VIVA : Installation guide (Mac)

一：环境要求

安装 Brew 参考 https://brew.sh/

$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
在 ~/.profile file 最后一行加入 export PATH=/usr/local/bin:/usr/local/sbin:$PATH

安装 Python

Brew install python

安装虚拟环境(virtual environment) 和 Tensorflow 参考 https://www.tensorflow.org/install/install_mac

```
$ sudo easy_install pip                                   # 安装pip管理器
$ pip install --upgrade virtualenv                        # 如果出现错误提示, 需要安装 nose 和 tornado
$ pip install nose
$ pip install tornado
$ virtualenv --system-site-packages ./tensorflow              # for Python 2.7
$ virtualenv --system-site-packages -p python3 ./tensorflow   # for Python 3.n
$ cd tensorflow
$ source ./bin/activate
(tensorflow)$ pip install --upgrade tensorflow                # for Python 2.7
(tensorflow)$ pip3 install --upgrade tensorflow               # for Python 3.n
```

安装 Keras 参考 https://keras.io/#installation
$ pip install keras

安装 “Keras model 存到磁盘 ” 所需的套件
$ brew install hdf5
$ pip install h5py # 若在 Keras安装过程没安装此依赖，使用此命令补充安装

安装 Pillow
$ pip install pillow

安装 XCode
到网页下载安装 XCode https://developer.apple.com/xcode/ 或者App-Store下载
$ sudo xcode-select --install
$ sudo xcodebuild -license # 滑动到底部并且接受条款

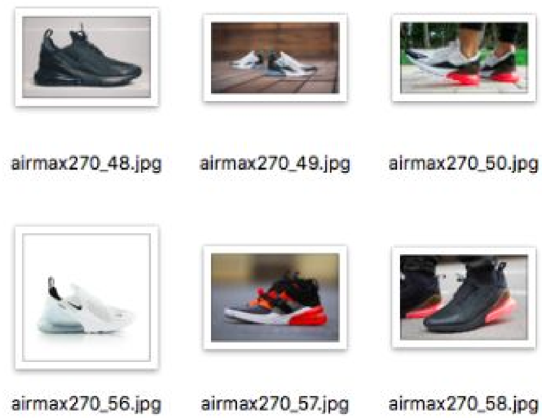
安装 OPENCV 参考：https://www.learnopencv.com/install-opencv3-on-macos/
$ brew install opencv

如果出现权限问题（在brew过程可能会出现创建brew link isl, brew link gcc, brew link hdf5的错误）
$ sudo chown -R 本机账户:admin /usr/local/bin 例如 sudo chown -R **eddieliu**:admin /usr/local/bin
$ sudo chown -R 本机账户:admin /usr/local/share 例如 sudo chown -R **eddieliu**:admin /usr/local/share

