("[INFO] 训练网络的头部...")

model.fit\_generator(aug.flow(trainX, trainY, batch\_size=BATCH\_SIZE),

validation\_data=(testX, testY), epochs=WARM\_UP\_EPOCHS,

steps\_per\_epoch=len(trainX) // BATCH\_SIZE, callbacks = callbacks, verbose=1)

执行代码块。

在Intel(R) Core(TM) i7-8750H [CPU@2.20GHz](mailto:CPU@2.20GHz)CPU，8GBRAM的计算机上，每个epoch的运行时间大约为1.9分钟，25个epoch大约需要47.5分钟。由于机器性能不同，运行时间有所不同，若训练速度过慢，建议同学们将WARM\_UP\_EPOCHS参数设置为5，仅用于查看案例效果即可。

模型训练过程如图27所示。

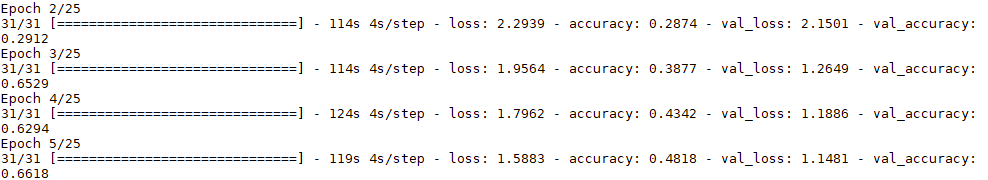


图27 模型训练过程

网络头部训练过程中，loss曲线图如图28所示，当随着epoch的递增，train\_loss和val\_loss曲线都不断下降，并趋于稳定，说明模型已经训练完成了，不再需要额外的epoch。



图28 网络头部训练loss图

网络头部训练过程中，accuracy曲线图如图29所示，随着epoch的递增，train\_acc和val\_acc的曲线都不断上升，而且在验证集中，模型达到了85.29%的准确率，说明模型性能基本上不错了。

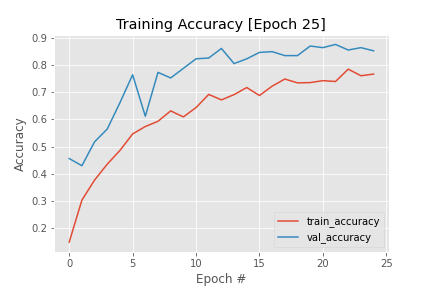


图29 网络头部训练accuracy图

7.3.2.17 评估初始网络性能

头部训练完成后，我们在测试集上评估模型性能。

【代码7-3-2-17】finetune\_flowers17.py

# 网络头部训练完成后，评估模型性能

print("[INFO] evaluating after initialization...")

predictions = model.predict(testX, batch\_size=BATCH\_SIZE)

print(classification\_report(testY.argmax(axis=1),

predictions.argmax(axis=1), target\_names=classNames))

执行代码块。

性能评估结果，如图30所示：

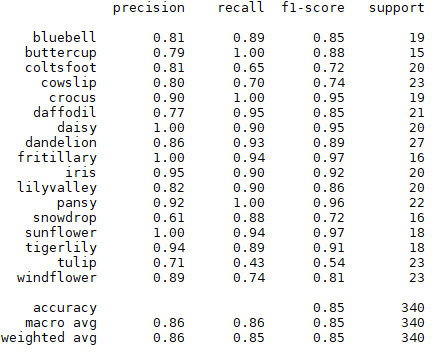


图30 测试评估结果

从图30结果可以看出，在测试集的340张图像上，每个类别precision、recall、f1-sorce的值大部分都在0.80以上，accuracy、macro avg（加权平均）、weighted avg（加权平均）都在0.85左右，还未达到0.90，说明模型还有进一步提升的空间。

注意：由于新建立的FC层在模型开始训练时，权重会随机初始化，所以同学们的训练结果不会和本案例一模一样。

7.3.2.18 解冻baseModel的层

由于在卷积神经网络中，浅层网络（前几层）捕获了图像的一般细节，例如边缘信息、颜色信息等，且在该分类任务中，前4个‘卷积池化块‘提取的特征已经足够，深层网络会捕获图像的更深层次的语义信息，因此，我们解冻baseModel的最后1个‘卷积池化层块’（最后三层卷积层和最后的池化层），在接下来的训练中微调其参数。

