

【深度学习】花果识别算法及部署（微信小程序&树莓派）

一、数据处理

- 1.1 数据集简介
- 1.2 数据收集（来源）
 - 1.2.1 现有数据集
 - 1.2.2 数据爬取
- 1.3 数据预处理
 - 1.3.1 数据划分
 - 1.3.2 图像格式的统一
 - 1.3.3 图像的解码、尺寸调整和归一化
- 1.4 MongoDB 存储
 - 1.4.1 数据库结构
 - 1.4.2 GridFS 储存

二、算法介绍

- 2.1 模型搭建
 - 2.1.1 实验1：仿照 Alex Net 网络结构
 - 代码
 - 评估
 - 2.1.2 实验2：
 - 模型介绍
 - 训练过程：
 - 模型评估
 - 2.1.3 实验3 含残差块的网络结构
 - 2.1.4 实验4：迁移学习（Mobile-Net V2）
 - 1. 迁移学习原理
 - 2. Mobile Net 介绍

- 2.2 模型选择
 - 2.2.1 因素比较
- 2.3 服务器部署
 - 2.3.1 配置服务器
 - 所需软件/环境
 - 服务器代码
 - 小程序部署

三、模型部署

- 3.1 TFLite 介绍
- 3.2 部署过程
 - 3.2.1 初始版本
 - 识别源码
 - 版本分析
 - 不足：
 - 3.2.2 部署改进
 - 改进思路
 - 改进源码
 - 改进评估
 - 3.2.3 目标检测

【深度学习】花果识别算法及部署（微信小程序 序&树莓派）

一、数据处理

- 此部分主要对本项目采用的数据集进行介绍，具体阐述了在训练模型前对数据集进行的所有预处理过程，并附上相关代码。

1.1 数据集简介

- 本项目使用的数据集包括花卉、水果共26种，其中水果13种，花卉13种。具体种类名称、对应图片数量以及分类（花卉记作“0”，水果记作“1”）见下方列表：

序号	名称	图片数量	备注	序号	名称	图片数量	备注
1	香蕉 (banana)	1127	1	14	千日红 (globe amaranth)	751	0
2	秋海棠 (begonia)	681	0	15	葡萄 (grape)	1348	1
3	蓝莓 (blueberry)	1040	1	16	柠檬 (lemon)	1600	1
4	樱桃 (cherry)	1071	1	17	荔枝 (litchi)	891	1
5	菊花 (chrysanthemum)	1050	0	18	芒果 (mango)	1422	1
6	仙客来 (cyclamen)	688	0	19	雾华泽兰 (mistflower)	900	0
7	大丽花 (dahlia)	1010	0	20	黑种草 (nigella)	954	0
8	雏菊 (daisy)	1952	0	21	橙子 (orange)	1672	1
9	飞燕草 (delphinium)	296	0	22	菠萝 (pineapple)	1390	1
10	榴莲 (durian)	1090	1	23	石榴 (pomegranate)	1249	1
11	无花果 (fig)	1049	1	24	草莓 (strawberry)	1183	1
12	紫茉莉 (four-o'clock)	520	0	25	向日葵 (sunflower)	1836	0
13	龙胆草 (gentian)	1295	0	26	桂竹香 (wallflower)	256	0

1.2 数据收集（来源）

本部分介绍本项目使用的**数据来源、收集方法和整理规范**。

1.2.1 现有数据集

- 本项目数据集整合了多个现有的花卉水果数据集，从中挑选出我们预先设定的种类的图片。在整合现有数据集时，我们侧重于**挑选图片尺寸大小适中、图片质量（像素）高、图片数量充足的数据集**，包括了：
 1. Oxford Flowers TFRecords (源于kaggle，链接：<https://www.kaggle.com/cdeotte/oxford-flowers-tfrecords>)
 2. Flower_Recognition_HE (源于Kaggle，链接：<https://www.kaggle.com/rednivrug/flower-recognition-he>)
 3. 102 Category Flower Dataset (链接：<https://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html>)
 4. Fruit Recognition (源于Kaggle，链接：<https://www.kaggle.com/chrisfilo/fruit-recognition>)
 5. Fruit and Vegetable Image Recognition (源于kaggle，链接：<https://www.kaggle.com/kritikseth/fruit-and-vegetable-image-recognition>)
 6. fruits_vegetables_photos (源于Kaggle，链接：<https://www.kaggle.com/balalexy/fruits-vegetables-photos>)
 7. 参考了一篇Github博客 (<https://github.com/quarrying/awesome-plant-recognition>)
 8. 获取了ImageNet上已有的数据集 (<http://www.image-net.org/>)
- 以上数据集的质量和数量都比较可观，我们从中挑选出需要的图片并进行整合。

1.2.2 数据爬取

- **爬取原因：**由于以上现有数据集中大多是经过处理的高质量图片，与实际生活中常见的场景有所差距。本项目的目的是希望应用于日常生活中的**普遍场景，能够识别噪声较大的花果图片**，因此为了**进一步丰富图片数量、增强训练模型的鲁棒性**，我们选择进一步从网络上进行数据爬取，获取一些未经专业处理的花果图片。
- **爬取来源：** Google Image
- **爬虫代码：**代码包括了**数据爬取、图片重命名和图片存储**。利用此代码从Google中分别给以每一类花果爬取最大为1000张的网络图片。

```
from selenium import webdriver
import time
import urllib.request
import os
from selenium.webdriver.common.keys import Keys

#需要安装Chrome浏览器
# 需要下载Chromedriver 下载地址:
http://chromedriver.storage.googleapis.com/index.html
# Chromedriver的版本号要与Chrome要对应

Chromedriver_path="D:/ChromeDriver/chromedriver_win32/chromedriver"# 下载
Chromedriver后的地址

name="barberton daisy" # 设置爬取标签
kinds=0 #1代表水果 0代表花卉
```

```
num=1000 #最大图片张数
pages=25# 网页翻页次数，就相当于下拉
def get_img_from_google(chromedriver_path,name,num,pages,kinds):
    chrome_options = webdriver.ChromeOptions()

    #chrome_options.add_argument('--headless')
    chrome_options.add_argument('--disable-gpu')

    #chrome_options.add_argument('--no-sandbox') # 以根用户打身份运行Chrome，使用-no-sandbox标记重新运行Chrome，禁止沙箱启动
    if kinds==1:
        key_words = name + " " + "fruit"
    elif kinds==0:
        key_words=name+" "+"flower"

    browser=webdriver.Chrome(options=chrome_options,executable_path=chromedriver_path)

    browser.get("https://www.google.com")
    search = browser.find_element_by_name("q")
    search.send_keys(key_words,Keys.ENTER)
    elem = browser.find_element_by_link_text('图片')
    elem.get_attribute('href')
    elem.click()
    value = 0
    for i in range(pages):
        browser.execute_script('scrollBy('+ str(value) +',+1000);')
        value += 1000
        time.sleep(3)

    elem1 = browser.find_element_by_id('isImp')
    sub = elem1.find_elements_by_tag_name('img')

    count = 0

    for i in sub:
        if count > num:
            break

        src = i.get_attribute('src')
        try:
            if src != None:
                src = str(src)
                print(src)
                count+=1
                urlib.request.urlretrieve(src,
os.path.join('E:/Dataset/barbeton daisy',name+str(count)+'.jpg')) #图片命名
                time.sleep(0.5)
        else:
            raise TypeError
        except TypeError:
            print('fail')

    time.sleep(3)
```

```

        browser.quit()
        print("爬取完成, 共获得"+str(count)+"张图片")
        return

get_img_from_google(chromedriver_path, name, num, pages, kinds)

```

- 图片筛选**: 由于搜索关键词后获取的网络图片纷杂, 可能与目标图片存在较大的差别, 因此需要进一步人工对爬取图片进行筛选, 去除差异较大的“脏数据”。
- 数据整合和规范命名**: 最后将筛选后的数据与收集到的已有数据集进行整合, 并进一步规范命名。重命名代码如下:

```

import os
path = 'E:/Dataset/barbeton daisy'    #文件夹位置
files = os.listdir(path)
for i, file in enumerate(files):
    NewName = os.path.join(path, "barbeton daisy"+str(i+1)+'.jpg')    #新名字:种类+序号
    oldName = os.path.join(path, file)
    os.rename(oldName, NewName)

```

1.3 数据预处理

1.3.1. 数据划分

- 为了方便后续模型的训练和测试, 我们首先将数据集以8: 1: 1的比例分别划分为训练集、验证集和测试集, 具体代码如下:

```

import numpy as np
import os
import shutil
# 图片文件夹的地址, 请务必复制一份到自己的文件夹下进行操作
path="/home/group6/deep_learning/experiment/zcl/images"
files = os.listdir(path)

val_path="/home/group6/deep_learning/experiment/zcl/val/" #验证集路径
test_path="/home/group6/deep_learning/experiment/zcl/test/" # 测试集路径
os.mkdir(val_path)
os.mkdir(test_path)

def split_val_test(file_name, val_rate, test_rate):
    #随机数可以改一下 不同随机数就是不同切分方法
    np.random.seed(10)
    path = "/home/group6/deep_learning/experiment/zcl/images" #与上面path相同
    img_path = os.path.join(path, file_name)
    img_files = os.listdir(img_path)

    np.random.shuffle(img_files)
    num = len(img_files)
    val_num = int(num * val_rate)

```

```

test_num = int(num * test_rate)
val_file = img_files[:val_num]
test_file = img_files[val_num:val_num + test_num]
#### split val_file 分割验证集
os.mkdir(os.path.join(val_path, file_name))
for img in val_file:
    val_img = os.path.join(img_path, img)
    shutil.move(val_img, os.path.join(val_path, file_name))
#### split test file 分割测试集
os.mkdir(os.path.join(test_path, file_name))
for img_2 in test_file:
    test_img = os.path.join(img_path, img_2)
    shutil.move(test_img, os.path.join(test_path, file_name))

for file_name in files:
    split_val_test(file_name, 0.1, 0.1) # 0.1为分割比例

```

- 通过设置随机种子数调整数据划分。划分后将获得三份数据集文件。

1.3.2 图像格式的统一

- 考虑到图片数据的来源不同，图片的格式不一。因此为了方便后续操作（如模型读取图片数据），我们进一步将图片的格式进行**统一为JPEG**。代码如下：

```

from PIL import Image
import cv2 as cv
import os

im_path="/home/group6/deep_learning/data/images/"
train_path="/home/group6/deep_learning/experiment/zcl/images"
val_path="/home/group6/deep_learning/experiment/zcl/val/" #验证集路径
test_path="/home/group6/deep_learning/experiment/zcl/test/" # 测试集路径

def PNG_JPG(PngPath):
    img = cv.imread(PngPath, 0)
    w, h = img.shape[::-1]
    infile = PngPath
    outfile = os.path.splitext(infile)[0] + ".jpg"
    img = Image.open(infile)
    img = img.resize((int(w / 2), int(h / 2)), Image.ANTIALIAS)
    try:
        if len(img.split()) == 4:
            # prevent IOError: cannot write mode RGBA as BMP
            r, g, b, a = img.split()
            img = Image.merge("RGB", (r, g, b))
            img.convert('RGB').save(outfile, quality=70)
            os.remove(PngPath)
        else:
            img.convert('RGB').save(outfile, quality=70)
            os.remove(PngPath)
        return outfile
    except Exception as e:
        print("PNG转换JPEG 错误", e)

```

```

os.remove(PngPath)

def transfer(file_name, path):
    img_path = os.path.join(path, file_name)
    img_files = os.listdir(img_path)

    for filename in img_files:
        newname=filename
        newname = newname.split(".")
        if newname[-1] in ["JPEG", "jpeg"]:
            newname[-1] = "jpg"
            newname = str.join(".", newname)# 这里要用str.join
            filename = os.path.join(img_path,filename)
            newname = os.path.join(img_path,newname)
            os.rename(filename, newname)
            print(newname, "updated successfully")
        elif newname[-1] in ["jpg", "JPg"]:
            continue
        elif newname[-1] in ["png", "PNG"]:
            PNG_JPG(os.path.join(img_path,filename))

    for files_1 in os.listdir(im_path):
        transfer(files_1, im_path)

# for files_2 in os.listdir(test_path):
#     transfer(files_2, test_path)
#
# for files_3 in os.listdir(val_path):
#     transfer(files_3, val_path)

```

1.3.3 图像的解码、尺寸调整和归一化

- 解码：**我们知道在图像处理中，卷积神经网络的输入通常为三通道的RGB图像，从而进入卷积层时，卷积核可以分别在三个通道上进行卷积操作。因此训练模型前，我们在读取图片时需要利用二进制的解码，将原始RGB图片转化为具有三个通道的图片作为卷积神经网络的输入。
- 尺寸调整：**为了保证在经过固定结构的卷积神经网络后，全连接层的输出为尺寸固定的信息张量，因此我们需要确保图像的输入尺寸一致。
- 归一化：**图像的像素值默认类型为unit8，取值范围为[0,255]。而在后续卷积过程中，我们需要对double类型的数据进行运算处理，因此需要将unit8类型转化为double类型。而double类型数据的取值范围为[0,1]，故当转为double类型时，为了显示图像要除255。另外注意的是，在代码中为了得到浮点数，在除以255时需要写作“/255.”