二：收集素材

图片收集准则：好的data是非常重要的一环，model的好坏，取决于训练的data的质量。

尺寸：大于300x300
张数：每个类别至少100张或以上。如果图片足够，尽量尝试500张以上，可以提高精准度。
格式：JPG



图片选择
-正面产品照
-穿/戴在人身上
-在实际世界中，有背景
-不同的角度
-不同的亮度

注意：图片类别的特征要清楚，人眼能够识别. 如果是一串小珠子，在图片的细微处，可能会造成辨识度下降。

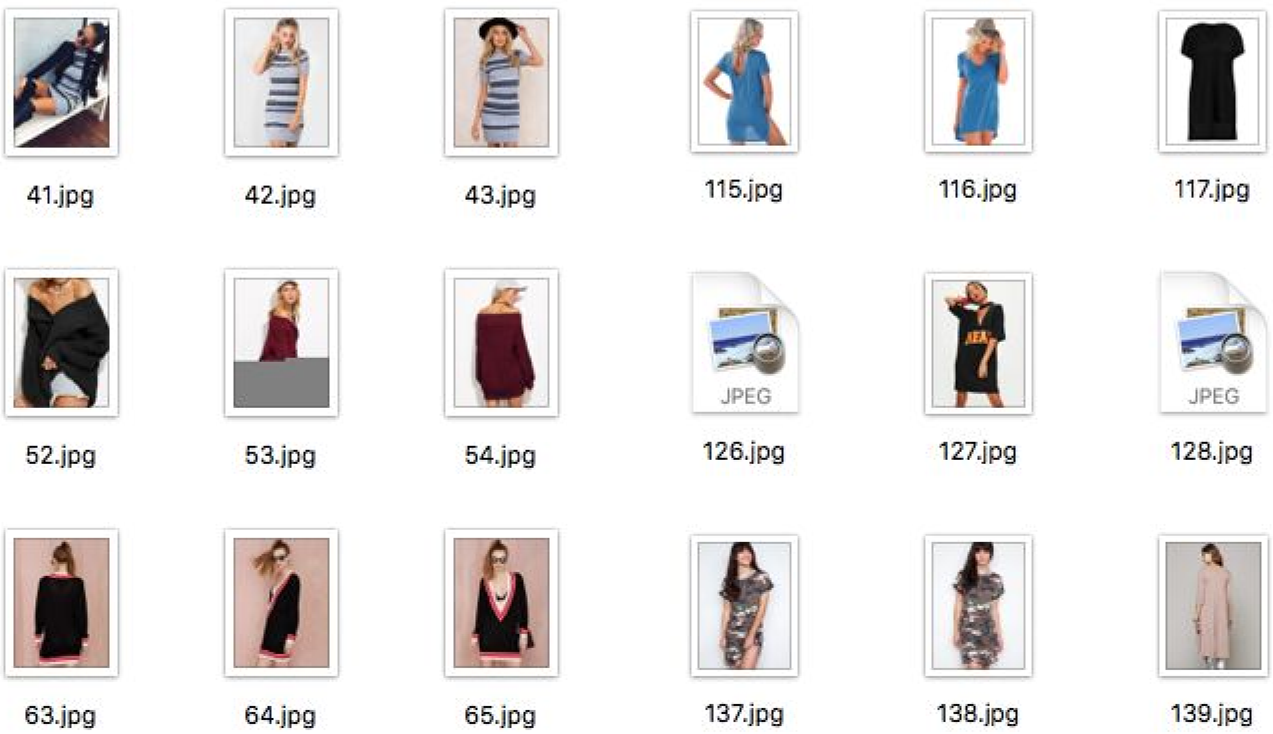
分类建议：类别要清晰，容易判断
Machine learning, 可以想像在教机器辨识各种类别分类，他会根据图片内容，自己来定义这个分类。
范例：牛仔裤

(√) 这是属于好的图片选择，因为牛仔裤占据了图片的主要内容。
想象程序会学习到，长长，蓝色，浅蓝色，上面有或没有一块不同颜色的布料，或者是下面有一双鞋子，都是属于牛仔裤。

(x) 这是比较不适合的图片选择, 因为它涵盖其他元素 e.g. 都有上半身, 头像, 鞋子，程序会认为, 必须包含这些部分才叫做 “牛仔裤”. 而这个训练过程, 可能也会和 “衣服” 的类别搞混, 特别是如果衣服类别中, 也都有全身的图片.

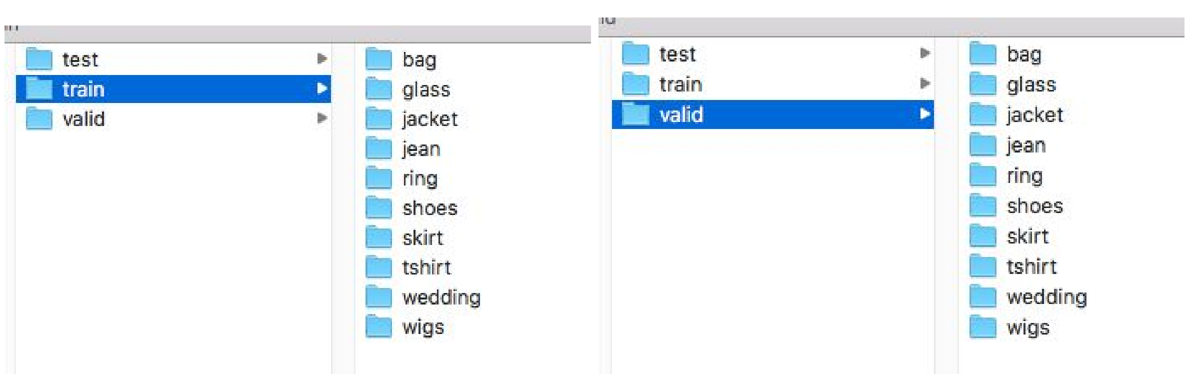
确保图片的完整性

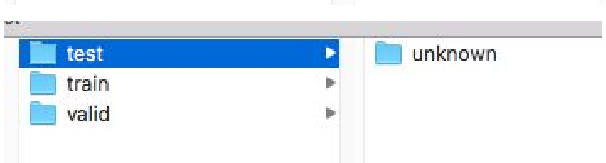
从网络上大量下载的 jpg, 很可能会出现档案有下载不完整, 必须要再 double check, 否则读取错误, 会造成程序读取错误而跳出. 以下 53.jpg, 126.jpg, 128.jpg, 都是下载不完整, 或有问题的. 务必删除, 或是重新下载该图片