【代码7-3-2-18】finetune\_flowers17.py

# 网络头部训练好后，把baseModel的最后1层‘卷积池化层块’解冻

for layer in baseModel.layers[15:]:

layer.trainable = True

执行代码块。

7.3.2.19 重新编译模型

为了微调最后一个卷积块和网络头部的参数，需要重新编译模型，优化算法选择SGD，loss值选择二元交叉熵，评估指标选择accuracy。

优化算法选择SGD，一般需要手动指定learning rate的值，甚至需要不断调整学习率的值。

【代码7-3-2-19】finetune\_flowers17.py

# 重新编译模型

print("[INFO] 重新编译模型...")

opt = SGD(lr=LR)

model.compile(loss="categorical\_crossentropy", optimizer=opt, metrics=["accuracy"])

执行代码块。

7.3.2.20 实时画出完整模型的loss和accuracy

该部分的作用是实时画出最终模型的loss和accuracy，每训练一个epoch，都实时更新loss和accuracy的曲线图。

【代码7-3-2-20】finetune\_flowers17.py

# callbacks，实时画出loss和accuracy

callbacks = [TrainingMonitor(fine\_tuning\_plot)]

执行代码块。

7.3.2.21 重新训练模型

该部分的目的是重新训练模型，微调最后一个卷积块和网络头部的参数。根据代码中设置的超参数FINE\_TUNING\_EPOCHS，我们需要训练100个epoch。且训练过程中使用了验证集，因此可以实时评估模型性能。

模型训练时调用aug.flow()方法，即使用数据增强技术。所以虽然trainX有1020条数据，但是真正训练模型时的训练集数据远远大于1020条。

【代码7-3-2-21】finetune\_flowers17.py

# 再次训练模型。微调最后一个卷积块和网络头部的参数

print("[INFO] 再次训练模型...")

model.fit\_generator(aug.flow(trainX, trainY, batch\_size=BATCH\_SIZE),

validation\_data=(testX, testY), epochs=FINE\_TUNING\_EPOCHS,

steps\_per\_epoch=len(trainX) // BATCH\_SIZE, callbacks = callbacks, verbose=1)

执行代码块。

在Intel(R) Core(TM) i7-8750H [CPU@2.20GHz](mailto:CPU@2.20GHz)CPU，8GBRAM的计算机上，每个epoch的运行时间大约为2.4分钟，100个epoch大约需要4小时。由于机器性能不同，运行时间有所不同，若训练速度过慢，建议同学们将FINE\_TUNING\_EPOCHS参数设置为5，仅用于查看案例效果即可。

完整模型训练过程如图31所示：

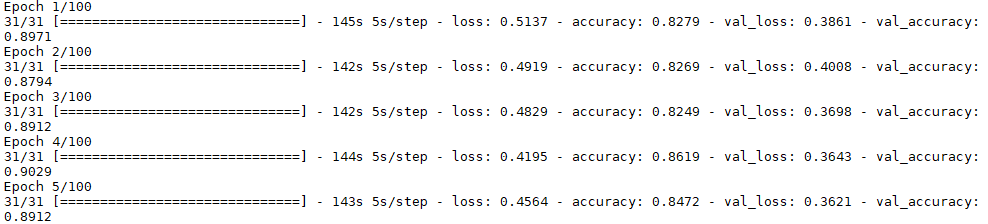


图31 完整模型训练过程图

整个模型训练过程中，loss曲线图如图32所示，当随着epoch的递增，train\_loss和val\_loss曲线都不断下降，并趋于稳定，说明模型已经训练完成了，不再需要额外的epoch。

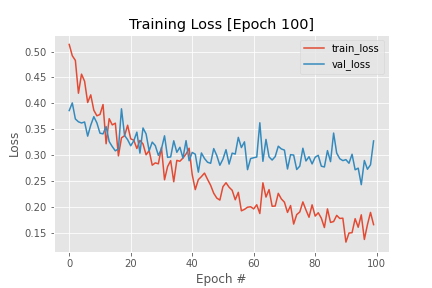


图32 完整模型训练loss图

整个模型训练过程中，accuracy曲线图如图33所示，随着epoch的递增，train\_acc和val\_acc的曲线都不断上升，且基本上重合，说明训练没有欠拟合，而且在验证集中，模型达到了91.76%的准确率，说明模型性能已经很好了。