以上操作的具体代码如下：

```

import tensorflow as tf
import os

def load_preprocess_image(input_path):
    image = tf.io.read_file(input_path) # 读取的是二进制格式 需要进行解码
    image = tf.image.decode_jpeg(image, channels=3) # 解码 是通道数为3
    image = tf.image.resize(image,[150,150]) # 统一图片大小

    image = image / 255.0 # 归一化
    return image

```

1.4 MongoDB存储

在经历了以上操作后，我们得到了一个数据集的文件夹。其文件结构为 数据集文件夹下内含26个子文件夹。代表26种花果。每一个种类文件夹内包含这个种类所有的图片

效果如下图

```

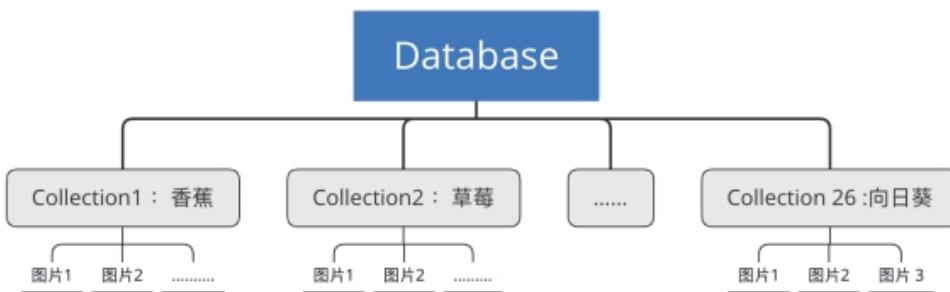
(base) [group6@node1 data]$ ls
images
(base) [group6@node1 data]$ cd images
(base) [group6@node1 images]$ ls
banana   cherry   dahlia   durian   gentian   lemon   mistflower   pineapple   sunflower
begonia  chrysanthemum  daisy   fig      globe amaranth litchi   nigella   pomegranate wallflower
blueberry cyclamen   delphinium four-o'clock grape     mango   orange   strawberry
(base) [group6@node1 images]$

```

1.4.1 数据库结构

我们已有用文件夹储存的数据，现在我们想将其**更方便的保存在MongoDB数据库中**

- 首先定义一个“Database”用来储存所有的图片，其中每一个“collection”为1类图片，“collection”中的每一条“documents”为单张图片。结构如下：



- 由于**MongoDB的文档结构为BJSON格式（BJSON全称：Binary JSON）**，而BJSON格式本身就支持保存二进制格式的数据，因此可以把文件的二进制格式的数据直接保存到MongoDB的文档结构中。但是由于一个BJSON的最大长度不能超过4M，所以限制了单个文档中能存入的最大文件不能超过4M。但偶尔会出现图片过大的情况。所以不适用将图片直接转为2进制的格式储存。

1.4.2 GridFS储存

- GridFS 也是文件存储的一种方式，但是它是存储在MongoDB的集合中。它可以更好的存储大于16M的文件。
- 所以整个流程代码如下：

```
from pymongo import MongoClient
from gridfs import *
import requests
img_path="/root/images"#图片储存位置

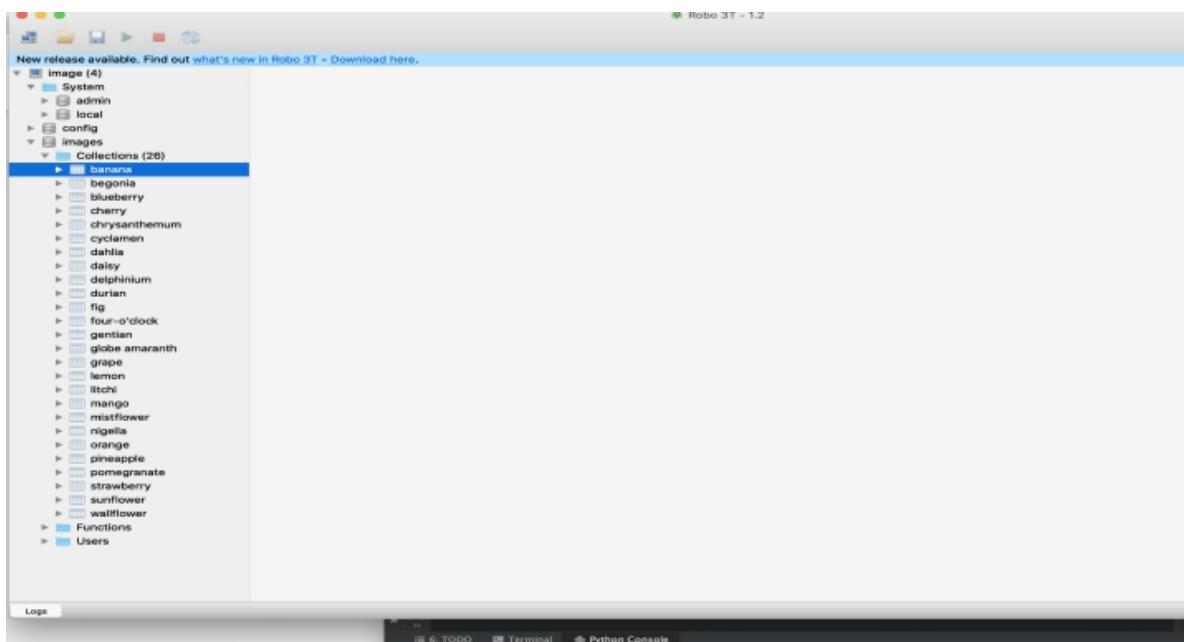
client = MongoClient('127.0.0.1', 27017) #连接mongodb
db = client["images"]#创建对应数据库

file_list=os.listdir(path)#获取种类

for class_name in file_list:
    mycol = db[class_name]
    fs = GridFS(db, collection="images") #连接collection#对每一个种类进行操作
    for img in os.path.join(path,files):
        with open (img,'rb') as myimage:
            data=myimage.read() #读取图片
            fs.put(data, content_type = "jpg", filename =img) #储存数据
```

- 成功后，images数据库下出现两个collection，分别为：images.files，images.chunks

运行以上代码，就可以将整个图片数据集保存在MongoDB数据库中。下图为部分运行结果。（使用Robo3T 数据库可视化软件）



二、算法介绍

在此任务中，我们的目的是识别26中花果，同时将算法部署至移动端设备。所以在设计所采用的算法时，有两个关键因素需要考虑：

1. 算法准确率

2. 算法计算量

算法准确率方面我们希望误分类个数尽可能少，同时尽可能希望我们的模型为轻量化。可以部署至树莓派上

- 模型的复杂度与硬件性能

1. 模型复杂度

模型复杂度可以表示为在进行推理时所需要的计算量。在CNN中，模型的计算量用如下的方法衡量

- **参数数量 (Params) :**

指模型含有多少参数，直接决定模型的大小，也影响推断时对内存的占用量
单位通常为 M，通常参数用 float32 表示，所以模型大小是参数数量的 4 倍左右

- **理论计算量 (FLOPs) :**

指模型推断时需要多少计算次数

可以用来衡量算法/模型的复杂度，这关系到算法速度，大模型的单位通常为 G (GFLOPs: 10亿次浮点运算)，小模型单位通常为 M

通常只考虑乘加操作(Multi-Adds)的数量，而且只考虑 CONV 和 FC 等参数层的计算量，忽略 BN 和 PReLU。

2.1 模型搭建

在这一部分，我们进行了大量实验，针对网络结构、模型参数等进行改进。在模型体积尽可能小的情况下，以求获得最佳的预测准确率。我们首先进行的是网络结构的实验，主要可分为以下四个方向。分别为仿照Alex Net的网络结构、仿照GoogleNet的网络结构、含残差块的网络结构、与MobileNet的迁移学习。在每一个网络结构实验中，我们均进行了大量的调参实验。最终经过对比，针对Mobile Net的迁移学习网络获得了最佳的效果。

实验环境

- python 3.8
- Anaconda3
- Tensorflow 2.2 (内置Keras)
- CUDA 10.1
- 调用GPU进行训练

```
import os
import warnings
warnings.filterwarnings("ignore")
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] ="0"
```

- 导入深度学习框架

```
import tensorflow as tf
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model,Sequential
from tensorflow.keras.optimizers import Adam, SGD,RMSprop
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import backend as K
```

```

import time
import numpy as np
from PIL import Image
from PIL import ImageFile
import tensorflow as tf
import random
import pathlib
from tensorflow.keras import backend as K
from tensorflow.keras.models import load_model
from tensorflow.keras.layers import
Conv2D,Dense,MaxPooling2D,Dropout,Flatten,BatchNormalization,add,Activation,Glob
alAveragePooling2D
from tensorflow.keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import os
from tensorflow.keras import regularizers

```

2.1.1 实验1：仿照Alex Net 网络结构

第一次训练的模型为由经典的AlexNet修改得来， AlexNet模型如下图所示：

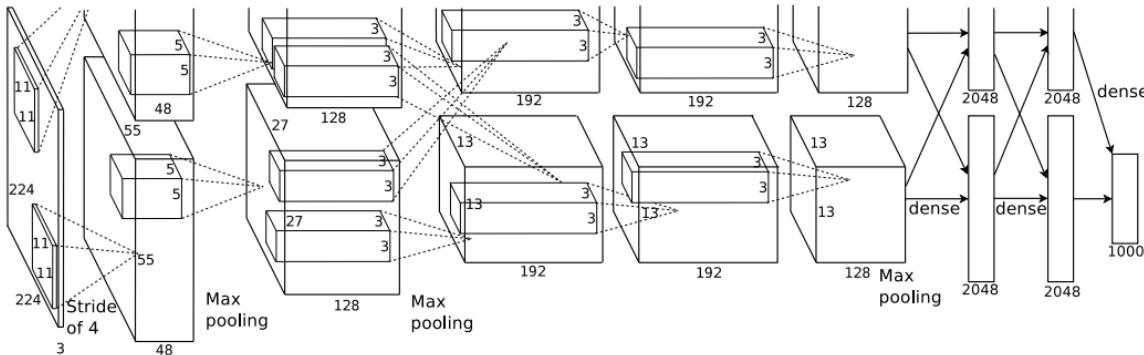


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

- 第一层：卷积层1，输入为 $224 \times 224 \times 3$ 的图像，卷积核的数量为96。
然后进行 (Local Response Normalized), 后面跟着池化 $\text{pool_size} = (3, 3)$, $\text{stride} = 2$, $\text{pad} = 0$
最终获得第一层卷积的feature map
- 第二层：卷积层2, 输入为上一层卷积的feature map, 卷积的个数为256个, 卷积核的大小为： $5 \times 5 \times 48$; $\text{padding} = 2$, $\text{stride} = 1$; 然后做 LRN, 最后 max_pooling , $\text{pool_size} = (3, 3)$, $\text{stride} = 2$;
- 第三层：卷积3, 输入为第二层的输出, 卷积核个数为384, $\text{kernel_size} = (3 \times 3 \times 256)$, $\text{padding} = 1$, 第三层没有做LRN和Pool
- 第四层：卷积4, 输入为第三层的输出, 卷积核个数为384, $\text{kernel_size} = (3 \times 3 \times 13)$, $\text{padding} = 1$, 和第三层一样, 没有LRN和Pool
- 第五层：卷积5, 输入为第四层的输出, 卷积核个数为256, $\text{kernel_size} = (3 \times 3 \times 13)$, $\text{padding} = 1$ 。然后直接进行 max_pooling , $\text{pool_size} = (3, 3)$, $\text{stride} = 2$;
- 第6,7,8层是全连接层, 每一层的神经元的个数为4096, 原文最终输出softmax为1000,这里根据水果和花卉的种类修改为25。全连接层中使用了RELU和Dropout。