文件夹结构

Train :子目录存放各种分类和图片 : 会从这个文件夹里面的图片来做训练
Valid : 子目录存放各种分类和图片 :会从这个文件夹的图片来做验证， 进一步调教 model
Test ：子目录存放 “unknown” : 最后要用来测试的图片
比例建议Train / Valid / Test : 80% /10% /10% , 也可以尝试70% / 20% / 10% 或是在之间调整





三：训练模型

1.更改 config.json 里面的参数

```
{
 "train_path" :"dataset",
 "batch_size_train":"15",
 "category_num": "3",
 "learning_rate":"0.0003",
 "epoch_num":"20",
 "steps_per_epoch":"6",
 "output_name":"model.h5"
}
```

参数	介绍	建议尝试的范围
train_path	训练图片的文件夹 注意:文件夹底下需要有Train, Valid, 和 Test 的文件夹	文件夹的路径
batch_size_train	每个批次训练几张图片	10 ~ 30
category_num	训练几种类别	训练几种类别
learning_rate	学习变化的幅度	0.001 ~ 0.00001
epoch_num	总共训练几轮，在每一轮会更新model的weight	100 ~ 200
steps_per_epoch	每轮会训练几个批次	6~10
output_name	输出档案的名字	xxxxx.h5

train model 需要的时间, 与要处理的图片数目batch size, steps per epoch, 和 epoch, 呈线性成长。
-每个 epoch 会处理的图片数目 = **batch_size_train * steps_per_epoch**
-总共需要 training 的时间 ：epoche_num * 每个 epoch 的时间

epoch_num	batch_size_train	steps_per_epoch	总共时间
10	10	10	1 T
10	20	10	2 T
10	10	20	2 T
10	15	20	3 T
80	15	20	24 T

2.命令行：python train_model.py

```
#!/usr/bin/env python
# -*- coding: utf-8 *

import numpy as np
import keras
from keras import backend as K
from keras.layers.core import Dense, Activation
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.models import Model
from keras.applications import imagenet_utils
import json

# Open config.json
def jsonReader():
   with open("./config.json", 'r') as load_f:
       jsonDict = json.load(load_f)
       return jsonDict

# Read data from config.json
jsonData = jsonReader()
train_path = jsonData['train_path'] + '/train'
valid_path = jsonData['train_path'] + '/valid'
test_path = jsonData['train_path'] + '/test'
bsize = int(jsonData['batch_size_train'])
catnum = int(jsonData['category_num'])
lrate = float(jsonData['learning_rate'])
steps = int(jsonData['steps_per_epoch'])
epoch_num = int(jsonData['epoch_num'])
output_name = jsonData['output_name']

# Train the model. Default shuffle = true
train_batches =
ImageDataGenerator(preprocessing_function=keras.applications.mobilenet.preproces
s_input).flow_from_directory( train_path, target_size=(224,224),
batch_size=bsize)

# Validate the model.  Default shuffle = true
valid_batches =
ImageDataGenerator(preprocessing_function=keras.applications.mobilenet.preproces
s_input).flow_from_directory( valid_path, target_size=(224,224), batch_size=10)

# Make prediction with the model trained
test_batches =
ImageDataGenerator(preprocessing_function=keras.applications.mobilenet.preproces
```

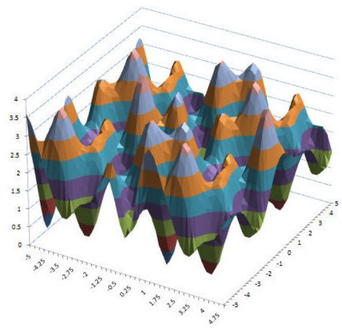
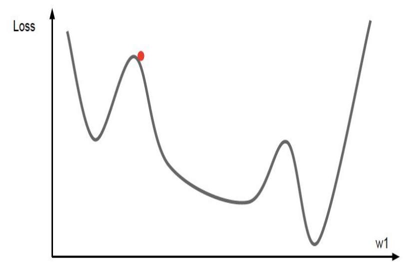
```
s_input).flow_from_directory(test_path, target_size=(224,224), batch_size=2,
shuffle=False)

mobile = keras.applications.mobilenet.MobileNet()

x = mobile.layers[-6].output
predictions = Dense(catnum, activation='softmax')(x)
model = Model(inputs=mobile.input, outputs=predictions)

model.summary()

# Only train the last 5 layers and make the previous layers fixed
for layer in model.layers[:-5]:
   layer.trainable = False
model.compile(Adam(lr=lrate), loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit_generator(train_batches, steps_per_epoch=steps,
                   validation_data=valid_batches, validation_steps=2,
epochs=epoch_num, verbose=2)

# Output the model
model.save(output_name)

# Make prediction with the model
predictions = model.predict_generator(test_batches,steps=1, verbose=2)
print(predictions)
print(train_batches.class_indices)
```

四：调教模型

本章节主要会描述各种 train model 的过程, 可能会遇到的一些状况, 以及如何调整参数, 来去做调教.