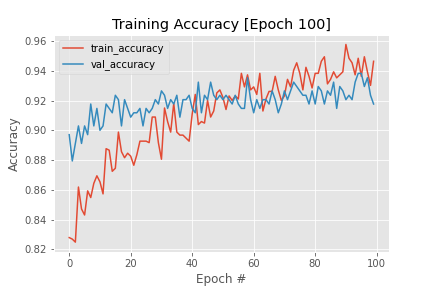


图33 完整模型训练accuracy图

7.3.2.22 评估模型分类性能

整个模型训练完成后，我们在测试集上评估模型性能。

【代码7-3-2-22】finetune\_flowers17.py

# 评估模型分类性能

print("[INFO] 评估模型分类性能...")

predictions = model.predict(testX, batch\_size=BATCH\_SIZE)

print(classification\_report(testY.argmax(axis=1),

predictions.argmax(axis=1), target\_names=classNames))

执行代码块。

最终模型性能评估结果如图34所示：

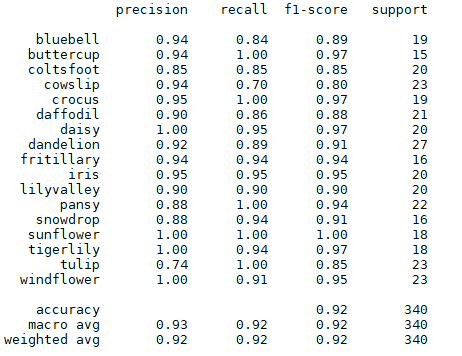


图34 完整模型性能评估结果

从图34结果可以看出，在测试集的340张图像上，每个类别precision、recall、f1-sorce的值大部分都在0.80以上，accuracy、macro avg（加权平均）、weighted avg（加权平均）都在0.92左右，证明模型分类的准确度比较高。

7.4对比两种迁移学习方法得到的分类模型

该模块的作用是从速度以及精度两个角度对比两种迁移学习方法得到的分类模型。

7.4.1 速度比较

在Intel(R) Core(TM) i7-8750H [CPU@2.20GHz](mailto:CPU@2.20GHz)CPU，8GBRAM的计算机上，使用预训练模型作为特征提取器的迁移学习方式，由7.2.2节可知，提取鲜花图像特征用时约两分钟，由7.2.4节可知，基于提取的特征建立逻辑回归模型进行鲜花分类用时约1分钟，使用此种迁移学习方法在flowers17数据集上用时共3分钟左右。

在同样的实验环境下，使用微调的迁移学习方式，由7.3.2.16节可知，训练网络头部用时约47.5分钟，由7.3.2.21节可知，微调整个网络参数，训练过程约18分钟，总共用时约23分钟。