代码

```
# coding=gbk
#import tensorflow
from keras.models import Sequential,load_model
from keras.layers import Dense, Dropout, Flatten, Conv2D,
MaxPooling2D,BatchNormalization
import keras
import time
from PIL import Image,ImageFile
import numpy as np
import os

ImageFile.LOAD_TRUNCATED_IMAGES = True
def imgToMat_RGB(img):
    img = Image.open(img)
    img = img.convert("RGB")
    #data = img.getdata()
    img = img.resize((227,227))
    mat = np.array(img)/255
    mat = mat.astype(np.float64)
    return mat

def get_train_data(data_folder_path):
    dirs = os.listdir(data_folder_path)
    data = []
    labels = []
    for dir_name in dirs:
        # i = 0;
        label = int(dir_name)
        subject_dir_path = data_folder_path + "/" + dir_name
        subject_images_name = os.listdir(subject_dir_path)
        for image_name in subject_images_name:
            # i+=1
            # if(i>=800):
            #
            break
            image_path = subject_dir_path + "/" + image_name
            image = imgToMat_RGB(image_path)
            data.append(image)
            labels.append(label)
    #print(labels)
    return np.array(data),np.array(labels)

def shuffle_data():
    X,Y = get_train_data("data/train")
    # print(Y,"1")
    Y = keras.utils.to_categorical(Y)
    # print(Y,"2")
    index = [i for i in range(len(Y))] # test_data为测试数据
    np.random.seed(1)
    np.random.shuffle(index) # 打乱索引
    train_data = X[index]
    train_labels = Y[index]
    print(train_labels)
    return train_data,train_labels
```

```

def define_model():
    model = Sequential()
    model.add(Conv2D(filters=96,
                    kernel_size=(11,11),
                    strides=(4,4),
                    padding='same',
                    input_shape=(227,227,3),
                    activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(3,3),
                          strides=(2,2),
                          padding='valid'))

    model.add(Conv2D(filters=256,
                    kernel_size=(5,5),
                    strides=(1,1),
                    padding='same',
                    activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(3,3),
                          strides=(2,2),
                          padding='valid'))

    model.add(Conv2D(filters=384,
                    kernel_size=(3,3),
                    strides=(1,1),
                    padding='same',
                    activation='relu'))
    model.add(Conv2D(filters=384,
                    kernel_size=(3,3),
                    strides=(1,1),
                    padding='same',
                    activation='relu'))
    model.add(Conv2D(filters=256,
                    kernel_size=(3,3),
                    strides=(1,1),
                    padding='same',
                    activation='relu'))

    model.add(MaxPooling2D(pool_size=(3,3),
                          strides=(2,2),
                          padding='valid'))

    model.add(Flatten())
    model.add(Dense(4096,activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096,activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000,activation='relu'))
    model.add(Dropout(0.5))

    model.add(Dense(25,activation='softmax'))
    model.compile(optimizer='sgd',loss='categorical_crossentropy',metrics=['accuracy'])
    model.summary()
    return model

def train_model():
    start_time = time.time()

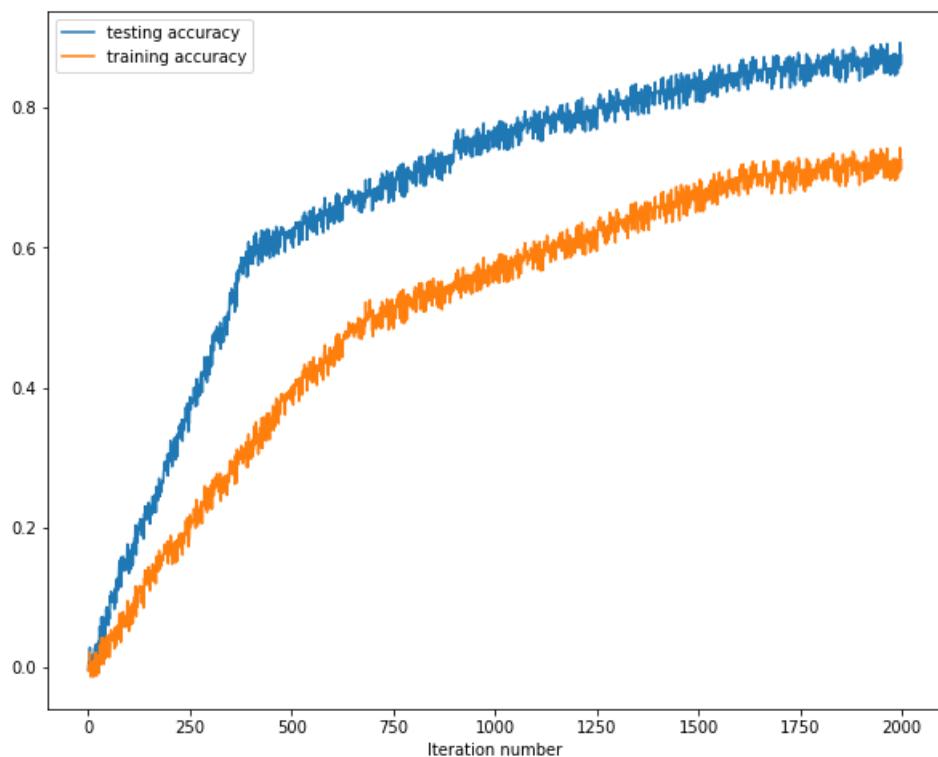
```

```
model = define_model()
#    model = load_model("model1.h5")
train_data,train_labels = shuffle_data()
#print(train_data)
model.fit(train_data,train_labels,epochs=100)
model.save('model.h5')

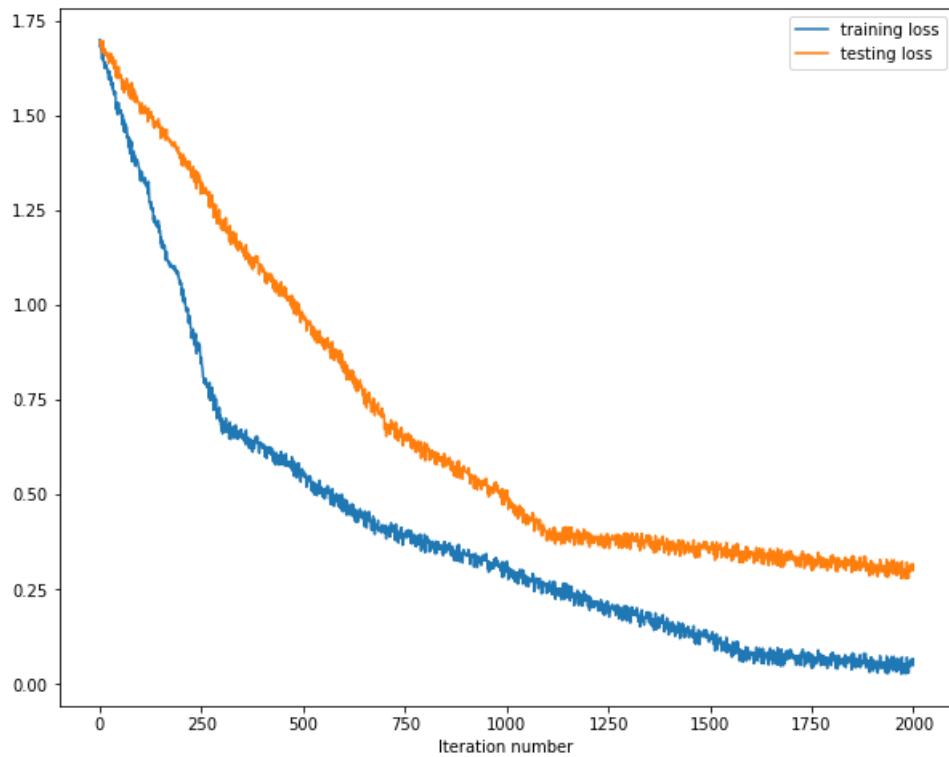
end_time = time.time()
run_time = (end_time-start_time)
print(run_time)
train_model()
```

训练过程

- Accuracy曲线:



- loss曲线:



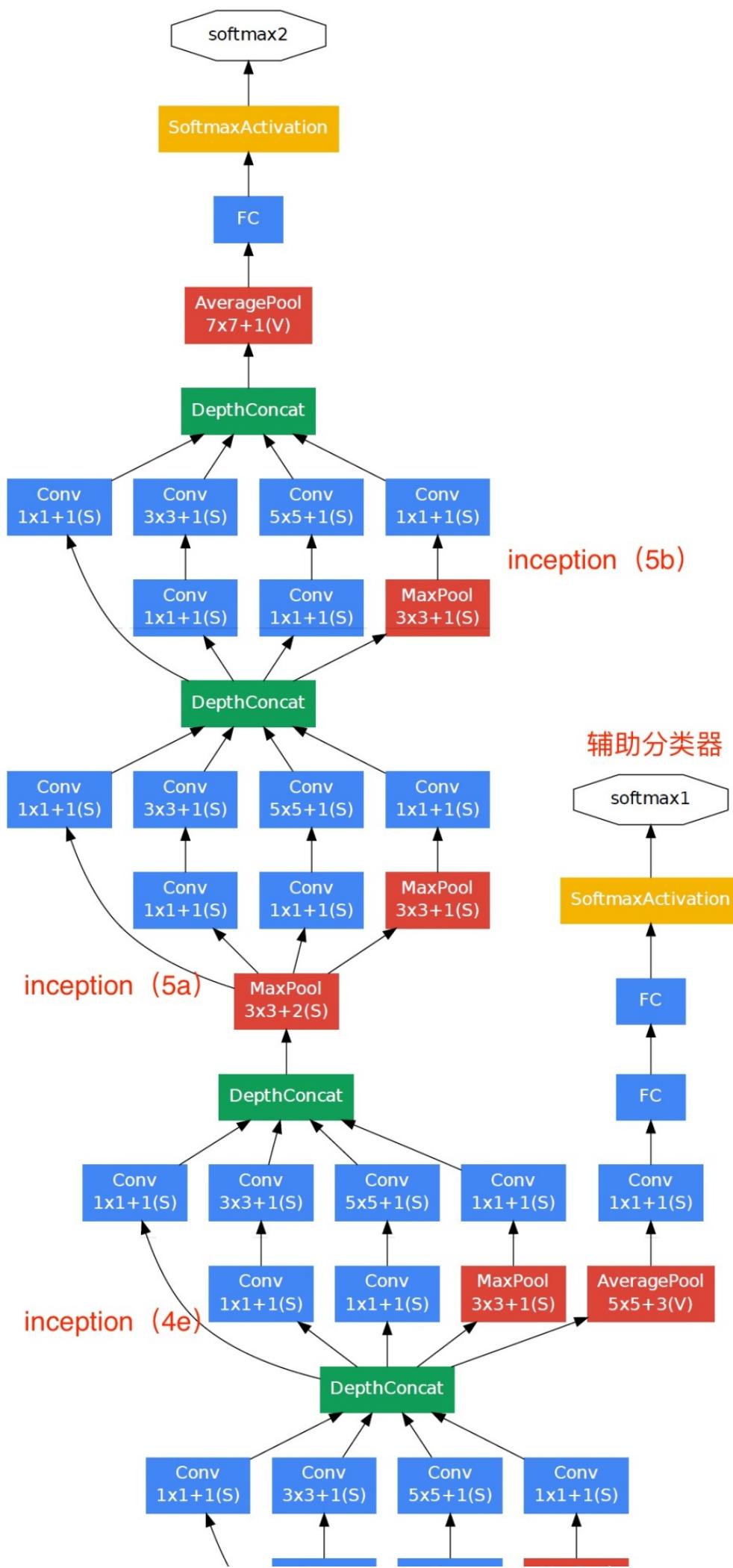
评估

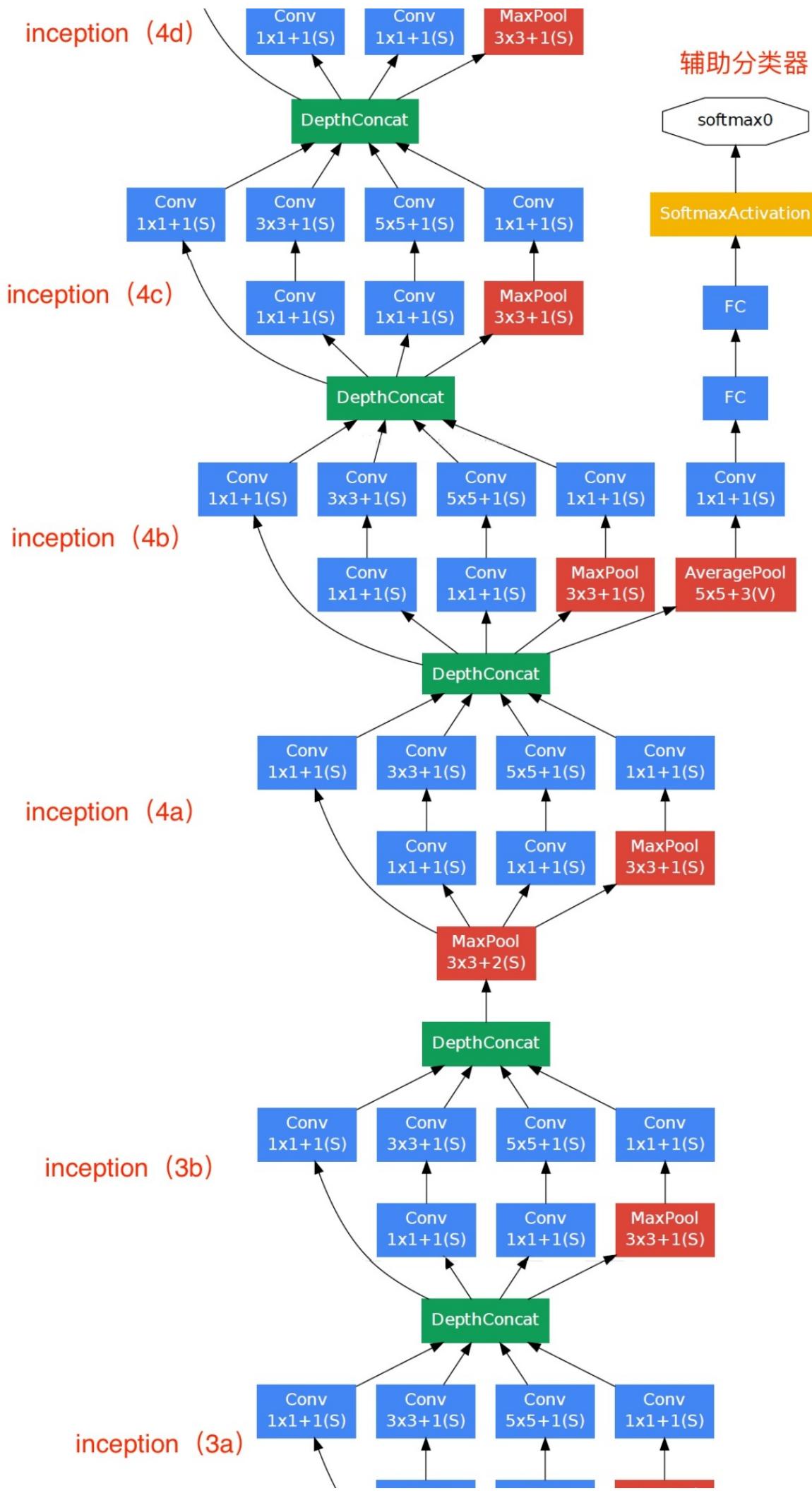
- 经过2000个epoch:
 - 训练集loss为0.09，准确率为92.18%
 - 验证集loss为0.31，准确率为70.56%
- 模型总参数为5240360
- 在测试集上的F1-Score : 74.832423
- 模型大小为123MB
- 训练总时长为8.5小时
- 由于准确率不高，且模型过大，经tflite格式转化后仍无法在树莓派zero上运行

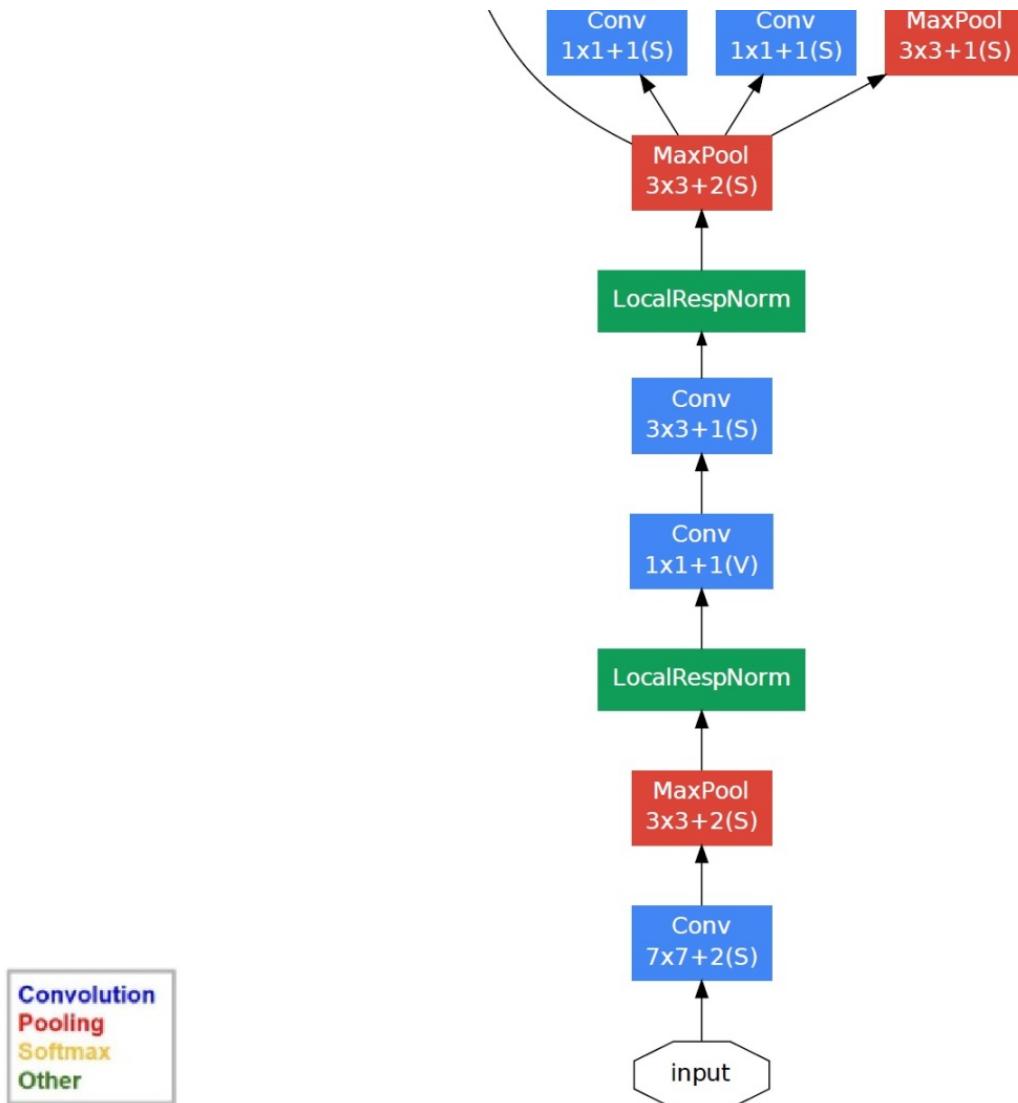
2.1.2 实验2：

模型介绍

结构图如下：







- 各层结构:

0、输入

原始输入图像为 $224 \times 224 \times 3$ ，且都进行了零均值化的预处理操作（图像每个像素减去均值）。

1、第一层（卷积层）

使用 7×7 的卷积核（滑动步长2, padding为3），64通道，输出为 $112 \times 112 \times 64$ ，卷积后进行ReLU操作

经过 3×3 的max pooling（步长为2），输出为 $((112 - 3 + 1)/2) + 1 = 56$ ，即 $56 \times 56 \times 64$ ，再进行ReLU操作

2、第二层（卷积层）

使用 3×3 的卷积核（滑动步长为1, padding为1），192通道，输出为 $56 \times 56 \times 192$ ，卷积后进行ReLU操作

经过 3×3 的max pooling（步长为2），输出为 $((56 - 3 + 1)/2) + 1 = 28$ ，即 $28 \times 28 \times 192$ ，再进行ReLU操作

3a、第三层（Inception 3a层）

分为四个分支，采用不同尺度的卷积核来进行处理

(1) 64个 1×1 的卷积核，然后ReLU，输出 $28 \times 28 \times 64$

(2) 96个 1×1 的卷积核，作为 3×3 卷积核之前的降维，变成 $28 \times 28 \times 96$ ，然后进行ReLU计算，再进行128个 3×3 的卷积（padding为1），输出 $28 \times 28 \times 128$

(3) 16个 1×1 的卷积核，作为 5×5 卷积核之前的降维，变成 $28 \times 28 \times 16$ ，进行ReLU计算后，再进行32个 5×5 的卷积（padding为2），输出 $28 \times 28 \times 32$

(4) pool层，使用 3×3 的核（padding为1），输出 $28 \times 28 \times 192$ ，然后进行32个 1×1 的卷积，输出 $28 \times 28 \times 32$ 。

将四个结果进行连接，对这四部分输出结果的第三维并联，即 $64 + 128 + 32 + 32 = 256$ ，最终输出

28x28x256

3b、第三层 (Inception 3b层)

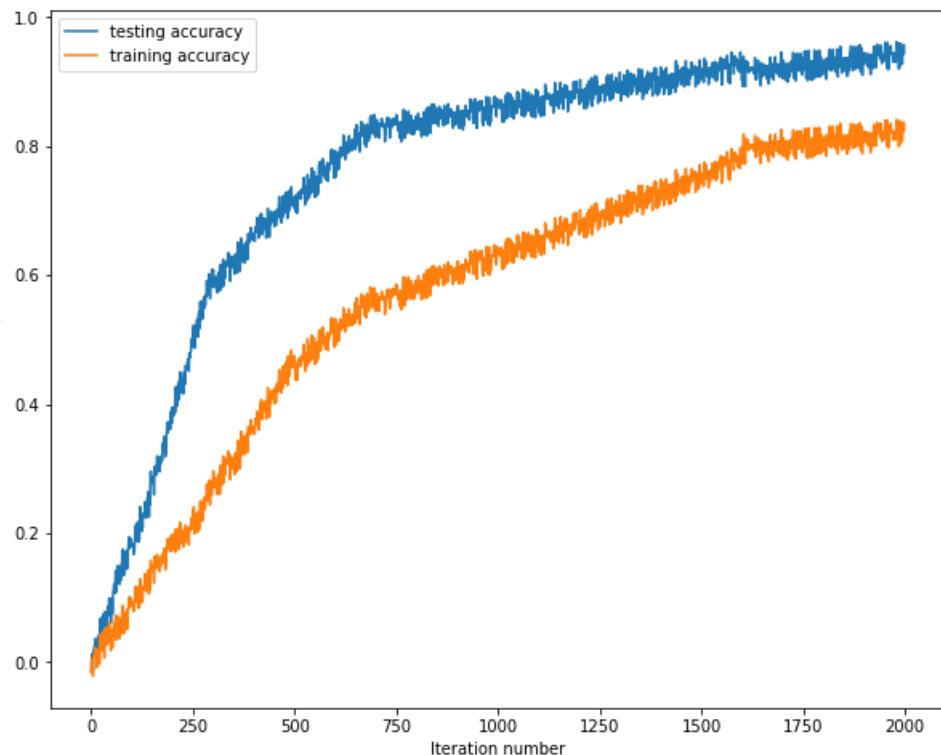
- (1) 128个 1×1 的卷积核，然后ReLU，输出 $28\times 28\times 128$
- (2) 128个 1×1 的卷积核，作为 3×3 卷积核之前的降维，变成 $28\times 28\times 128$ ，进行ReLU，再进行192个 3×3 的卷积 (padding为1)，输出 $28\times 28\times 192$
- (3) 32个 1×1 的卷积核，作为 5×5 卷积核之前的降维，变成 $28\times 28\times 32$ ，进行ReLU计算后，再进行96个 5×5 的卷积 (padding为2)，输出 $28\times 28\times 96$
- (4) pool层，使用 3×3 的核 (padding为1)，输出 $28\times 28\times 256$ ，然后进行64个 1×1 的卷积，输出 $28\times 28\times 64$ 。

