Tensorflow知识点杂记

1. tensorflow lite

|  |
| --- |
| TensorFlow Lite 是 Google I/O 2017 大会上的其中一个重要宣布，有了TensorFlow Lite，应用开发者可以在移动设备上部署人工智能。  Google 表示 Lite 版本 TensorFlow 是 TensorFlow Mobile 的一个延伸版本。尽管是一个轻量级版本，依然是在智能手机和嵌入式设备上部署深度学习的一大动作。此前，通过TensorFlow Mobile API，TensorFlow已经支持手机上的模型嵌入式部署。TensorFlow Lite应该被视为TensorFlow Mobile的升级版。  TensorFlow Lite 目前仍处于“积极开发”状态，目前仅有少量预训练AI模型面世，比如MobileNet、用于计算机视觉物体识别的Inception v3、用于自然语言处理的Smart Reply，当然，TensorFlow Lite上也可以部署用自己的数据集定制化训练的模型。 |

1. Tensorflow 有几种模型格式

|  |
| --- |
| **1. CheckPoint(\*.ckpt)**  在训练 TensorFlow 模型时，每迭代若干轮需要保存一次权值到磁盘，称为“checkpoint”，如下图所示：  https://ask.qcloudimg.com/http-save/yehe-1008345/67vg6y19zs.jpeg?imageView2/2/w/1620  这种格式文件是由 tf.train.Saver() 对象调用 saver.save() 生成的，只包含若干 Variables 对象序列化后的数据，不包含图结构，所以只给 checkpoint 模型不提供代码是无法重新构建计算图的。  载入 checkpoint 时，调用 saver.restore(session, checkpoint\_path)。  **2. GraphDef(\*.pb)**  这种格式文件包含 protobuf 对象序列化后的数据，包含了计算图，可以从中得到所有运算符（operators）的细节，也包含张量（tensors）和 Variables 定义，但不包含 Variable 的值，因此只能从中恢复计算图，但一些训练的权值仍需要从 checkpoint 中恢复。下面代码实现了利用 \*.pb 文件构建计算图：  https://ask.qcloudimg.com/http-save/yehe-1008345/sa43no37sq.jpeg?imageView2/2/w/1620  TensorFlow 一些例程中用到 \*.pb 文件作为预训练模型，这和上面 GraphDef 格式稍有不同，属于冻结（Frozen）后的 GraphDef 文件，简称 FrozenGraphDef 格式。这种文件格式不包含 Variables 节点。将 GraphDef 中所有 Variable 节点转换为常量（其值从 checkpoint 获取），就变为 FrozenGraphDef 格式。代码可以参考 tensorflow/python/tools/freeze\_graph.py  \*.pb 为二进制文件，实际上 protobuf 也支持文本格式（\*.pbtxt），但包含权值时文本格式会占用大量磁盘空间，一般不用。  **3. SavedModel**  在使用 TensorFlow Serving 时，会用到这种格式的模型。该格式为 GraphDef 和 CheckPoint 的结合体，另外还有标记模型输入和输出参数的 SignatureDef。从 SavedModel 中可以提取 GraphDef 和 CheckPoint 对象。  SavedModel 目录结构如下：  https://ask.qcloudimg.com/http-save/yehe-1008345/r3zrd8uguz.jpeg?imageView2/2/w/1620  其中 saved\_model.pb（或 saved\_model.pbtxt）包含使用 MetaGraphDef protobuf 对象定义的计算图；assets 包含附加文件；variables 目录包含 tf.train.Saver() 对象调用 save() API 生成的文件。  以下代码实现了保存 SavedModel：  https://ask.qcloudimg.com/http-save/yehe-1008345/m1zqk8yu1e.jpeg?imageView2/2/w/1620  载入 SavedModel：  更多细节可以参考 tensorflow/python/saved\_model/README.md。  **4. 小结**  本文总结了 TensorFlow 常见模型格式和载入、保存方法。部署在线服务（Serving）时官方推荐使用 SavedModel 格式，而部署到手机等移动端的模型一般使用 FrozenGraphDef 格式（最近推出的 TensorFlow Lite 也有专门的轻量级模型格式 \*.lite，和 FrozenGraphDef 十分类似）。这些格式之间关系密切，可以使用 TensorFlow 提供的 API 来互相转换 |

1. tensorflow项目说明
   1. io：
      1. 是一个与i/o相关的模块，可以通过pip进行按装
   2. tensorflow:
      1. 核心功能
   3. tfx:
      1. tensorflow基础上再次开发，是一个训练平台可以直接使用，是为了帮助开发人员快速入门
   4. datasets：
      1. 谷哥提供的开放的训练数据
   5. addons：
      1. tensorflow越来越大，更新也越来越快，会产生很多过时的或自定义功能的一些代码，这些都没有必要在tensorflow核心项目中保留，因此把这些全部单出提出来成立了这个扩展插件。
   6. custom-op：
      1. tensorflow核心概念之op