在训练的过程中，要注意 loss (模型与实际结果的差别) 和 valid_acc (valid 集合里面的精准度)

想像一下, 我们要找到整个山脉群里面, 最低的地方 (最小的 loss), 而 learning rate 是每次移动的步伐

假设最佳的山谷低点, 在前方 5.25 km 处, 要如何去探索呢?

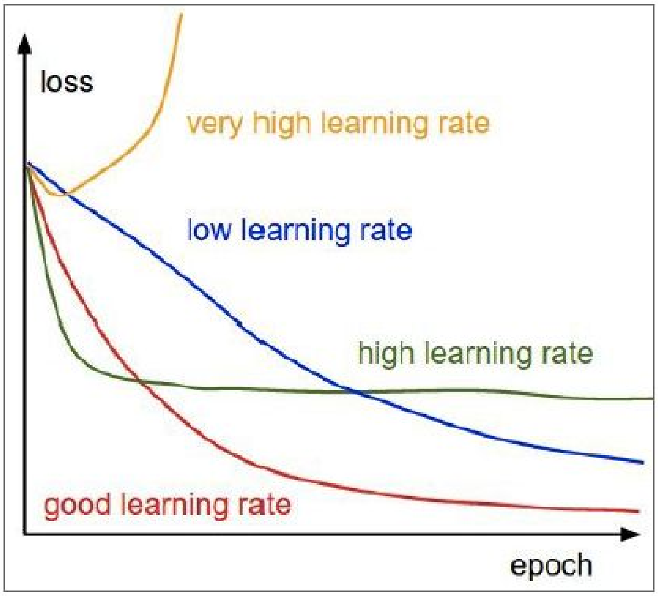
Very high learning rate : 每一步都是 100 km, 跨一步, 都会离目标更远

High learning rate : 每一步都是 1 km, 前期 loss rate 下降快速, 但会有瓶颈，最后在 5 ~ 6 km 之间探索。

Low learning rate : 每一步都是 0.001 km, loss rate下降非常地缓慢。

Good learning rate : 每一步 0.1 km, **前期 loss rate 下降快速，最后趋于平稳**, 在 5.2 ~ 5.3 km 附近探索

注：每次训练的内容和顺序都是随机选择的，但是会按照所给的config.json的要求去进行，所以即使初始配置一致，也会出现不同的训练反馈。



让我们来看选择不同数值的训练过程

Very high learning rate : 10
Loss 马上飚高到超过 15 且无法收敛, 可以停止训练了
Epoch 1/100 - 51s - loss: 14.0048 - acc: 0.0537 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 2/100 - 46s - loss: 14.9361 - acc: 0.0733 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 3/100 - 43s - loss: 15.3122 - acc: 0.0500 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 4/100 - 46s - loss: 14.8824 - acc: 0.0767 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 5/100 - 43s - loss: 14.8824 - acc: 0.0767 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 6/100 - 41s - loss: 14.7749 - acc: 0.0833 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 7/100 - 44s - loss: 14.9898 - acc: 0.0700 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 8/100 - 42s - loss: 14.7212 - acc: 0.0867 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 9/100 - 42s - loss: 15.0436 - acc: 0.0667 - val_loss: 16.1181 - val_acc: 0.0000e+00 Epoch 10/100 - 42s - loss: 14.8824 - acc: 0.0767 - val_loss: 16.1181 - val_acc: 0.0000e+00
下一步：大幅降低 learning rate

High learning rate : 0.01	
前 10 步loss 快速下降, 从 2.7 到 0.5 和 7.1 到 2.1 accuracy 也快速到达 0.7~0.8 和 0.3~0.4	Train 到后面, valid accuracy 到达了瓶颈 0.4~0.5. 注意: 这边 train的 loss 小于 0.1, 且 train accuracy 接近 1.0, 表示model过分贴合训练的模型, 反而造成预测能力下降。（此时的model比较贴合于理想环境，对新数据的内容识别能力下降）