将四个结果进行连接，对这四部分输出结果的第三维并联，即 $128+192+96+64=480$ ，最终输出输出为**28x28x480**

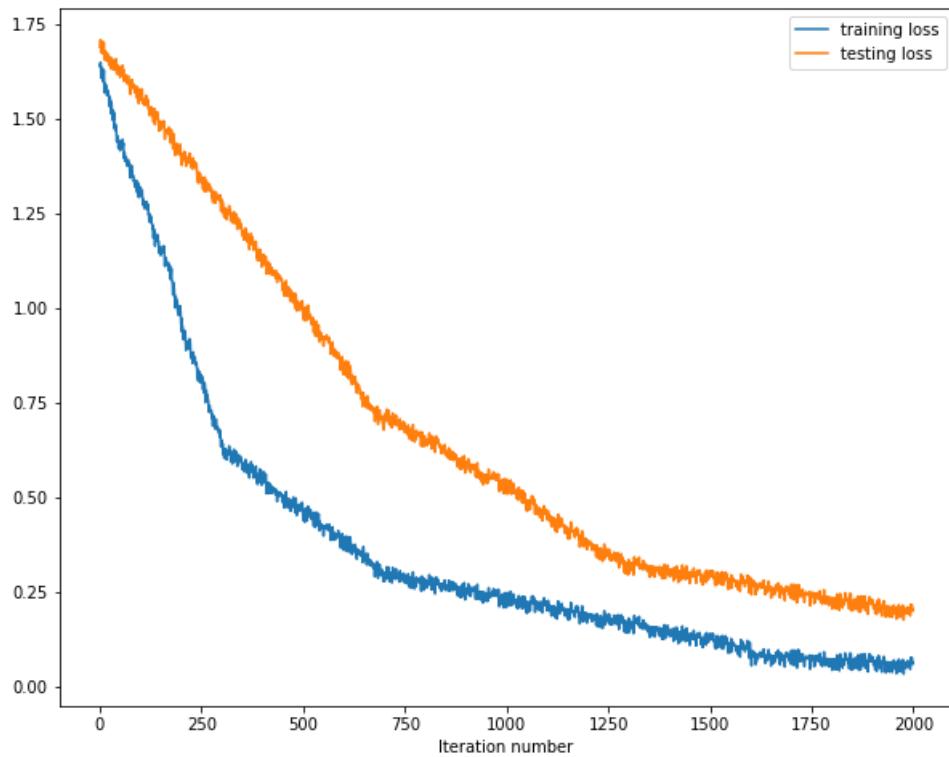
第四层 (4a,4b,4c,4d,4e) 、第五层 (5a,5b)，与3a、3b类似，在此就不再重复。

训练过程：

- Accuracy曲线：



- loss曲线：



模型评估

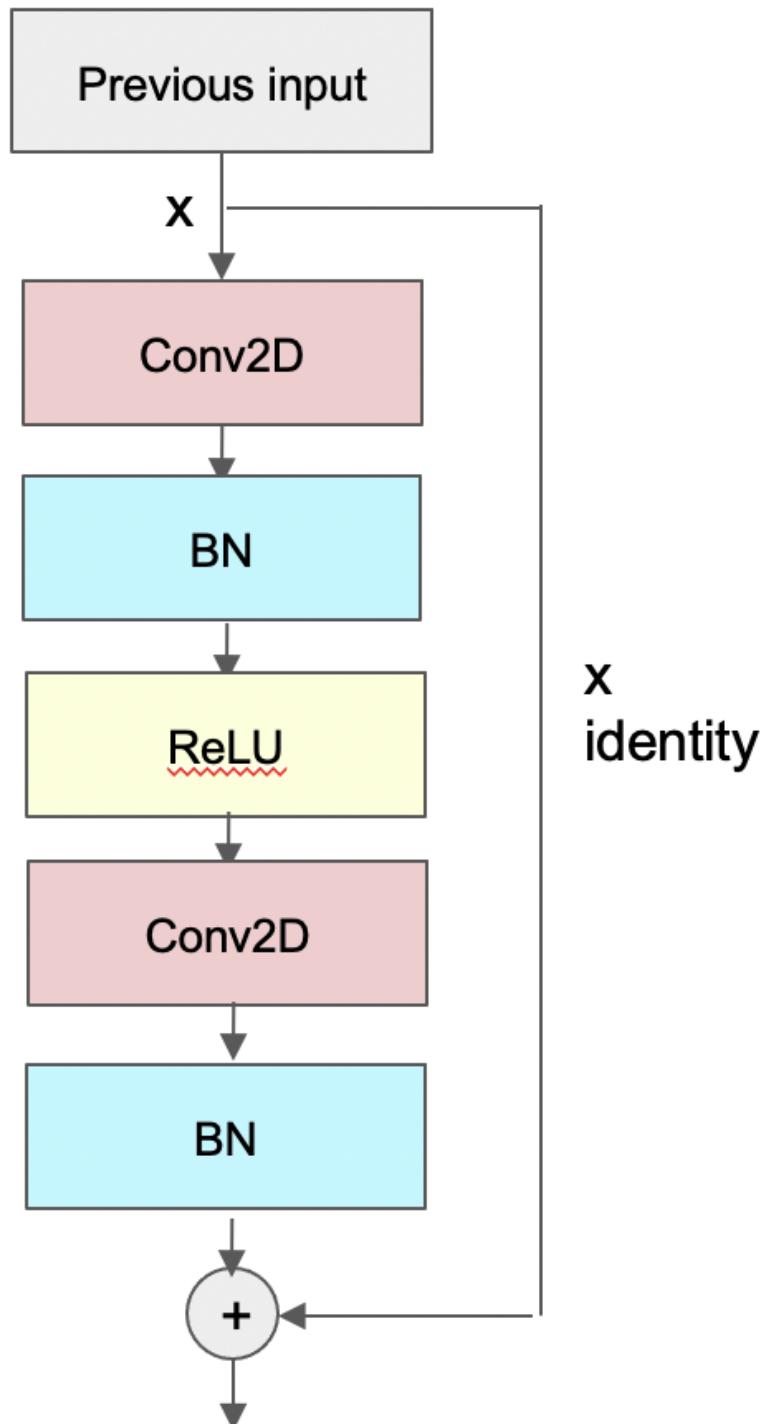
- 经过2000个epoch:
 - 训练集Loss为0.06，准确率为98.82%
 - 验证集Loss为0.19，准确率为81.64%
- 模型总参数为62404777
- 在测试集上的F1-Score : 80.8946423
- 模型大小为231MB
- 训练总时长为25小时
- 由于模型过于庞大，即便转化为优化后的tflite格式，在树莓派上运行时间过长，故无法实际使用。

2.1.3 实验3 含残差块的网络结构

- **残差块实现**

残差网络在图像领域已然成为了一种主流模型，虽然这种网络范式的提出是为了解决网络退化问题，同时可有效防止过拟合

- 残差块结构图

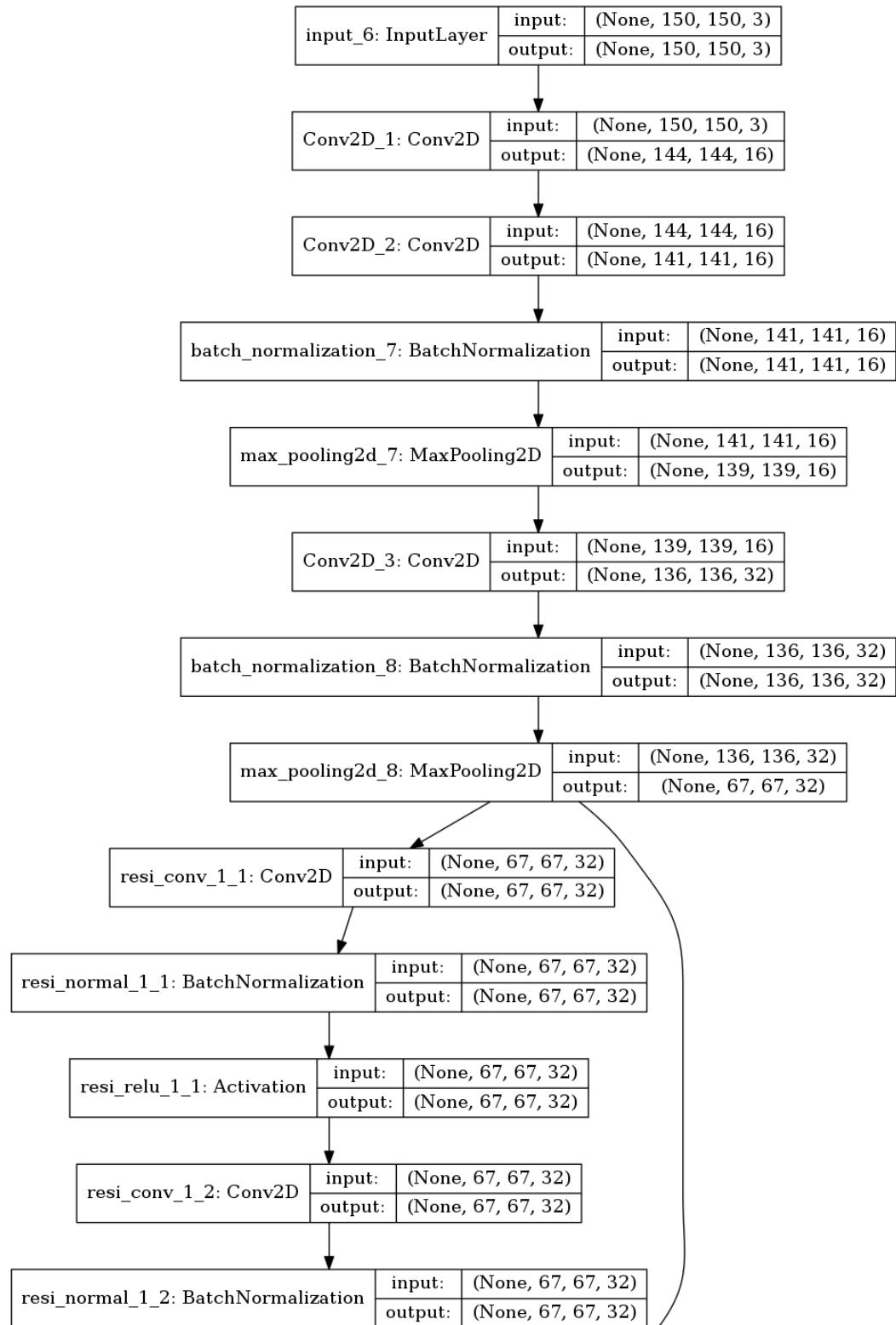


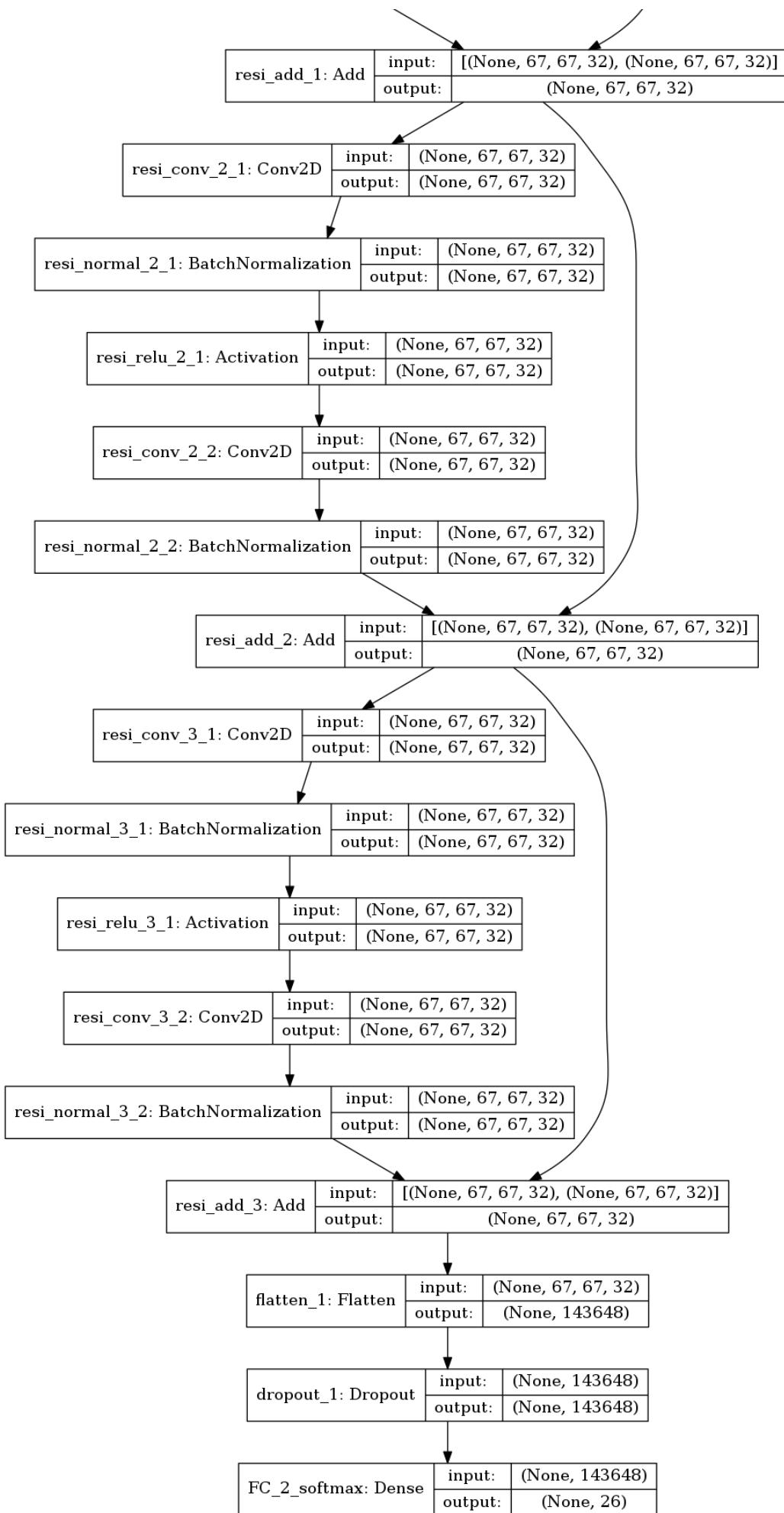
```

def residual_block(x, num,filters,size):
    shortcut = x
    x = Conv2D(filters, (size, size), strides=1, padding='same',
name='resi_conv_%d_1' % num)(x)
    x = BatchNormalization(name='resi_normal_%d_1' % num)(x)
    x = Activation('relu', name='resi_relu_%d_1' % num)(x)
    x = Conv2D(filters, (size, size), strides=1, padding='same',
name='resi_conv_%d_2' % num)(x)
    x = BatchNormalization(name='resi_normal_%d_2' % num)(x)
    m = add([x, shortcut], name='resi_add_%d' % num)
    return m