TF中的op代表了对“操作”的抽象，说它抽象是因为，op仅仅说明了操作是做什么用的，但没有说明具体怎么做。举个例子，MatMul是一个操作，它表示了矩阵乘法，但并不包含矩阵乘法的具体实现，因为我们知道，在CPU和GPU上，矩阵乘法的高效率实现是完全不同的。为了把不同设备上的实现细节隐藏起来，为相同的计算提供统一的对外表示，TF提出了op的概念。

为了方便序列化，TF中很多核心概念定义在proto文件中，操作的定义OpDef放在op.proto（/tensorflow/core/framework/op.proto）文件中。因此，OpDef仅包含一些静态的数值信息，比如操作的名称，输入、输出类型，参数，描述，以及一些计算相关的属性信息（是否可交换、是否可聚集、是否有状态）。随着对TF剖析的深入，我们会发现核心概念往往都有xx\_def定义，比如graph\_def，node\_def，kernel\_def等。

仅有静态信息是不够的，我们在构建一个op的时候，还需要灵活的对op\_def中的属性进行设置，于是就有了对OpDefBuilder类的需求。后续我们还会看到针对其它核心概念的构建类，比如graph\_builder，node\_builder等等。

有了OpDefBuilder类之后，我们构建一个op就比较简单了，首先创建一个OpDefBuilder的对象，然后依次调用该对象的属性set函数，直到完成构建。这个过程对于单个op来说相当轻松，但是当我们需要构建上百个op时，这个任务就会变得非常繁琐。一个想法是，如果能把OpDefBuilder对象属性设置的函数串联起来，形成一个chain，这个过程就会方便很多，于是就诞生了OpDefBuilderWrapper，它仅仅是OpDefBuilder的一个封装，使属性值设置的行为可以串联进行，简化了操作定义。

TF的ops目录下存放了上百个op定义，但并不是每一个TF应用都需要所有的定义。为了为TF底层库进行减负，我们需要把不需要的op省略掉，为了对此作出支持，OpDefBuilderWrapper被设计成为模板类，我们可以通过宏定义选择哪些操作被编译进TF，哪些可以被省略。

TF官方定义了大量的op，用户本身也会有大量的自定义op，为了对op进行集中管理，TF提供了OpRegistryInterface接口，它只包含一个函数，Lookup，通过操作名称查找OpDef。具体由OpRegistry和OpListRegistry两个类来实现，我们主要关注前者。OpRegistry内部包含了一个op名称到op\_def的映射，在注册的时候，采取了延迟注册的机制。对于申请注册的OpDef，先放入一个deferred\_的向量，在下次需要对已注册OpDef进行查找之前，将deferred\_中的操作注册，并清空deferred\_，这样既能保证所有的操作都被注册，也能保证注册过程的高效。

* + 1. 作用

主要是为了方便用户自定义

* 1. tfjs-models
     1. 用途说明

在js中可以直接调用相关模块，这里面有一些提前训练好的模型

* 1. model-optimization
  2. examples
  3. hub
  4. magenta
  5. metadata
  6. kfac
  7. Workshops
  8. Magenta-demos
  9. 训练记录