<table>
<tr><td>Epoch 1/100
- 42s - loss: 2.7851 - acc: 0.1833 - val_loss: 7.1870 - val_acc: 0.0000e+00
Epoch 2/100
- 38s - loss: 2.1982 - acc: 0.3008 - val_loss: 5.9909 - val_acc: 0.2000
Epoch 3/100
- 42s - loss: 2.0329 - acc: 0.3667 - val_loss: 3.1621 - val_acc: 0.4000
Epoch 4/100
- 41s - loss: 1.6221 - acc: 0.4867 - val_loss: 3.8205 - val_acc: 0.2000
Epoch 5/100
- 39s - loss: 1.5566 - acc: 0.4600 - val_loss: 6.1969 - val_acc: 0.0000e+00
Epoch 6/100
- 37s - loss: 1.4096 - acc: 0.5400 - val_loss: 5.0262 - val_acc: 0.2000
Epoch 7/100
- 37s - loss: 0.7210 - acc: 0.7500 - val_loss: 5.4693 - val_acc: 0.0000e+00
Epoch 8/100
- 38s - loss: 0.6067 - acc: 0.8333 - val_loss: 3.3162 - val_acc: 0.2000
Epoch 9/100
- 37s - loss: 0.4823 - acc: 0.8767 - val_loss: 2.9067 - val_acc: 0.4000
Epoch 10/100
- 36s - loss: 0.5013 - acc: 0.8733 - val_loss: 2.1334 - val_acc: 0.3000</td>
<td>Epoch 81/100
- 38s - loss: 0.0121 - acc: 1.0000 - val_loss: 2.5431 - val_acc: 0.4000
Epoch 82/100
- 47s - loss: 0.0113 - acc: 1.0000 - val_loss: 2.3135 - val_acc: 0.3000
Epoch 83/100
- 44s - loss: 0.0086 - acc: 1.0000 - val_loss: 2.4654 - val_acc: 0.4000
Epoch 84/100
- 45s - loss: 0.0166 - acc: 0.9967 - val_loss: 2.8840 - val_acc: 0.5000
Epoch 85/100
- 39s - loss: 0.0105 - acc: 1.0000 - val_loss: 2.6925 - val_acc: 0.5000
Epoch 86/100
- 38s - loss: 0.0083 - acc: 1.0000 - val_loss: 2.1870 - val_acc: 0.4000
Epoch 87/100
- 39s - loss: 0.0071 - acc: 1.0000 - val_loss: 2.2050 - val_acc: 0.4000
Epoch 88/100
- 37s - loss: 0.0080 - acc: 1.0000 - val_loss: 2.2907 - val_acc: 0.4000
Epoch 89/100
- 39s - loss: 0.0068 - acc: 1.0000 - val_loss: 2.2843 - val_acc: 0.4000</td></tr>
<tr><td colspan="2">下一步：稍微降低 learning rate</td></tr>
</table>

<table>
<tr><th colspan="2">Low learning rate : 0.00001</th></tr>
<tr><td>最初10步骤, loss从 3.26 缓慢下降到2.85</td><td>最后10步骤, loss 都大于1, accuracy 停留在0.6 / 0.3</td></tr>
<tr><td>Epoch 1/100
- 41s - loss: 3.2658 - acc: 0.0867 - val_loss: 2.9313 - val_acc: 0.2000
Epoch 2/100
- 39s - loss: 3.3687 - acc: 0.0767 - val_loss: 2.9106 - val_acc: 0.2000
Epoch 3/100
- 42s - loss: 3.1668 - acc: 0.0667 - val_loss: 2.8740 - val_acc: 0.2000
Epoch 4/100
- 43s - loss: 3.1335 - acc: 0.0900 - val_loss: 2.8356 - val_acc: 0.2000
Epoch 5/100
- 42s - loss: 3.1657 - acc: 0.0800 - val_loss: 2.8029 - val_acc: 0.2000
Epoch 6/100
- 38s - loss: 3.1580 - acc: 0.0872 - val_loss: 2.7779 - val_acc: 0.2000
Epoch 7/100
- 43s - loss: 2.9513 - acc: 0.0667 - val_loss: 2.7578 - val_acc: 0.2000
Epoch 8/100
- 44s - loss: 2.9811 - acc: 0.0900 - val_loss: 2.7353 - val_acc: 0.2000
Epoch 9/100
- 42s - loss: 2.9064 - acc: 0.1100 - val_loss: 2.7239 - val_acc: 0.2000
Epoch 10/100
- 41s - loss: 2.8591 - acc: 0.1133 - val_loss: 2.7067 - val_acc: 0.2000</td>
<td>Epoch 91/100
- 37s - loss: 1.4415 - acc: 0.5930 - val_loss: 1.9144 - val_acc: 0.3000
Epoch 92/100
- 40s - loss: 1.4591 - acc: 0.6167 - val_loss: 1.9073 - val_acc: 0.3000
Epoch 93/100
- 43s - loss: 1.4684 - acc: 0.5867 - val_loss: 1.9024 - val_acc: 0.3000
Epoch 94/100
- 42s - loss: 1.4227 - acc: 0.5800 - val_loss: 1.9028 - val_acc: 0.3000
Epoch 95/100
- 43s - loss: 1.4395 - acc: 0.5800 - val_loss: 1.9013 - val_acc: 0.3000
Epoch 96/100
- 41s - loss: 1.3721 - acc: 0.6267 - val_loss: 1.8974 - val_acc: 0.3000
Epoch 97/100
- 42s - loss: 1.4560 - acc: 0.6033 - val_loss: 1.8928 - val_acc: 0.3000
Epoch 98/100
- 43s - loss: 1.4318 - acc: 0.5700 - val_loss: 1.8878 - val_acc: 0.3000
Epoch 99/100
- 42s - loss: 1.4992 - acc: 0.5982 - val_loss: 1.8804 - val_acc: 0.3000
Epoch 100/100
- 37s - loss: 1.3924 - acc: 0.6100 - val_loss: 1.8779 - val_acc: 0.3000</td></tr>
<tr><td colspan="2">下一步：稍微提升 learning rate</td></tr>
</table>