在速度上，使用预训练模型作为特征提取器的迁移学习方式要比使用微调的迁移学习方式快。

7.4.2 模型分类性能比较

在340张图片上评估模型性能，两个模型的性能评价指标如图35所示。

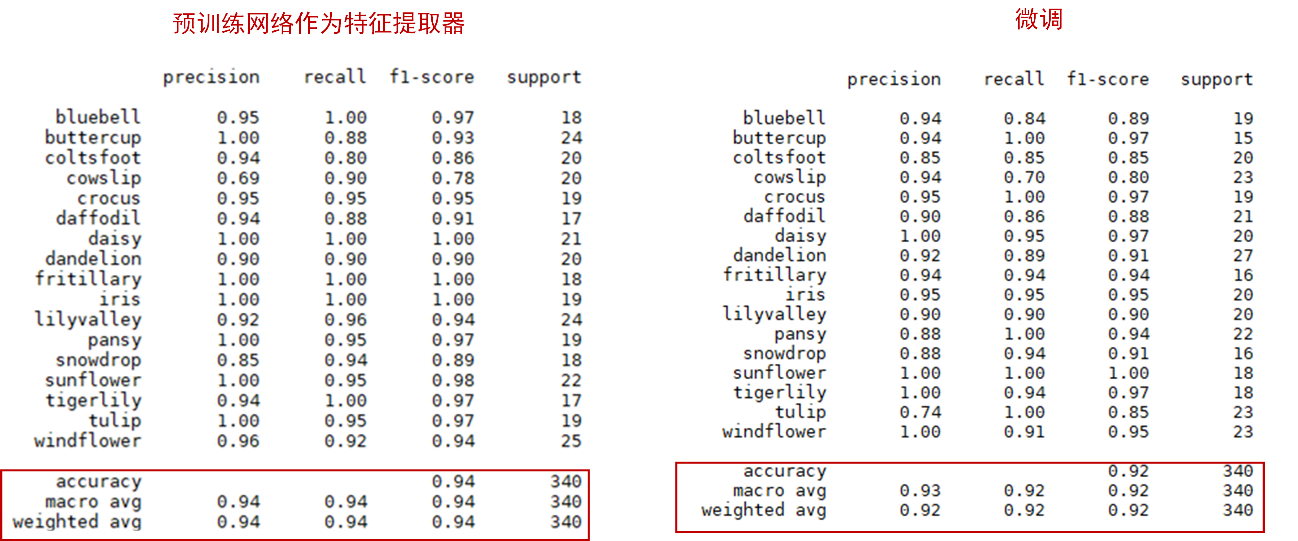


图35 使用两种迁移学习方式分类模型的性能比较

比较图35中使用两种迁移学习方式的模型评估结果，使用预训练网络作为特征提取器的迁移学习方式在340张测试图像上的accuracy、macro avg、weighted avg高于使用微调的迁移学习方式。

综合7.4.1节和7.4.2节，使用预训练网络作为特征提取器的迁移学习方式在flowers17数据集上的实现速度要远远快于使用微调的迁移学习方式，在模型分类性能上也优于使用微调的迁移学习方式。但是，这并不能证明使用微调的迁移学习方式不如使用预训练网络作为特征提取器的迁移学习方式，若在规模较大、特征较为复杂的数据集上，或者进行其他任务，如目标检测、图像语义分割等，使用微调的迁移学习方式能够大大缩短训练时间。

8.案例代码

【案例代码】hdf5datasetwriter.py

# import the necessary packages

import h5py

import os

class HDF5DatasetWriter:

def \_\_init\_\_(self, dims, outputPath, dataKey="images",bufSize=1000):

# check to see if the output path exists, and if so, raise

# an exception

if os.path.exists(outputPath):

raise ValueError("The supplied `outputPath` already "

"exists and cannot be overwritten. Manually delete "

"the file before continuing.", outputPath)

# open the HDF5 database for writing and create two datasets:

# one to store the images/features and another to store the

# class labels

self.db = h5py.File(outputPath, "w")

self.data = self.db.create\_dataset(dataKey, dims,dtype="float")

self.labels = self.db.create\_dataset("labels", (dims[0],),dtype="int")

# store the buffer size, then initialize the buffer itself

# along with the index into the datasets

self.bufSize = bufSize

self.buffer = {"data": [], "labels": []}

self.idx = 0

def add(self, rows, labels):

# add the rows and labels to the buffer

self.buffer["data"].extend(rows)

self.buffer["labels"].extend(labels)

# check to see if the buffer needs to be flushed to disk

if len(self.buffer["data"]) >= self.bufSize:

self.flush()

def flush(self):

# write the buffers to disk then reset the buffer

i = self.idx + len(self.buffer["data"])

self.data[self.idx:i] = self.buffer["data"]

self.labels[self.idx:i] = self.buffer["labels"]

self.idx = i

self.buffer = {"data": [], "labels": []}

def storeClassLabels(self, classLabels):

# create a dataset to store the actual class label names,

# then store the class labels

dt = h5py.special\_dtype(vlen=str) # `vlen=unicode` for Py2.7

labelSet = self.db.create\_dataset("label\_names",(len(classLabels),), dtype=dt)

labelSet[:] = classLabels

def close(self):

# check to see if there are any other entries in the buffer

# that need to be flushed to disk

if len(self.buffer["data"]) > 0:

self.flush()

# close the dataset

self.db.close()

**【案例代码】extract\_features.py**

# -\*- coding: utf-8 -\*-

'''

从鲜花数据集提取图像特征

'''

# 导入库

from keras.applications import VGG16

from keras.applications import imagenet\_utils

from keras.preprocessing.image import img\_to\_array

from keras.preprocessing.image import load\_img

from sklearn.preprocessing import LabelEncoder

from hdf5datasetwriter import HDF5DatasetWriter

from imutils import paths

import numpy as np

import progressbar

import random

import os

# 全局常量

BATCH\_SIZE = 32

BUFFER\_SIZE = 1000

# 全局变量

dataset\_path = 'dataset/flowers17'

output\_path = 'output/flowers17\_features.hdf5'

# 获取图像名字，并打乱顺序

print("[INFO] 导入图像...")

imagePaths = list(paths.list\_images(dataset\_path))

random.shuffle(imagePaths)

# 提取图像的标签

labels = [p.split(os.path.sep)[-2] for p in imagePaths]

le = LabelEncoder()

labels = le.fit\_transform(labels)

# 加载 VGG16 网络

print("[INFO] 加载 VGG16 网络...")