|  |
| --- |
| 1. Tensorflow-hub中image\_retraining/retrain.py   这个代码一直报错  urllib.error.URLError: <urlopen error [Errno 65] No route to host>  本来想手动下载，并且在环境变量中填加了变量  #=======tensorflow-hub缓存路径=========  export TFHUB\_CACHE\_DIR=./my\_module\_cache  但还是没用，只能重新找一个训练代码：  # Copyright 2015 The TensorFlow Authors. All Rights Reserved. # # Licensed under the Apache License, Version 2.0 (the "License"); # you may not use this file except in compliance with the License. # You may obtain a copy of the License at # # http://www.apache.org/licenses/LICENSE-2.0 # # Unless required by applicable law or agreed to in writing, software # distributed under the License is distributed on an "AS IS" BASIS, # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. # See the License for the specific language governing permissions and # limitations under the License. # ============================================================================== *r"""Simple transfer learning with Inception v3 or Mobilenet models.  With support for TensorBoard.  This example shows how to take a Inception v3 or Mobilenet model trained on ImageNet images, and train a new top layer that can recognize other classes of images.  The top layer receives as input a 2048-dimensional vector (1001-dimensional for Mobilenet) for each image. We train a softmax layer on top of this representation. Assuming the softmax layer contains N labels, this corresponds to learning N + 2048\*N (or 1001\*N) model parameters corresponding to the learned biases and weights.  Here's an example, which assumes you have a folder containing class-named subfolders, each full of images for each label. The example folder flower\_photos should have a structure like this:  ~/flower\_photos/daisy/photo1.jpg ~/flower\_photos/daisy/photo2.jpg ... ~/flower\_photos/rose/anotherphoto77.jpg ... ~/flower\_photos/sunflower/somepicture.jpg  The subfolder names are important, since they define what label is applied to each image, but the filenames themselves don't matter. Once your images are prepared, you can run the training with a command like this:   bash: bazel build tensorflow/examples/image\_retraining:retrain && \ bazel-bin/tensorflow/examples/image\_retraining/retrain \  --image\_dir ~/flower\_photos   Or, if you have a pip installation of tensorflow, `retrain.py` can be run without bazel:  bash: python tensorflow/examples/image\_retraining/retrain.py \  --image\_dir ~/flower\_photos   You can replace the image\_dir argument with any folder containing subfolders of images. The label for each image is taken from the name of the subfolder it's in.  This produces a new model file that can be loaded and run by any TensorFlow program, for example the label\_image sample code.  By default this script will use the high accuracy, but comparatively large and slow Inception v3 model architecture. It's recommended that you start with this to validate that you have gathered good training data, but if you want to deploy on resource-limited platforms, you can try the `--architecture` flag with a Mobilenet model. For example:  bash: python tensorflow/examples/image\_retraining/retrain.py \  --image\_dir ~/flower\_photos --architecture mobilenet\_1.0\_224   There are 32 different Mobilenet models to choose from, with a variety of file size and latency options. The first number can be '1.0', '0.75', '0.50', or '0.25' to control the size, and the second controls the input image size, either '224', '192', '160', or '128', with smaller sizes running faster. See https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html for more information on Mobilenet.  To use with TensorBoard:  By default, this script will log summaries to /tmp/retrain\_logs directory  Visualize the summaries with this command:  tensorboard --logdir /tmp/retrain\_logs  """* from \_\_future\_\_ import absolute\_import from \_\_future\_\_ import division from \_\_future\_\_ import print\_function  import argparse from datetime import datetime import hashlib import os.path import random import re import sys import tarfile  import numpy as np from six.moves import urllib import tensorflow as tf  from tensorflow.python.framework import graph\_util from tensorflow.python.framework import tensor\_shape from tensorflow.python.platform import gfile from tensorflow.python.util import compat  FLAGS = None  # These are all parameters that are tied to the particular model architecture # we're using for Inception v3. These include things like tensor names and their # sizes. If you want to adapt this script to work with another model, you will # need to update these to reflect the values in the network you're using. MAX\_NUM\_IMAGES\_PER\_CLASS = 2 \*\* 27 - 1 # ~134M   def create\_image\_lists(image\_dir, testing\_percentage, validation\_percentage):  *"""Builds a list of training images from the file system.   Analyzes the sub folders in the image directory, splits them into stable  training, testing, and validation sets, and returns a data structure  describing the lists of images for each label and their paths.   Args:  image\_dir: String path to a folder containing subfolders of images.  testing\_percentage: Integer percentage of the images to reserve for tests.  validation\_percentage: Integer percentage of images reserved for validation.   Returns:  A dictionary containing an entry for each label subfolder, with images split  into training, testing, and validation sets within each label.  """* if not gfile.Exists(image\_dir):  tf.logging.error("Image directory '" + image\_dir + "' not found.")  