<table>
<tr><th colspan="2">Good learning rate : 0.00005</th></tr>
<tr><td>Good learning rate : 0.00005
最初10步骤, loss从 3下降到1.8</td><td>最后10步骤, loss 和 accuracy 都趋于稳定.
loss 在 0.3~0.4 / 1.2~1.3; accuracy 在0.95 / 0.7~0.8</td></tr>
</table>

Epoch 1/100 - 39s - loss: 3.0220 - acc: 0.0967 - val_loss: 2.5357 - val_acc: 0.2000 Epoch 2/100 - 36s - loss: 2.8572 - acc: 0.1233 - val_loss: 2.5735 - val_acc: 0.2000 Epoch 3/100 - 36s - loss: 2.6347 - acc: 0.1533 - val_loss: 2.5309 - val_acc: 0.2000 Epoch 4/100 - 36s - loss: 2.4472 - acc: 0.2000 - val_loss: 2.5096 - val_acc: 0.2000 Epoch 5/100 - 37s - loss: 2.4605 - acc: 0.1667 - val_loss: 2.4614 - val_acc: 0.2000 Epoch 6/100 - 45s - loss: 2.3777 - acc: 0.2151 - val_loss: 2.3947 - val_acc: 0.3000 Epoch 7/100 - 37s - loss: 2.1120 - acc: 0.3127 - val_loss: 2.3821 - val_acc: 0.2000 Epoch 8/100 - 42s - loss: 2.0617 - acc: 0.3500 - val_loss: 2.3293 - val_acc: 0.3000 Epoch 9/100 - 43s - loss: 1.9819 - acc: 0.3667 - val_loss: 2.3203 - val_acc: 0.4000 Epoch 10/100 - 1136s - loss: 1.8970 - acc: 0.4033 - val_loss: 2.2391 - val_acc: 0.3000	Epoch 91/100 - 37s - loss: 0.4574 - acc: 0.9426 - val_loss: 1.2659 - val_acc: 0.7000 Epoch 92/100 - 43s - loss: 0.3990 - acc: 0.9633 - val_loss: 1.2293 - val_acc: 0.8000 Epoch 93/100 - 42s - loss: 0.4320 - acc: 0.9600 - val_loss: 1.2142 - val_acc: 0.8000 Epoch 94/100 - 42s - loss: 0.3907 - acc: 0.9667 - val_loss: 1.2125 - val_acc: 0.8000 Epoch 95/100 - 39s - loss: 0.3538 - acc: 0.9767 - val_loss: 1.2097 - val_acc: 0.8000 Epoch 96/100 - 42s - loss: 0.4138 - acc: 0.9600 - val_loss: 1.2195 - val_acc: 0.7000 Epoch 97/100 - 43s - loss: 0.4060 - acc: 0.9598 - val_loss: 1.2310 - val_acc: 0.7000 Epoch 98/100 - 47s - loss: 0.3656 - acc: 0.9700 - val_loss: 1.2328 - val_acc: 0.7000 Epoch 99/100 - 42s - loss: 0.3816 - acc: 0.9667 - val_loss: 1.2316 - val_acc: 0.8000 Epoch 100/100 - 43s - loss: 0.3710 - acc: 0.9733 - val_loss: 1.2435 - val_acc: 0.7000
下一步：这是适合的 learning rate,仅作微调, 看是否让 accuracy 能够更高	

五：TFJS converter (Keras h5 转 TFJS)

参考： https://github.com/tensorflow/tfjs-converter

安装 Tensorflowjs

pip install tensorflowjs

转换format

```
tensorflowjs_converter \
    --input_format=keras \
    /tmp/my_keras_model.h5 \
    /tmp/my_tfjs_model
```