```

- 网络结构





```
def fruit_model(input_shape):
```

```

X_input = Input(input_shape)
X = Conv2D(16, (7, 7), activation='relu', name="Conv2D_1")(X_input)
X = Conv2D(16, (4, 4), activation='relu', name="Conv2D_2",
kernel_regularizer=regularizers.l1(0.01))(X)
X = BatchNormalization(momentum=0.99, epsilon=0.001)(X)
X = MaxPooling2D(pool_size=(3, 3), strides=1, padding='valid')(X)
X = Conv2D(32, (4, 4), activation='relu', name="Conv2D_3",
kernel_regularizer=regularizers.l1(0.01))(X)
X = BatchNormalization(momentum=0.99, epsilon=0.001)(X)
X = MaxPooling2D(pool_size=(3, 3), strides=2, padding='valid')(X)

X=residual_block(X,1,32,3)
X=residual_block(X,2,32,3)
X=residual_block(X,3,32,3)
X = Flatten()(X)

X = Dropout(0.5)(X)
X = Dense(26, activation='softmax', name='FC_2_softmax')(X)
model = Model(inputs=X_input, outputs=X, name='fruit_model')

model.compile(loss='categorical_crossentropy', optimizer="adam", metrics=['accuracy'])
return model

```

- 训练过程

```

# 使用Tensorflow 读取数据集
def get_image(path, batch_size, input_size, NUM, kinds):
    AUTOTUNE = tf.data.experimental.AUTOTUNE
    data_path = pathlib.Path(path)
    all_image_paths = list(data_path.glob('*/*'))
    all_image_paths = [str(path) for path in all_image_paths] # 所有图片路径的列表
    image_count = len(all_image_paths)
    random.shuffle(all_image_paths)
    label_names = sorted(item.name for item in data_path.glob('*')
if item.is_dir())
    path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
    image_ds = path_ds.map(load_preprocess_image, num_parallel_calls=AUTOTUNE)
    label_to_index = dict((name, index) for index, name in
enumerate(label_names))
    all_image_labels = [label_to_index[pathlib.Path(path).parent.name] for path
in all_image_paths]
    # one hot
    all_image_labels = tf.one_hot(all_image_labels, depth=NUM)
    label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(all_image_labels,
tf.int64))
    image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
    ds = image_label_ds.shuffle(buffer_size=image_count)
    ds = ds.repeat()
    ds = ds.batch(batch_size)
    # 当模型在训练的时候，`prefetch` 使数据集在后台取得 batch。
    ds = ds.prefetch(tf.data.experimental.AUTOTUNE)
    return ds

# 开始训练
def train(total_epoch, train_path, input_shape):
    save_dir = "/home/group6/deep_learning/experiment/zcl/model_2"

```

```

train_set=get_image(train_path, batch_size, input_shape, 26, "train")
val_set=get_image(val_path, val_size, input_shape, 26, "val")

our_model = fruit_model((input_shape[0], input_shape[1], 3))

#our_model=mobile()
save_fname = os.path.join(save_dir, 'model_at_{epoch:02d}.h5') # 将时间和
epoch id存到模型名称中
# 设置回调函数 当准确率不再上升或下降时保存模型
callback = [ModelCheckpoint(filepath=save_fname, monitor='val_acc',
save_best_only=False, save_freq=200, mode='max')]
print(our_model.summary())
print("开始训练")
History = our_model.fit(
    train_set,
    steps_per_epoch=train_img_num // batch_size,
    epochs=total_epoch,
    validation_data=val_set,
    validation_steps=10,
    callbacks=callback
)
print("训练完成")

our_model.save("/home/group6/deep_learning/experiment/zcl/model_1/final_model.h
5")
print("保存成功")
#####可视化
plt.plot(History.history['accuracy'])
plt.plot(History.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.savefig('/home/group6/deep_learning/experiment/zcl/model_2/pic/acc.png')
plt.clf()
plt.plot(History.history['loss'])
plt.plot(History.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.savefig('/home/group6/deep_learning/experiment/zcl/model_2/pic/loss.png')

if __name__ == "__main__":
    train(total_epoch, train_path, input_shape)

```

- 模型评估

```

Model=load_model("/group6/deep_learning/final_model.h5")
datagen = ImageDataGenerator(rescale=1./255)
test_generator = datagen.flow_from_directory(
    test_path,
    target_size=input_shape,
    batch_size=800,
    seed=1,

```

```

class_mode='sparse')

# 用测试集计算F1-Score
for x,y in test_generator:
    x=2*x-1
    y_pred=Model1.predict(x)
    predict_label=np.argmax(y_pred, axis=1)

    f1_score(y,predict,average='macro')
    #C=confusion_matrix(y,predict_label)
    break

```

F1-score :63.375698

2.1.4 实验4：迁移学习（Mobile-Net V2）

1. 迁移学习原理

迁移学习(Transfer learning) 就是把已学训练好的模型参数迁移到新的模型来帮助新模型训练。考虑到大部分图片数据或任务是存在相关性的，所以通过迁移学习我们可以将已经学到的模型参数（也可理解为模型学到的知识）通过某种方式来分享给新模型从而加快并优化模型的学习效率不用像大多数网络那样从零学习（starting from scratch, tabula rasa）。一般操作方式为 1.选取合适的预训练模型 2.固定网络的前若干层 3.根据任务需求修改网络Softmax输出层 4.在自己的数据集上开始训练

2. Mobile Net 介绍

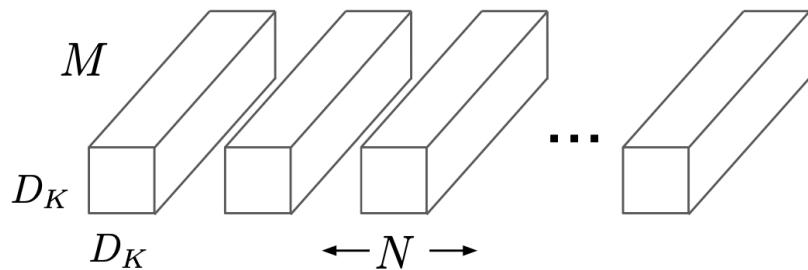
MobileNet 由谷歌在 2017 年提出，是一款专注于在移动设备和嵌入式设备上的 轻量级 CNN 神经网络，并迅速衍生了 v1 v2 v3 三个版本；我们选择 MobileNet V2 作为迁移学习的与训练模型。相比于传统的 CNN 网络，在准确率小幅降低的前提下，大大减小模型参数和运算量；

- 深度可分离卷积

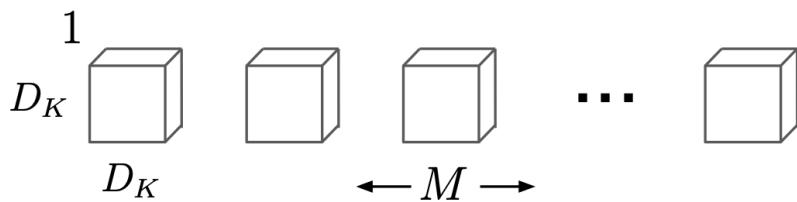
MobileNet 模型的核心就是深度可分离卷积，它是因式分解卷积的一种。

具体地，深度可分离卷积将标准化卷积分解为逐深度卷积（depthwise convolution）和逐点 1×1 卷积（pointwise convolution）。对于 MobileNets，逐个深度卷积将单个滤波器应用到每一个输入通道。然后，逐点卷积用 1×1 卷积来组合不同深度卷积的输出。在一个步骤，一个标准的卷积过程将输入滤波和组合成一组新的输出。深度可分离卷积将其分成两层，一层用于滤波，一层用于组合。这种分解过程能极大减少计算量和模型大小。下图展示了如何将一个标准卷积分解为深度卷积

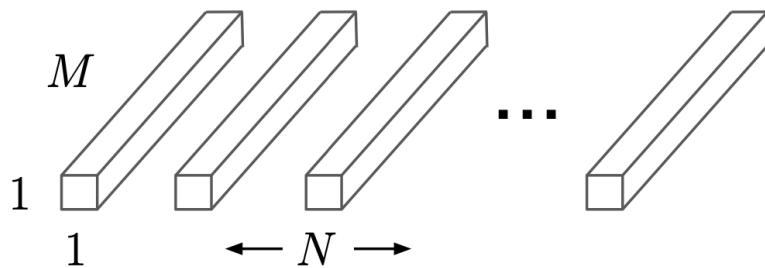
和 1×1 逐点卷积。



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

- **网络结构**

每行描述一个或多个相同（步长）层的序列，每个bottleneck重复n次。相同序列中的所有层具有相同数量的输出通道。每个序列的第一层有使用步长s，所有其他层使用步长1。所有的空间卷积使用 3×3 的内核。扩展因子t始终应用于输入大小。假设输入某一层的tensor的通道数为k，那么应用在这一层上的filters数就为 $k * t$

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$28^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times k$	conv2d 1x1	-	k	-	-

Table 2: MobileNetV2 : Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size as described in Table 1.

<https://blog.csdn.net/u011974639>

其中 迁移学习代码为，除去最后的softmax层。保留网络前的所有参数和结构。只添加一层平均池化层，和修改过输出的Softmax层。

```
def transfer_mobile():
    mobile_net = tf.keras.applications.MobileNetV2(input_shape=(192, 192, 3),
include_top=False)
    mobile_net.trainable = True
    model = tf.keras.Sequential([
        mobile_net,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(26, activation='softmax')])
    model.compile(optimizer=tf.keras.optimizers.Adam(),
                  loss='sparse_categorical_crossentropy',
                  metrics=["accuracy"])
    return model
```

原网络的输入大小为 (224, 224) , 为了进一步缩小模型体积,我们缩小了网络输入大小至 (192, 192)