return None  result = {}  sub\_dirs = [x[0] for x in gfile.Walk(image\_dir)]  # The root directory comes first, so skip it.  is\_root\_dir = True  for sub\_dir in sub\_dirs:  if is\_root\_dir:  is\_root\_dir = False  continue  extensions = ['jpg', 'jpeg', 'JPG', 'JPEG']  file\_list = []  dir\_name = os.path.basename(sub\_dir)  if dir\_name == image\_dir:  continue  tf.logging.info("Looking for images in '" + dir\_name + "'")  for extension in extensions:  file\_glob = os.path.join(image\_dir, dir\_name, '\*.' + extension)  file\_list.extend(gfile.Glob(file\_glob))  if not file\_list:  tf.logging.warning('No files found')  continue  if len(file\_list) < 20:  tf.logging.warning(  'WARNING: Folder has less than 20 images, which may cause issues.')  elif len(file\_list) > MAX\_NUM\_IMAGES\_PER\_CLASS:  tf.logging.warning(  'WARNING: Folder {} has more than {} images. Some images will '  'never be selected.'.format(dir\_name, MAX\_NUM\_IMAGES\_PER\_CLASS))  label\_name = re.sub(r'[^a-z0-9]+', ' ', dir\_name.lower())  training\_images = []  testing\_images = []  validation\_images = []  for file\_name in file\_list:  base\_name = os.path.basename(file\_name)  # We want to ignore anything after '\_nohash\_' in the file name when  # deciding which set to put an image in, the data set creator has a way of  # grouping photos that are close variations of each other. For example  # this is used in the plant disease data set to group multiple pictures of  # the same leaf.  hash\_name = re.sub(r'\_nohash\_.\*$', '', file\_name)  # This looks a bit magical, but we need to decide whether this file should  # go into the training, testing, or validation sets, and we want to keep  # existing files in the same set even if more files are subsequently  # added.  # To do that, we need a stable way of deciding based on just the file name  # itself, so we do a hash of that and then use that to generate a  # probability value that we use to assign it.  hash\_name\_hashed = hashlib.sha1(compat.as\_bytes(hash\_name)).hexdigest()  percentage\_hash = ((int(hash\_name\_hashed, 16) %  (MAX\_NUM\_IMAGES\_PER\_CLASS + 1)) \*  (100.0 / MAX\_NUM\_IMAGES\_PER\_CLASS))  if percentage\_hash < validation\_percentage:  validation\_images.append(base\_name)  elif percentage\_hash < (testing\_percentage + validation\_percentage):  testing\_images.append(base\_name)  else:  training\_images.append(base\_name)  result[label\_name] = {  'dir': dir\_name,  'training': training\_images,  'testing': testing\_images,  'validation': validation\_images,  }  return result   def get\_image\_path(image\_lists, label\_name, index, image\_dir, category):  *""""Returns a path to an image for a label at the given index.   Args:  image\_lists: Dictionary of training images for each label.  label\_name: Label string we want to get an image for.  index: Int offset of the image we want. This will be moduloed by the  available number of images for the label, so it can be arbitrarily large.  image\_dir: Root folder string of the subfolders containing the training  images.  category: Name string of set to pull images from - training, testing, or  validation.   Returns:  File system path string to an image that meets the requested parameters.   """* if label\_name not in image\_lists:  tf.logging.fatal('Label does not exist %s.', label\_name)  label\_lists = image\_lists[label\_name]  if category not in label\_lists:  tf.logging.fatal('Category does not exist %s.', category)  category\_list = label\_lists[category]  if not category\_list:  tf.logging.fatal('Label %s has no images in the category %s.',  label\_name, category)  mod\_index = index % len(category\_list)  base\_name = category\_list[mod\_index]  sub\_dir = label\_lists['dir']  full\_path = os.path.join(image\_dir, sub\_dir, base\_name)  return full\_path   def get\_bottleneck\_path(image\_lists, label\_name, index, bottleneck\_dir,  category, architecture):  *""""Returns a path to a bottleneck file for a label at the given index.   Args:  image\_lists: Dictionary of training images for each label.  label\_name: Label string we want to get an image for.  index: Integer offset of the image we want. This will be moduloed by the  available number of images for the label, so it can be arbitrarily large.  bottleneck\_dir: Folder string holding cached files of bottleneck values.  category: Name string of set to pull images from - training, testing, or  validation.  architecture: The name of the model architecture.   Returns:  File system path string to an image that meets the requested parameters.  """* return get\_image\_path(image\_lists, label\_name, index, bottleneck\_dir,  category) + '\_' + architecture + '.txt'   def create\_model\_graph(model\_info):  *""""Creates a graph from saved GraphDef file and returns a Graph object.   Args:  model\_info: Dictionary containing information about the model architecture.   Returns:  Graph holding the trained Inception network, and various tensors we'll be  manipulating.  """* with tf.Graph().as\_default() as graph:  model\_path = os.path.join(FLAGS.model\_dir, model\_info['model\_file\_name'])  with gfile.FastGFile(model\_path, 'rb') as f:  graph\_def = tf.GraphDef()  graph\_def.ParseFromString(f.read())  bottleneck\_tensor, resized\_input\_tensor = (tf.import\_graph\_def(  graph\_def,  name='',  return\_elements=[  model\_info['bottleneck\_tensor\_name'],  model\_info['resized\_input\_tensor\_name'],  ]))  return graph, bottleneck\_tensor, resized\_input\_tensor   def run\_bottleneck\_on\_image(sess, image\_data, image\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor):  *"""Runs inference on an image to extract the 'bottleneck' summary layer.   