六：接入前端网页应用

1. 上传转换过后的模型到服务器上，并记下 **model.json** 的位置
2. **创建 MobileNet 物件**，物件内包含加载模型，进行预测及取回预测结果进行商务逻辑

```
var MobileNet = {
  /* TODO: 加载 model 的 url */
  MODEL_URL: 'https://your.model.url',
  /* 分辨率为固定值 请勿修改*/
  PREPROCESS_DIVISOR: 127.5,
  /* 预测几率在多少以上当成成功*/
  PICKUP_RATE: 0.7,
  constructor: function (){},
  /* TODO(1): 加入是否加载过模型 */

  /* TODO(2): 加入判断网络环境 */

  /* 加载 model 的 function */
  load: async function (){
    this.model = await tf.loadModel(this.MODEL_URL);
    /* 加载完 model 后进行第一次模拟，可加速用户第一次辨识的速度 */
    this.predict(tf.zeros([1, 224, 224, 3])).dispose();
    /* 加载完后绑定开启相机事件 *可自行加入商务条件 */
    camera_ops.setup('detector', 'video', 'closeCamera', 'outputCanvas',
'camera');
  },
  /* 预测处置，清空模型内预测后的缓存 */
  dispose: function () {
    this.model.dispose();
  },
  /* 预测 function，先将图片正规化后进行预测 */
  predict: function (screenshot) {
    console.time();
    const offset = tf.scalar(this.PREPROCESS_DIVISOR);
    const nomalized = screenshot.sub(offset).div(offset);
    const batched = nomalized.reshape([1,224,224,3]);

    return this.model.predict(batched);
  },
  /* 传入预测结果物件 ,取出预测结果数值进行商务逻辑 */
  getTopResult: async function (logits) {
    /* 取出结果数值*/
    const values = await logits.data();
```

```
    /* 将结果传入阵列*/
    const valuesAndIndices = [];
    for (let i = 0; i < values.length; i++) {
      valuesAndIndices.push({value: values[i], index: i});
    }
    /*进行排序*/
    valuesAndIndices.sort((a, b) => {
      return b.value - a.value;
    });

    /* TODO：使用结果阵列引入商务逻辑*/

    console.log(valuesAndIndices);
    console.timeEnd();

  }
}
```

3. **使用 PWA 功能检查用户当下条件是否符合加载模型**
   a. **是否支持 Service Worker**

```
if ('serviceWorker' in navigator) {
        window.addEventListener('load', function() {
          navigator.serviceWorker.register('/sw.js').then(function(registration) {
            // Registration was successful
            console.log('ServiceWorker registration successful with scope: ',
registration.scope);
          }, function(err) {
            // registration failed :(
            console.log('ServiceWorker registration failed: ', err);
          });
          });
        }
```

   b. **是否已经加载过 model：TODO(1)**

```
check_if_model_exist: function () {
    caches.has(‘https://your.model.url’).then(function (result){
      return result;
    });
  }
```

   c. **未加载过 model 是否当下在 wifi 的环境：TODO(2)**

```
check_isWifi: function() {
    if (!navigator.connection || navigator.connection.type != "wifi") {
      return false;
    }
```

```
    return true;
  }
```

结合后会变成

```
if(!MobileNet.check_if_model_exist()) {
  if (MobileNet.check_isWifi()){
    MobileNet.load();
  }
} else {
  MobileNet.load();
}
```

4. **在 HTML 上引入Tensorflow JS库**

```
<script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@0.12.0/dist/tf.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-converter@0.5.0/dist/tf-converter.min.js"></script>
```

5. **在 HTML body 里面加上相机功能和开启相机后的图层**

```
<div id="cameraLayer" style="display:none;">
  <i class="material-icons" id="closeButton" style="display:none;">close</i>
  <video id="video" playinline="playinline" style="display:none;"></video>
</div>
<canvas id="middleCanvas" style="display:none;"></canvas>
<img id="imageToPredict" style="display:none;" src="" width="224" height="224"
/>
<div id="loadingLayer"><img id="loading" src="/images/loading.gif" /></div>
```