• 训练过程

```
# 使用Tensorflow 读取数据集
def get_image(path, batch_size, input_size, NUM, kinds):
    AUTOTUNE = tf.data.experimental.AUTOTUNE
    data_path = pathlib.Path(path)
    all_image_paths = list(data_path.glob('*/*'))
    all_image_paths = [str(path) for path in all_image_paths] # 所有图片路径的列表
    image_count = len(all_image_paths)
    random.shuffle(all_image_paths)
    label_names = sorted(item.name for item in data_path.glob('*/*') if
item.is_dir())
    path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
    image_ds = path_ds.map(load_preprocess_image, num_parallel_calls=AUTOTUNE)
    label_to_index = dict((name, index) for index, name in
enumerate(label_names))
    all_image_labels = [label_to_index[pathlib.Path(path).parent.name] for path
in all_image_paths]
    # one hot
    all_image_labels = tf.one_hot(all_image_labels, depth=NUM)
    label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(all_image_labels,
tf.int64))
    image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
    ds = image_label_ds.shuffle(buffer_size=image_count)

    ds = ds.repeat()
    ds = ds.batch(batch_size)
    # 当模型在训练的时候，`prefetch` 使数据集在后台取得 batch。
    ds = ds.prefetch(tf.data.experimental.AUTOTUNE)
    return ds

# 开始训练
def train(total_epoch, train_path, input_shape):
    save_dir = "/home/group6/deep_learning/experiment/zcl/model_2"
    train_set = get_image(train_path, batch_size, input_shape, 26, "train")
    val_set = get_image(val_path, val_size, input_shape, 26, "val")
    our_model = transfer_mobile()
    #our_model=mobile()
    save_fname = os.path.join(save_dir, 'model_at_{epoch:02d}.h5') # 将时间和
epoch id存到模型名称中
    # 设置回调函数 当准确率不再上升或下降时保存模型
    callback = [ModelCheckpoint(filepath=save_fname, monitor='val_acc',
save_best_only=False, save_freq=200, mode='max')]
    print(our_model.summary())
    print("开始训练")
    History = our_model.fit(
        train_set,
        steps_per_epoch=train_img_num // batch_size,
        epochs=total_epoch,
        validation_data=val_set,
        validation_steps=10,
        callbacks=callback
    )
    print("训练完成")

    our_model.save("/home/group6/deep_learning/experiment/zcl/model_1/final_model.h
5")
```

```

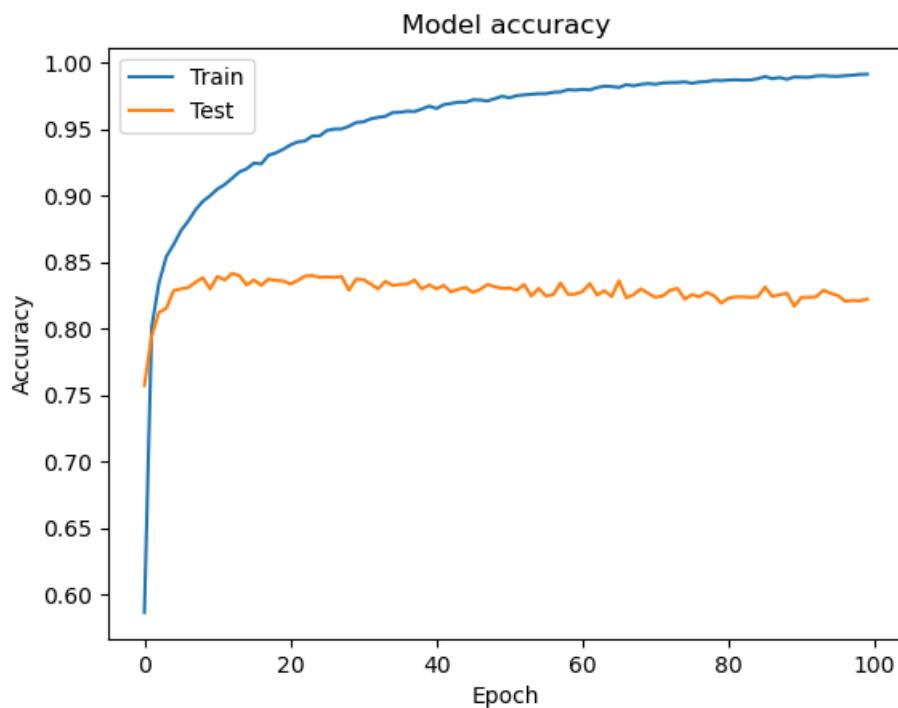
print("保存成功")
#####可视化
plt.plot(History.history['accuracy'])
plt.plot(History.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.savefig('/home/group6/deep_learning/experiment/zcl/model_2/pic/acc.png')
plt.clf()
plt.plot(History.history['loss'])
plt.plot(History.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

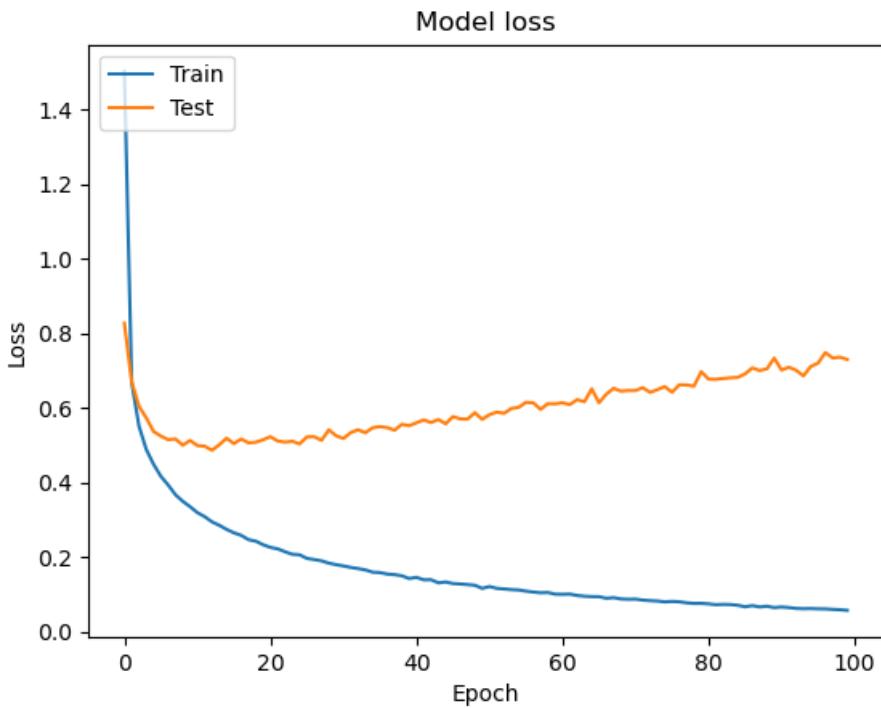
plt.savefig('/home/group6/deep_learning/experiment/zcl/model_2/pic/loss.png')

if __name__ == "__main__":
    train(total_epoch, train_path, input_shape)

```

- 训练曲线如下





- 模型评估

```

datagen = ImageDataGenerator(rescale=1./255)
test_generator = datagen.flow_from_directory(
    test_path,
    target_size=input_shape,
    batch_size=800,
    seed=1,
    class_mode='sparse')
# 用测试集计算F1-Score 和混淆矩阵
for x,y in test_generator:
    x=2*x-1
    y_pred=Model1.predict(x)
    predict_label=np.argmax(y_pred, axis=1)
    #true_classes = test_generator.classes
    #class_labels = list(test_generator.class_indices.keys())
    f1_score(y,predict,average='macro')
    #C=confusion_matrix(y,predict_label)
    break

```

F1 Score :0.9188857213160468

```

import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(25, 23))
plt.figure(dpi=180)
sns.set()
f,ax=plt.subplots()
C = C.astype('float') / C.sum(axis=1)[:, np.newaxis] #normalize the confusion
matrix
sns.heatmap(C, annot=True, ax=ax, cmap="Blues_r") #plot the matrix

ax.set_title('confusion matrix') #set title
ax.set_xlabel('predict') # x label
ax.set_ylabel('true') #Y label
plt.show()

```

2.2 模型选择

2.2.1 因素比较

模型	F1-Score	模型大小	参数量
模型1	74.	123MB	5240360
模型2	81.	231MB	62404777
模型3	63.	27MB	3878,900
模型4	91.	9.3MB	2,291,290

在这些结果中，我们惊喜的发现。经过迁移学习训练而得的模型正确率不仅最高，模型的体积也最小最易于部署

所以 根据模型在测试集上的表现，我们选取模型4作为我们部署所用的模型。

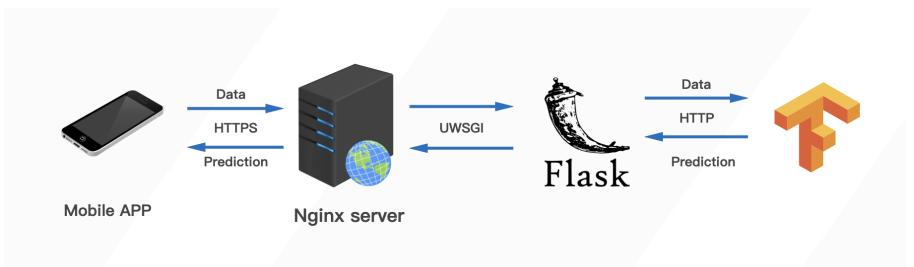
2.3 服务器部署

2.3.1 配置服务器

所需软件/环境

- 基本Python环境
- Flask
- UWSGI
- Nginx
- Docker
- Tensorflow Serving

具体工作流程为



其中在Docker中运行的Tensorflow Serving 可以部署我们之前训练好的模型，并接受Flask客户端发来的数据，并将预测结果返回

服务器代码

```

from imageio import imwrite,imread
import numpy as np
from PIL import Image
#import nets
import os
import string
import random
import json
import requests
from flask import Flask, request, redirect, url_for, render_template,send_file
import base64

MODEL_URI = 'http://localhost:8502/v1/models/mobile:predict' #Tensorflow serving 的使用方式
OUTPUT_DIR = 'static'
CLASSES = ['cat', 'Dog']

app = Flask(__name__)
class_dict={0: 'banana', 1: 'begonia', 2: 'blueberry', 3: 'cherry', 4: 'chrysanthemum', 5: 'cyclamen', 6: 'dahlia', 7: 'daisy', 8: 'delphinium', 9: 'durian', 10: 'fig', 11: "four-o'clock", 12: 'gentian', 13: 'globe amaranth', 14: 'grape', 15: 'lemon', 16: 'litchi', 17: 'mango', 18: 'mistflower', 19: 'nigella', 20: 'orange', 21: 'pineapple', 22: 'pomegranate', 23: 'strawberry', 24: 'sunflower', 25: 'wallflower'}
```

```

def imgToMat_RGB(img):
    img = Image.open(img)
    img = img.convert("RGB")
    #data = img.getdata()
    img = img.resize((192,192))
    mat = np.array(img)/255.0
    mat = mat.astype(np.float64)
    mat=2*mat-1
    return np.array([mat])

def get_prediction(image_path):
#    image = tf.keras.preprocessing.image.load_img(image_path, target_size=(SIZE, SIZE))
#    image = tf.keras.preprocessing.image.img_to_array(image)
#    image = tf.keras.applications.mobilenet_v2.preprocess_input(image)
    image = imgToMat_RGB(image_path)
    print(image.shape)
    data = json.dumps({'instances': image.tolist() })

```

```

response = requests.post(MODEL_URI, data=data.encode())
result = json.loads(response.text)
print(result)
prediction = result['predictions'][0]
prediction = np.array(prediction)
index=np.argmax(prediction)
return class_dict[index]

def generate_filename():
    a=random.choices(string.ascii_lowercase, k=20)
    path=''.join(a) + '.jpg'
    name=''.join(a)
    return path,name

@app.route('/', methods=['POST'])
def index():
    return_dict= {'return_code': '404',"return_info":"xxx"}
    if request.method == 'POST':
        uploaded_file = request.files['file']
        print(uploaded_file)
        if uploaded_file.filename != '':
            if uploaded_file.filename[-3:] in ['jpg', 'png']:
                path_new,name=generate_filename()
                image_path = os.path.join("images", path_new)
                print(image_path)
                uploaded_file.save(image_path)

                print("ready")
                class_name = get_prediction(image_path)
                result = {
                    'class_name': class_name,
                }
                print("OK")
                return json.dumps(result)
#
    return json.dumps(return_dict)

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=3900)

```

小程序部署

在服务区上部署成功之后，我们可以通过微信小程序发送POST请求至服务器。之后服务器会将识别结果返回给小程序。因为个人所能申请的小程序数量有限。我们在之前小程序的基础上添加了一个用于识别水果的页面