Args:  sess: Current active TensorFlow Session.  image\_data: String of raw JPEG data.  image\_data\_tensor: Input data layer in the graph.  decoded\_image\_tensor: Output of initial image resizing and preprocessing.  resized\_input\_tensor: The input node of the recognition graph.  bottleneck\_tensor: Layer before the final softmax.   Returns:  Numpy array of bottleneck values.  """* # First decode the JPEG image, resize it, and rescale the pixel values.  resized\_input\_values = sess.run(decoded\_image\_tensor,  {image\_data\_tensor: image\_data})  # Then run it through the recognition network.  bottleneck\_values = sess.run(bottleneck\_tensor,  {resized\_input\_tensor: resized\_input\_values})  bottleneck\_values = np.squeeze(bottleneck\_values)  return bottleneck\_values   def maybe\_download\_and\_extract(data\_url):  *"""Download and extract model tar file.   If the pretrained model we're using doesn't already exist, this function  downloads it from the TensorFlow.org website and unpacks it into a directory.   Args:  data\_url: Web location of the tar file containing the pretrained model.  """* dest\_directory = FLAGS.model\_dir  if not os.path.exists(dest\_directory):  os.makedirs(dest\_directory)  filename = data\_url.split('/')[-1]  filepath = os.path.join(dest\_directory, filename)  if not os.path.exists(filepath):   def \_progress(count, block\_size, total\_size):  sys.stdout.write('\r>> Downloading %s %.1f%%' %  (filename,  float(count \* block\_size) / float(total\_size) \* 100.0))  sys.stdout.flush()   filepath, \_ = urllib.request.urlretrieve(data\_url, filepath, \_progress)  print()  statinfo = os.stat(filepath)  tf.logging.info('Successfully downloaded', filename, statinfo.st\_size,  'bytes.')  tarfile.open(filepath, 'r:gz').extractall(dest\_directory)   def ensure\_dir\_exists(dir\_name):  *"""Makes sure the folder exists on disk.   Args:  dir\_name: Path string to the folder we want to create.  """* if not os.path.exists(dir\_name):  os.makedirs(dir\_name)   bottleneck\_path\_2\_bottleneck\_values = {}   def create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,  image\_dir, category, sess, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor):  *"""Create a single bottleneck file."""* tf.logging.info('Creating bottleneck at ' + bottleneck\_path)  image\_path = get\_image\_path(image\_lists, label\_name, index,  image\_dir, category)  if not gfile.Exists(image\_path):  tf.logging.fatal('File does not exist %s', image\_path)  image\_data = gfile.FastGFile(image\_path, 'rb').read()  try:  bottleneck\_values = run\_bottleneck\_on\_image(  sess, image\_data, jpeg\_data\_tensor, decoded\_image\_tensor,  resized\_input\_tensor, bottleneck\_tensor)  except Exception as e:  raise RuntimeError('Error during processing file %s (%s)' % (image\_path,  str(e)))  bottleneck\_string = ','.join(str(x) for x in bottleneck\_values)  with open(bottleneck\_path, 'w') as bottleneck\_file:  bottleneck\_file.write(bottleneck\_string)   def get\_or\_create\_bottleneck(sess, image\_lists, label\_name, index, image\_dir,  category, bottleneck\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor, architecture):  *"""Retrieves or calculates bottleneck values for an image.   If a cached version of the bottleneck data exists on-disk, return that,  otherwise calculate the data and save it to disk for future use.   Args:  sess: The current active TensorFlow Session.  image\_lists: Dictionary of training images for each label.  label\_name: Label string we want to get an image for.  index: Integer offset of the image we want. This will be modulo-ed by the  available number of images for the label, so it can be arbitrarily large.  image\_dir: Root folder string of the subfolders containing the training  images.  category: Name string of which set to pull images from - training, testing,  or validation.  bottleneck\_dir: Folder string holding cached files of bottleneck values.  jpeg\_data\_tensor: The tensor to feed loaded jpeg data into.  decoded\_image\_tensor: The output of decoding and resizing the image.  resized\_input\_tensor: The input node of the recognition graph.  bottleneck\_tensor: The output tensor for the bottleneck values.  architecture: The name of the model architecture.   Returns:  Numpy array of values produced by the bottleneck layer for the image.  """* label\_lists = image\_lists[label\_name]  sub\_dir = label\_lists['dir']  sub\_dir\_path = os.path.join(bottleneck\_dir, sub\_dir)  ensure\_dir\_exists(sub\_dir\_path)  bottleneck\_path = get\_bottleneck\_path(image\_lists, label\_name, index,  bottleneck\_dir, category, architecture)  if not os.path.exists(bottleneck\_path):  create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,  image\_dir, category, sess, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor)  with open(bottleneck\_path, 'r') as bottleneck\_file:  bottleneck\_string = bottleneck\_file.read()  did\_hit\_error = False  try:  bottleneck\_values = [float(x) for x in bottleneck\_string.split(',')]  except ValueError:  tf.logging.warning('Invalid float found, recreating bottleneck')  did\_hit\_error = True  if did\_hit\_error:  create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,  image\_dir, category, sess, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor)  with open(bottleneck\_path, 'r') as bottleneck\_file:  bottleneck\_string = bottleneck\_file.read()  # Allow exceptions to propagate here, since they shouldn't happen after a  # fresh creation  bottleneck\_values = [float(x) for x in bottleneck\_string.split(',')]  return bottleneck\_values   def cache\_bottlenecks(sess, image\_lists, image\_dir, bottleneck\_dir,  jpeg\_data\_tensor, decoded\_image\_tensor,  resized\_input\_tensor, bottleneck\_tensor, architecture):  *"""Ensures all the training, testing, and validation bottlenecks are cached.   