6. **绑定相机以触发预测**

```
/* 设置相机物件以触发图形预测 */
var camera_ops = {
  IDENTIFY_WAIT_TIME: 1500, //开启相机后截取图片等待时间
  cameraLayer: undefined, // 相机图层
  video: undefined, // 相机视频 DOM
  closeButton: undefined, // 关闭相机按钮
  middleCanvas: undefined, // 截取视频转换为图片的中间层
  mainContent: undefined, // 相机图层欲覆盖的主要图层
  videoStream: undefined, // 开启相机后的视频串流
```

```
  loadingLayer: undefined, // 进行预测是的中间加载效果
  /* 绑定相机事件 */
  setupCamera: function(cameraLayerId, videoId, closeButtonId, middleCanvasId,
cameraTriggerId) {
    this.cameraLayer = document.getElementById(cameraLayerId);
    this.video = document.getElementById(videoId);
    this.closeButton = document.getElementById(closeButtonId);
    this.middleCanvas = document.getElementById(middleCanvasId);
    // TODO: 更改加载图层ID
    this.loadingLayer = document.getElementById('loadingLayer')
    // TODO：更改图层内容主要图层 Class
    this.mainContent = document.getElementsByClassName('yourMainContentClass');
    //setup camera trigger
    this.addCameraEvent(cameraTriggerId);
    this.bindCameraLayerClose();
  },
  /* 检查浏览器是否支持相机功能 */
  checkAvailable: function () {
    if (!navigator.mediaDevices || !navigator.mediaDevices.getUserMedia) {
      return false;
    }
    return true;
  },
  /* 为相机按钮绑上开启相机事件 */
  addCameraEvent: function (id) {
    const w = screen.width;
    const h = screen.width;
    console.log('width:' + w + ", h:" + h);
    const cameraButton = document.getElementById(id);

    cameraButton.addEventListener('click', async function () {
      /* OPTIOAL TODO：可加入其他条件是否继续绑定开启相机事件 */

      /* 检查浏览器是否支持相机功能 */
      if (!camera_ops.checkAvailable()) {
        general_ops.showMessage('You browser not support browser camera api');
        return;
      }

      camera_ops.video.width = w;
      camera_ops.video.height = h;
      /* 设置相机基本参数，参数内容可更动，参数内容请参考：
https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamConstraints */
```

```
    camera_ops.videoStream = await navigator.mediaDevices.getUserMedia({
      audio: false,
      video :{
        facingMode: 'environment',
        width: { min: 224, ideal: 448, max: 1344 },
        height: { min: 224, ideal: 448, max: 1344 }
      }
    });
    /* 将串流导入视频物件*/
    camera_ops.video.srcObject = camera_ops.videoStream;

    return (new Promise((resolve) => {
        /* 确认视频串流正确载入后的动作 */
        camera_ops.video.onloadedmetadata = () => {
          camera_ops.mainContent[0].style = 'display: none';
          camera_ops.cameraLayer.style.display = 'block';
          camera_ops.video.play();
          camera_ops.video.style.display = 'block';
          camera_ops.closeButton.style.display = 'block';
          console.time();
          resolve('init');

        };
      })).then(function() {
        /* 等待时间截取目前用户的相机画面 */
        setTimeout(function () {
          camera_ops.loadingLayer.classList.add('show');
          this.video.pause();
          camera_ops.snapshot(); },
          camera_ops.IDENTIFY_WAIT_TIME);

      });
  })

  cameraButton.classList.add('fadeIn');
},
/* 绑定关闭相机图层 */
bindCameraLayerClose: function () {
  this.closeButton.addEventListener('click', function () {
    camera_ops.mainContent[0].style.display = 'block';
    camera_ops.cameraLayer.style.display = 'none';
    camera_ops.video.pause();
    camera_ops.videoStream.getTracks()[0].stop();
```

```
      camera_ops.video.style.display = 'none';
      camera_ops.closeButton.style.display = 'none';
      camera_ops.loadingLayer.className = '';
    });
  },
  /* 截取图片到 Canvas 后转成图片物件导入模型进行预测 */
  snapshot: function () {
    console.timeEnd();
    const imageToPredict = document.getElementById('imageToPredict');
    this.middleCanvas.width = camera_ops.video.width;
    this.middleCanvas.height = camera_ops.video.height;
    let canvasContext = this.middleCanvas.getContext('2d');
    canvasContext.drawImage(camera_ops.video, 0, 0, camera_ops.video.width,
camera_ops.video.height);
    imageToPredict.src = this.middleCanvas.toDataURL('image/jpeg');
    imageToPredict.width = '224';
    imageToPredict.height = '224';
    imageToPredict.onload = function ()
{predict_ops.runPredict(imageToPredict);}
  }
};
```

7. **链接相机和 Mobilenet 模组**

```
/* 链接相机和 Mobilenet 模组*/
var predict_ops = {
  runPredict: function (input) {
    /* 转成 tensorflow 可用的图片格式 */
    const pixels = tf.fromPixels(input).toFloat();
    let result = MobileNet.predict(pixels);
    MobileNet.getTopResult(result);
  }
}
```

The document is credit to gTech Velocity MSC CN team (Eddie Liu, Palances Liao, Keith Gu, Cecilia Cong, Victor Shen), and advices from CN TensorFlow team (Tiezhen Wang)