model = VGG16(weights="imagenet", include\_top=False) # VGG16是去掉头部的（去掉FC层）

# 初始化HDF5类

dataset = HDF5DatasetWriter((len(imagePaths), 512 \* 7 \* 7),output\_path, dataKey="features", bufSize=BUFFER\_SIZE)

# 写入标签

dataset.storeClassLabels(le.classes\_)

# 初始化进度条

widgets = ["Extracting Features: ", progressbar.Percentage(), " ",progressbar.Bar(), " ", progressbar.ETA()]

pbar = progressbar.ProgressBar(maxval=len(imagePaths),widgets=widgets).start()

# 循环遍历每一批图片

for i in np.arange(0, len(imagePaths), BATCH\_SIZE):

batchPaths = imagePaths[i:i + BATCH\_SIZE]

batchLabels = labels[i:i + BATCH\_SIZE]

batchImages = []

# 循环遍历这批图片中的每一张图片

for (j, imagePath) in enumerate(batchPaths):

# 确保输入尺寸是224x224，这是VGG16的输入图像尺寸

image = load\_img(imagePath, target\_size=(224, 224))

image = img\_to\_array(image)

# 使用图像预处理函数preprocess\_input()，执行mean subtraction运算

image = np.expand\_dims(image, axis=0)

image = imagenet\_utils.preprocess\_input(image)

# 把单张图片加入到这一批中

batchImages.append(image)

# 把这批图片传入到网络中

batchImages = np.vstack(batchImages)

features = model.predict(batchImages, batch\_size=BATCH\_SIZE)

# 存储平铺的特征

features = features.reshape((features.shape[0], 512 \* 7 \* 7))

# 把特征和标签加入到 HDF5 中

dataset.add(features, batchLabels)

pbar.update(i)

# 关闭数据集

dataset.close()

# 停止进度条

pbar.finish()

**【案例代码】check\_hdf5.py**

import h5py

p = 'output/flowers17\_features.hdf5'

db = h5py.File(p)

print(list(db.keys())) # ['features', 'label\_names', 'labels']

print(db["features"].shape) # (1360, 25088) (25088=512 \* 7 \* 7)

print(db["labels"].shape) # (1360,)

print(db["label\_names"].shape) # (17,)

**【案例代码】train\_model.py**

# -\*- coding: utf-8 -\*-

'''

基于提取的特征建立逻辑回归模型做分类

f1-socre: 91%

'''

# 导入包

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

import h5py

import time

import math

# 全局变量

feature\_path = 'output/flowers17\_features.hdf5'

# 75%数据做训练集，25%数据做测试集

db = h5py.File(feature\_path, "r")

i = int(db["labels"].shape[0] \* 0.75)

# 训练模型

print("[INFO] 训练模型...")

start\_time = time.time()

model = LogisticRegression()

model.fit(db["features"][:i], db["labels"][:i])

end\_time = time.time()

print('Used %d minutes' % math.ceil((end\_time - start\_time)/60))

# 评估模型

print("[INFO] evaluating...")

preds = model.predict(db["features"][i:])

print(classification\_report(db["labels"][i:], preds,target\_names=db["label\_names"]))

# 关闭数据集

db.close()

【案例代码】inspect\_model.py

# -\*- coding: utf-8 -\*-

'''

查看VGG16结构

'''

# 导入库

from keras.applications import VGG16

INCLUDE\_TOP = False # if True, FC layers are included; if False, FC layers are excluded.

# 加载VGG16网络（如果VGG16不存在，那么自动从网上下载）（16的含义是：13个卷积层+3个全连接层）

print("[INFO] 加载VGG16网络...")

model = VGG16(weights="imagenet", include\_top = INCLUDE\_TOP)

print("[INFO] 显示VGG16的层...")

print("[INFO] 显示输入层：")

print(model.input) # 输入层

# 遍历显示VGG16的层

print("[INFO] 显示隐藏层：")

for (i, layer) in enumerate(model.layers):

print("[INFO] {}\t{}".format(i, layer)) # 隐藏层

print("[INFO] 显示输出层：")

print(model.output) # 输出层

【案例代码】finetune\_flowers17.py

# -\*- coding: utf-8 -\*-

'''

使用迁移学习训练鲜花分类模型

'''