Because we're likely to read the same image multiple times (if there are no  distortions applied during training) it can speed things up a lot if we  calculate the bottleneck layer values once for each image during  preprocessing, and then just read those cached values repeatedly during  training. Here we go through all the images we've found, calculate those  values, and save them off.   Args:  sess: The current active TensorFlow Session.  image\_lists: Dictionary of training images for each label.  image\_dir: Root folder string of the subfolders containing the training  images.  bottleneck\_dir: Folder string holding cached files of bottleneck values.  jpeg\_data\_tensor: Input tensor for jpeg data from file.  decoded\_image\_tensor: The output of decoding and resizing the image.  resized\_input\_tensor: The input node of the recognition graph.  bottleneck\_tensor: The penultimate output layer of the graph.  architecture: The name of the model architecture.   Returns:  Nothing.  """* how\_many\_bottlenecks = 0  ensure\_dir\_exists(bottleneck\_dir)  for label\_name, label\_lists in image\_lists.items():  for category in ['training', 'testing', 'validation']:  category\_list = label\_lists[category]  for index, unused\_base\_name in enumerate(category\_list):  get\_or\_create\_bottleneck(  sess, image\_lists, label\_name, index, image\_dir, category,  bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor,  resized\_input\_tensor, bottleneck\_tensor, architecture)   how\_many\_bottlenecks += 1  if how\_many\_bottlenecks % 100 == 0:  tf.logging.info(  str(how\_many\_bottlenecks) + ' bottleneck files created.')   def get\_random\_cached\_bottlenecks(sess, image\_lists, how\_many, category,  bottleneck\_dir, image\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_input\_tensor,  bottleneck\_tensor, architecture):  *"""Retrieves bottleneck values for cached images.   If no distortions are being applied, this function can retrieve the cached  bottleneck values directly from disk for images. It picks a random set of  images from the specified category.   Args:  sess: Current TensorFlow Session.  image\_lists: Dictionary of training images for each label.  how\_many: If positive, a random sample of this size will be chosen.  If negative, all bottlenecks will be retrieved.  category: Name string of which set to pull from - training, testing, or  validation.  bottleneck\_dir: Folder string holding cached files of bottleneck values.  image\_dir: Root folder string of the subfolders containing the training  images.  jpeg\_data\_tensor: The layer to feed jpeg image data into.  decoded\_image\_tensor: The output of decoding and resizing the image.  resized\_input\_tensor: The input node of the recognition graph.  bottleneck\_tensor: The bottleneck output layer of the CNN graph.  architecture: The name of the model architecture.   Returns:  List of bottleneck arrays, their corresponding ground truths, and the  relevant filenames.  """* class\_count = len(image\_lists.keys())  bottlenecks = []  ground\_truths = []  filenames = []  if how\_many >= 0:  # Retrieve a random sample of bottlenecks.  for unused\_i in range(how\_many):  label\_index = random.randrange(class\_count)  label\_name = list(image\_lists.keys())[label\_index]  image\_index = random.randrange(MAX\_NUM\_IMAGES\_PER\_CLASS + 1)  image\_name = get\_image\_path(image\_lists, label\_name, image\_index,  image\_dir, category)  bottleneck = get\_or\_create\_bottleneck(  sess, image\_lists, label\_name, image\_index, image\_dir, category,  bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor,  resized\_input\_tensor, bottleneck\_tensor, architecture)  ground\_truth = np.zeros(class\_count, dtype=np.float32)  ground\_truth[label\_index] = 1.0  bottlenecks.append(bottleneck)  ground\_truths.append(ground\_truth)  filenames.append(image\_name)  else:  # Retrieve all bottlenecks.  for label\_index, label\_name in enumerate(image\_lists.keys()):  for image\_index, image\_name in enumerate(  image\_lists[label\_name][category]):  image\_name = get\_image\_path(image\_lists, label\_name, image\_index,  image\_dir, category)  bottleneck = get\_or\_create\_bottleneck(  sess, image\_lists, label\_name, image\_index, image\_dir, category,  bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor,  resized\_input\_tensor, bottleneck\_tensor, architecture)  ground\_truth = np.zeros(class\_count, dtype=np.float32)  ground\_truth[label\_index] = 1.0  bottlenecks.append(bottleneck)  ground\_truths.append(ground\_truth)  filenames.append(image\_name)  return bottlenecks, ground\_truths, filenames   def get\_random\_distorted\_bottlenecks(  sess, image\_lists, how\_many, category, image\_dir, input\_jpeg\_tensor,  distorted\_image, resized\_input\_tensor, bottleneck\_tensor):  *"""Retrieves bottleneck values for training images, after distortions.   