HIIT 动作识别

•••



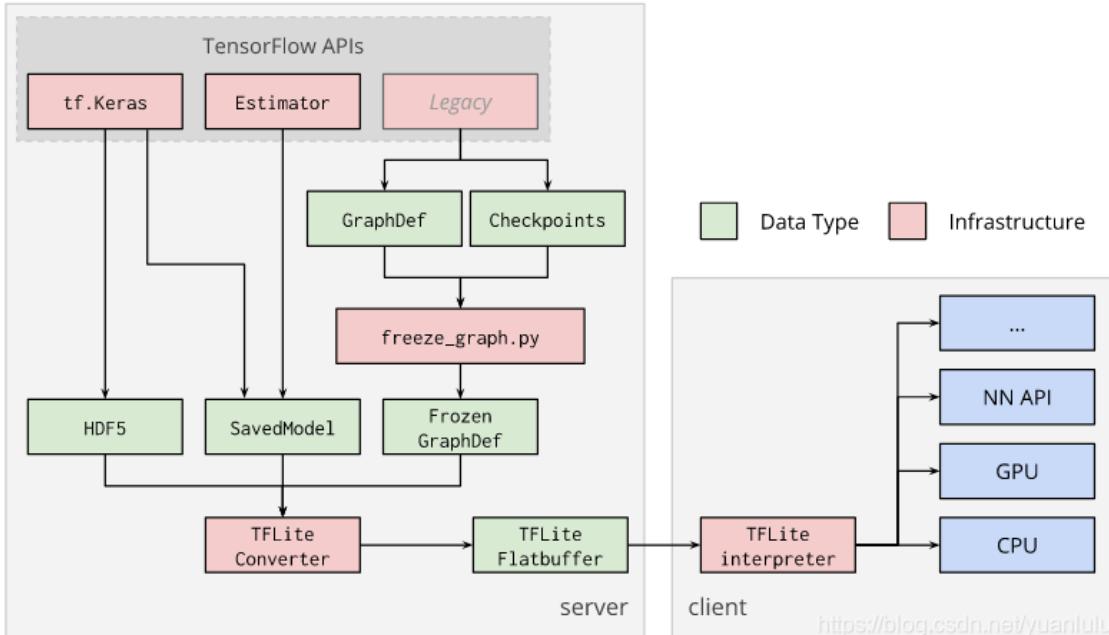
```
{"class_name": "banana"}
```

开始

三、模型部署

3.1 TFLite介绍

- 由于h5格式模型无法很好的在树莓派zero上运行，故采用专为移动端使用的tflite格式模型。
- tflite使用的思路主要是从预训练的模型转换为tflite模型文件，拿到移动端部署。
- tflite的源模型可以来自tensorflow的saved model或者frozen model,也可以来自keras。



- TFLite优化：
 - 用Flatbuffer序列化模型文件，这种格式磁盘占用少，加载快
 - 量化。这个特性是可以开关的，可以把float参数量化为uint8类型，模型文件更小、计算更快。
 - 剪枝、结构合并和蒸馏。
 - 对NNAPI的支持。上三个特性都是转换模型文件的特性，这个是运行时的特性。也就是调用安卓底层的接口，把异构的计算能力利用起来。虽然TFLite基于NNAPI，理论上是可以利用上各种运算芯片的，但目前还没有很多运算芯片支持NNAPI。

3.2 部署过程

- 由于TFLite开源包仅支持树莓派2、3、4版本，对于树莓派zero不兼容，无法直接安装TFLite包，但树莓派的性能不足以直接使用keras。
- 最终决定在树莓派系统上安装tensorflow1.4版本，调用tensorflow里的tflite包完成部署工作。但是速度较慢。

3.2.1 初始版本

- 初始版本实现了单一拍照识别功能，摄像头进行拍摄保存，模型读取图片进行识别，最终打印结果
- 树莓派系统运行的脚本为 `recognition.py`。其中主要命令有两条，分别为执行拍照指令并保存图片和读取图片进行预测：
 - 拍照：
 - `raspistill -o img.jpg`

- 图片预测
- python classify.py

识别源码

```

import numpy as np
import tensorflow as tf
from PIL import Image
def imgToMat_RGB(img):
    img = Image.open(img)
    img = img.convert("RGB")
    #data = img.getdata()
    img = img.resize((192,192))
    mat = np.array(img)
    mat=mat/255.0
    mat=2*mat-1
    mat = mat.astype(np.float32)
    return mat

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

img = imgToMat_RGB("img.jpg")
# Test model on random input data.
input_shape = input_details[0]['shape']
l = []
l.append(img)
l = np.array(l)
#print(input_shape)
#input_data = np.array(np.random.random_sample(input_shape), dtype=np.float64)
#print(input_data)
interpreter.set_tensor(input_details[0]['index'], l)

interpreter.invoke()

# The function `get_tensor()` returns a copy of the tensor data.
# Use `tensor()` in order to get a pointer to the tensor.
output_data = interpreter.get_tensor(output_details[0]['index'])
print(output_data)

```

版本分析

不足:

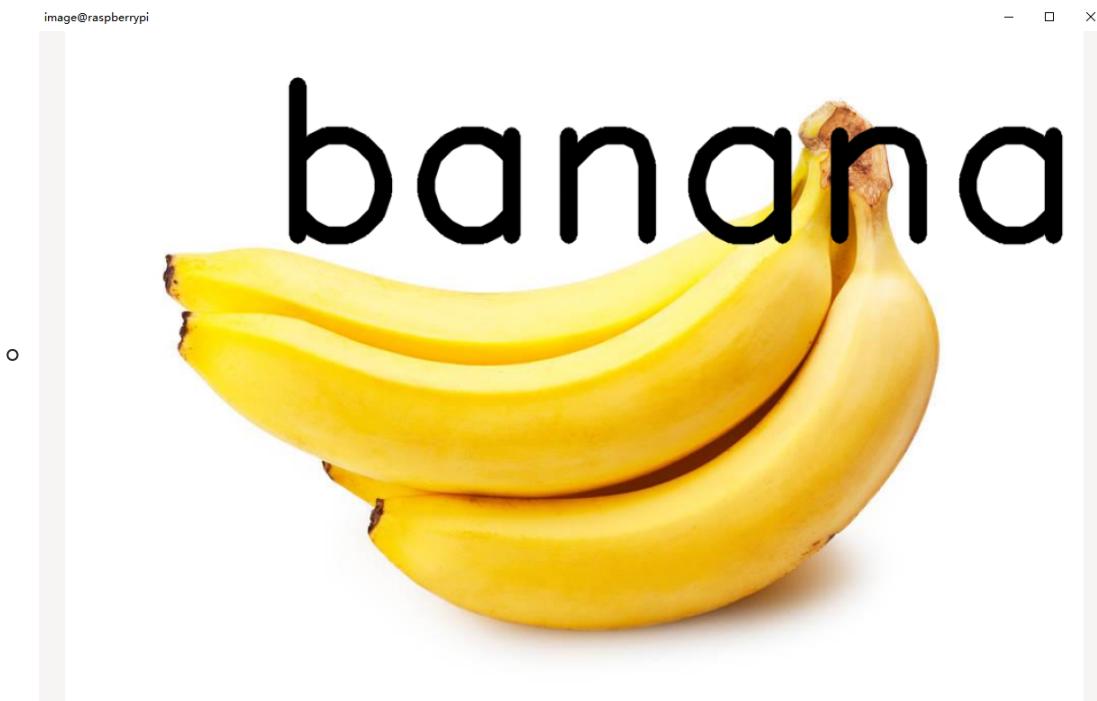
- 只能进行单一识别，每次识别都需要重新加载整个程序识别较慢，且不连续
- 最终结果通过在终端打印呈现，需要借助远程设备，且可视化较低

基于以上不足，对版本进行了以下升级。

3.2.2部署改进

改进思路

- 模型加载后，在识别过程中采用while循环进行连续识别，可实现一次加载多次识别，极大提高识别速度，且支持连续识别。
- 外接显示屏幕，将最终识别结果印在摄像头获取的图片上，最终通过显示屏对图片进行连续呈现。效果图如下：



- 每进行一次识别，终端会进行识别次数及结果的输出，如下图：

```
pi@raspberrypi:~/deep_learning $ python recognition.py
Take 1th photo...
Recognition...
banana
Take 2th photo...
Recognition...
banana
Take 3th photo...
Recognition...
banana
Take 4th photo...
Recognition...
banana
Take 5th photo...
Recognition...
banana
```

改进源码

```
import numpy as np
import tensorflow as tf
from PIL import Image
def imgToMat_RGB(img):
    img = Image.open(img)
```

```

    img = img.convert("RGB")
    #data = img.getdata()
    img = img.resize((192,192))
    mat = np.array(img)
    mat=mat/255.0
    mat=2*mat-1
    mat = mat.astype(np.float32)
    return mat

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
while(True):
    i+=1
    os.system("echo 'Take %sth photo...'" % (i))
    os.system("raspistill -t 1000 -o img.jpg")
    os.system("echo 'Recognition...'")
    img = imgToMat_RGB("img.jpg")
    # Test model on random input data.
    input_shape = input_details[0]['shape']
    l = []
    l.append(img)
    l = np.array(l)
    #print(input_shape)
    #input_data = np.array(np.random.random_sample(input_shape),
    dtype=np.float64)
    #print(input_data)
    interpreter.set_tensor(input_details[0]['index'], l)

    interpreter.invoke()
    # The function `get_tensor()` returns a copy of the tensor data.
    # Use `tensor()` in order to get a pointer to the tensor.
    output_data = interpreter.get_tensor(output_details[0]['index'])
    print(output_data)
    bk_img = cv2.imread("img.jpg")
    cv2.putText(bk_img,class_name,(200,200),cv2.FONT_HERSHEY_SIMPLEX,7,
(0,0,0),15, cv2.LINE_AA)
    cv2.namedWindow("image",0)
    cv2.resizeWindow("image",800,600)
    cv2.imshow("image",bk_img)
    cv2.waitKey()
    cv2.imwrite("img.jpg",bk_img)

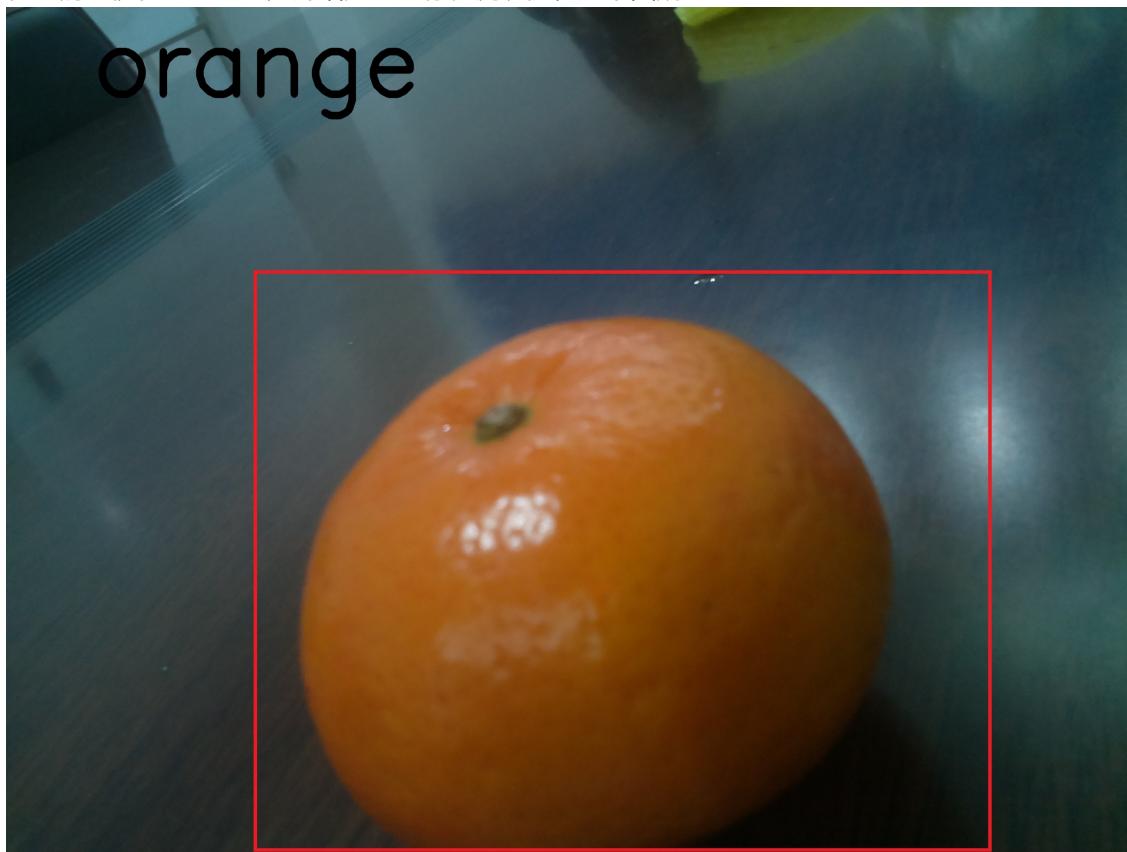
```

改进评估

- 极大提高识别速度，且实现连续识别
- 对识别结果进行可视化，不再依赖于电脑等设备，可独立工作。
- 由于树莓派zero性能过低，整个过程运行较慢。
- 经过测试，每加载一次模型大约需要30~40秒，模型加载完毕后每计算识别一张图片大约需要10秒。

3.2.3 目标检测

- 在之前可视化的基础上另外增加了目标检测功能，如下图所示：



- 最终呈现的结果图中，不仅含有类别名称，还含有目标的位置，摄像头获取到照片后，判断照片的位置，并用红色框标记，使结果更加明显。