# 导入库

from sklearn.preprocessing import LabelBinarizer

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report

from computervision.callbacks import TrainingMonitor

from computervision.preprocessing import ImageToArrayPreprocessor

from computervision.preprocessing import AspectAwarePreprocessor

from computervision.datasets import SimpleDatasetLoader

from computervision.nn.conv import FCHeadNet

from keras.preprocessing.image import ImageDataGenerator

from keras.optimizers import RMSprop

from keras.optimizers import SGD

from keras.applications import VGG16

from keras.layers import Input

from keras.models import Model

from imutils import paths

import numpy as np

import os

# 模型超参数

WARM\_UP\_EPOCHS = 25

FINE\_TUNING\_EPOCHS = 100

BATCH\_SIZE = 32

LR = 0.001

# 全局变量

dataset\_path = 'dataset/flowers17'

warm\_up\_plot = 'output/flowers17\_warmup.png'

fine\_tuning\_plot = 'output/flowers17\_finetuning.png'

# 图像增强

aug = ImageDataGenerator(rotation\_range=30, width\_shift\_range=0.1,

height\_shift\_range=0.1, shear\_range=0.2, zoom\_range=0.2,

horizontal\_flip=True, fill\_mode="nearest")

# 读取图像数据集

print("[INFO] 加载图片...")

imagePaths = list(paths.list\_images(dataset\_path))

classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]

classNames = [str(x) for x in np.unique(classNames)]

# 实例化图像处理类

aap = AspectAwarePreprocessor(224, 224)

iap = ImageToArrayPreprocessor()

# 处理图像数据

sdl = SimpleDatasetLoader(preprocessors=[aap, iap])

(data, labels) = sdl.load(imagePaths, verbose=500)

data = data.astype("float") / 255.0

# 拆分数据集

(trainX, testX, trainY, testY) = train\_test\_split(data, labels,test\_size=0.25, random\_state=42)

# 对因变量做独热编码

lb = LabelBinarizer()

trainY = lb.fit\_transform(trainY)

testY = lb.transform(testY)

# 实例化VGG16网络（确保不包含 head FC (Fully Connected) layer）

baseModel = VGG16(weights="imagenet", include\_top=False,input\_tensor=Input(shape=(224, 224, 3))) # 如果VGG16不存在，那么自动从网上下载

# 实例化新的FC

headModel = FCHeadNet.build(baseModel, len(classNames), 256)

# 把FC放到baseModel的上面

model = Model(inputs=baseModel.input, outputs=headModel)

# 循环遍历baseModel的所有层，并冻结他们

for layer in baseModel.layers:

layer.trainable = False

# 编译模型

print("[INFO] 编译图片...")

opt = RMSprop(lr=LR)

model.compile(loss="categorical\_crossentropy", optimizer=opt, metrics=["accuracy"])

# callbacks，实时画出loss和accuracy

callbacks = [TrainingMonitor(warm\_up\_plot)]

# 训练网络的头部（其它部分被冻结了）

print("[INFO] 训练网络的头部...")

model.fit\_generator(aug.flow(trainX, trainY, batch\_size=BATCH\_SIZE),

validation\_data=(testX, testY), epochs=WARM\_UP\_EPOCHS,

steps\_per\_epoch=len(trainX) // BATCH\_SIZE, callbacks = callbacks, verbose=1)

# 网络头部训练完成后，评估模型性能

print("[INFO] evaluating after initialization...")

predictions = model.predict(testX, batch\_size=BATCH\_SIZE)

print(classification\_report(testY.argmax(axis=1),

predictions.argmax(axis=1), target\_names=classNames))

# 网络头部训练好后，把baseModel的最后1层‘卷积池化层块’解冻

for layer in baseModel.layers[15:]:

layer.trainable = True

# 重新编译模型

print("[INFO] 重新编译模型...")

opt = SGD(lr=LR)

model.compile(loss="categorical\_crossentropy", optimizer=opt, metrics=["accuracy"])

# callbacks，实时画出loss和accuracy

callbacks = [TrainingMonitor(fine\_tuning\_plot)]

# 再次训练模型。微调最后一个卷积块和网络头部的参数

print("[INFO] 再次训练模型...")

model.fit\_generator(aug.flow(trainX, trainY, batch\_size=BATCH\_SIZE),

validation\_data=(testX, testY), epochs=FINE\_TUNING\_EPOCHS,

steps\_per\_epoch=len(trainX) // BATCH\_SIZE, callbacks = callbacks, verbose=1)

# 评估模型分类性能

print("[INFO] 评估模型分类性能...")

predictions = model.predict(testX, batch\_size=BATCH\_SIZE)

print(classification\_report(testY.argmax(axis=1),

predictions.argmax(axis=1), target\_names=classNames))