If we're training with distortions like crops, scales, or flips, we have to  recalculate the full model for every image, and so we can't use cached  bottleneck values. Instead we find random images for the requested category,  run them through the distortion graph, and then the full graph to get the  bottleneck results for each.   Args:  sess: Current TensorFlow Session.  image\_lists: Dictionary of training images for each label.  how\_many: The integer number of bottleneck values to return.  category: Name string of which set of images to fetch - training, testing,  or validation.  image\_dir: Root folder string of the subfolders containing the training  images.  input\_jpeg\_tensor: The input layer we feed the image data to.  distorted\_image: The output node of the distortion graph.  resized\_input\_tensor: The input node of the recognition graph.  bottleneck\_tensor: The bottleneck output layer of the CNN graph.   Returns:  List of bottleneck arrays and their corresponding ground truths.  """* class\_count = len(image\_lists.keys())  bottlenecks = []  ground\_truths = []  for unused\_i in range(how\_many):  label\_index = random.randrange(class\_count)  label\_name = list(image\_lists.keys())[label\_index]  image\_index = random.randrange(MAX\_NUM\_IMAGES\_PER\_CLASS + 1)  image\_path = get\_image\_path(image\_lists, label\_name, image\_index, image\_dir,  category)  if not gfile.Exists(image\_path):  tf.logging.fatal('File does not exist %s', image\_path)  jpeg\_data = gfile.FastGFile(image\_path, 'rb').read()  # Note that we materialize the distorted\_image\_data as a numpy array before  # sending running inference on the image. This involves 2 memory copies and  # might be optimized in other implementations.  distorted\_image\_data = sess.run(distorted\_image,  {input\_jpeg\_tensor: jpeg\_data})  bottleneck\_values = sess.run(bottleneck\_tensor,  {resized\_input\_tensor: distorted\_image\_data})  bottleneck\_values = np.squeeze(bottleneck\_values)  ground\_truth = np.zeros(class\_count, dtype=np.float32)  ground\_truth[label\_index] = 1.0  bottlenecks.append(bottleneck\_values)  ground\_truths.append(ground\_truth)  return bottlenecks, ground\_truths   def should\_distort\_images(flip\_left\_right, random\_crop, random\_scale,  random\_brightness):  *"""Whether any distortions are enabled, from the input flags.   Args:  flip\_left\_right: Boolean whether to randomly mirror images horizontally.  random\_crop: Integer percentage setting the total margin used around the  crop box.  random\_scale: Integer percentage of how much to vary the scale by.  random\_brightness: Integer range to randomly multiply the pixel values by.   Returns:  Boolean value indicating whether any distortions should be applied.  """* return (flip\_left\_right or (random\_crop != 0) or (random\_scale != 0) or  (random\_brightness != 0))   def add\_input\_distortions(flip\_left\_right, random\_crop, random\_scale,  random\_brightness, input\_width, input\_height,  input\_depth, input\_mean, input\_std):  *"""Creates the operations to apply the specified distortions.   During training it can help to improve the results if we run the images  through simple distortions like crops, scales, and flips. These reflect the  kind of variations we expect in the real world, and so can help train the  model to cope with natural data more effectively. Here we take the supplied  parameters and construct a network of operations to apply them to an image.   Cropping  ~~~~~~~~   Cropping is done by placing a bounding box at a random position in the full  image. The cropping parameter controls the size of that box relative to the  input image. If it's zero, then the box is the same size as the input and no  cropping is performed. If the value is 50%, then the crop box will be half the  width and height of the input. In a diagram it looks like this:   < width >  +---------------------+  | |  | width - crop% |  | < > |  | +------+ |  | | | |  | | | |  | | | |  | +------+ |  | |  | |  +---------------------+   Scaling  ~~~~~~~   Scaling is a lot like cropping, except that the bounding box is always  centered and its size varies randomly within the given range. For example if  the scale percentage is zero, then the bounding box is the same size as the  input and no scaling is applied. If it's 50%, then the bounding box will be in  a random range between half the width and height and full size.   Args:  flip\_left\_right: Boolean whether to randomly mirror images horizontally.  random\_crop: Integer percentage setting the total margin used around the  crop box.  random\_scale: Integer percentage of how much to vary the scale by.  random\_brightness: Integer range to randomly multiply the pixel values by.  graph.  input\_width: Horizontal size of expected input image to model.  input\_height: Vertical size of expected input image to model.  input\_depth: How many channels the expected input image should have.  input\_mean: Pixel value that should be zero in the image for the graph.  input\_std: How much to divide the pixel values by before recognition.   Returns:  The jpeg input layer and the distorted result tensor.  """* jpeg\_data = tf.placeholder(tf.string, name='DistortJPGInput')  decoded\_image = tf.image.decode\_jpeg(jpeg\_data, channels=input\_depth)  decoded\_image\_as\_float = tf.cast(decoded\_image, dtype=tf.float32)  decoded\_image\_4d = tf.expand\_dims(decoded\_image\_as\_float, 0)  margin\_scale = 1.0 + (random\_crop / 100.0)  resize\_scale = 1.0 + (random\_scale / 100.0)  margin\_scale\_value = tf.constant(margin\_scale)  resize\_scale\_value = tf.random\_uniform(tensor\_shape.scalar(),  minval=1.0,  maxval=resize\_scale)  scale\_value = tf.multiply(margin\_scale\_value, resize\_scale\_value)  precrop\_width = tf.multiply(scale\_value, input\_width)  precrop\_height = tf.multiply(scale\_value, input\_height)  precrop\_shape = tf.stack([precrop\_height, precrop\_width])  precrop\_shape\_as\_int = tf.cast(precrop\_shape, dtype=tf.int32)  precropped\_image = tf.image.resize\_bilinear(decoded\_image\_4d,  precrop\_shape\_as\_int)  precropped\_image\_3d = tf.squeeze(precropped\_image, squeeze\_dims=[0])  cropped\_image = tf.random\_crop(precropped\_image\_3d,  [input\_height, input\_width, input\_depth])  if flip\_left\_right:  flipped\_image = tf.image.random\_flip\_left\_right(cropped\_image)  else:  flipped\_image = cropped\_image  brightness\_min = 1.0 - (random\_brightness / 100.0)  brightness\_max = 1.0 + (random\_brightness / 100.0)  brightness\_value = tf.random\_uniform(tensor\_shape.scalar(),  minval=brightness\_min,  maxval=brightness\_max)  brightened\_image = tf.multiply(flipped\_image, brightness\_value)  offset\_image = tf.subtract(brightened\_image, input\_mean)  mul\_image = tf.multiply(offset\_image, 1.0 / input\_std)  distort\_result = tf.expand\_dims(mul\_image, 0, name='DistortResult')  return jpeg\_data, distort\_result   def variable\_summaries(var):  *"""Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""* with tf.name\_scope('summaries'):  mean = tf.reduce\_mean(var)  tf.summary.scalar('mean', mean)  with tf.name\_scope('stddev'):  stddev = tf.sqrt(tf.reduce\_mean(tf.square(var - mean)))  tf.summary.scalar('stddev', stddev)  tf.summary.scalar('max', tf.reduce\_max(var))  tf.summary.scalar('min', tf.reduce\_min(var))  tf.summary.histogram('histogram', var)   def add\_final\_training\_ops(class\_count, final\_tensor\_name, bottleneck\_tensor,  bottleneck\_tensor\_size):  *"""Adds a new softmax and fully-connected layer for training.   We need to retrain the top layer to identify our new classes, so this function  adds the right operations to the graph, along with some variables to hold the  weights, and then sets up all the gradients for the backward pass.   The set up for the softmax and fully-connected layers is based on:  https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html   Args:  class\_count: Integer of how many categories of things we're trying to  recognize.  final\_tensor\_name: Name string for the new final node that produces results.  bottleneck\_tensor: The output of the main CNN graph.  bottleneck\_tensor\_size: How many entries in the bottleneck vector.   Returns:  The tensors for the training and cross entropy results, and tensors for the  bottleneck input and ground truth input.  """* with tf.name\_scope('input'):  bottleneck\_input = tf.placeholder\_with\_default(  bottleneck\_tensor,  shape=[None, bottleneck\_tensor\_size],  name='BottleneckInputPlaceholder')   ground\_truth\_input = tf.placeholder(tf.float32,  [None, class\_count],  name='GroundTruthInput')   # Organizing the following ops as `final\_training\_ops` so they're easier  # to see in TensorBoard  layer\_name = 'final\_training\_ops'  with tf.name\_scope(layer\_name):  with tf.name\_scope('weights'):  initial\_value = tf.truncated\_normal(  [bottleneck\_tensor\_size, class\_count], stddev=0.001)   layer\_weights = tf.Variable(initial\_value, name='final\_weights')   variable\_summaries(layer\_weights)  with tf.name\_scope('biases'):  layer\_biases = tf.Variable(tf.zeros([class\_count]), name='final\_biases')  variable\_summaries(layer\_biases)  with tf.name\_scope('Wx\_plus\_b'):  logits = tf.matmul(bottleneck\_input, layer\_weights) + layer\_biases  tf.summary.histogram('pre\_activations', logits)   final\_tensor = tf.nn.softmax(logits, name=final\_tensor\_name)  tf.summary.histogram('activations', final\_tensor)   with tf.name\_scope('cross\_entropy'):  cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(  labels=ground\_truth\_input, logits=logits)  with tf.name\_scope('total'):  cross\_entropy\_mean = tf.reduce\_mean(cross\_entropy)  tf.summary.scalar('cross\_entropy', cross\_entropy\_mean)   with tf.name\_scope('train'):  optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning\_rate)  train\_step = optimizer.minimize(cross\_entropy\_mean)   return (train\_step, cross\_entropy\_mean, bottleneck\_input, ground\_truth\_input,  final\_tensor)   def add\_evaluation\_step(result\_tensor, ground\_truth\_tensor):  *"""Inserts the operations we need to evaluate the accuracy of our results.   Args:  result\_tensor: The new final node that produces results.  ground\_truth\_tensor: The node we feed ground truth data  into.   Returns:  Tuple of (evaluation step, prediction).  """* with tf.name\_scope('accuracy'):  with tf.name\_scope('correct\_prediction'):  prediction = tf.argmax(result\_tensor, 1)  correct\_prediction = tf.equal(  prediction, tf.argmax(ground\_truth\_tensor, 1))  with tf.name\_scope('accuracy'):  evaluation\_step = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  tf.summary.scalar('accuracy', evaluation\_step)  return evaluation\_step, prediction   def save\_graph\_to\_file(sess, graph, graph\_file\_name):  output\_graph\_def = graph\_util.convert\_variables\_to\_constants(  sess, graph.as\_graph\_def(), [FLAGS.final\_tensor\_name])  with gfile.FastGFile(graph\_file\_name, 'wb') as f:  f.write(output\_graph\_def.SerializeToString())  return   def prepare\_file\_system():  # Setup the directory we'll write summaries to for TensorBoard  if tf.gfile.Exists(FLAGS.summaries\_dir):  tf.gfile.DeleteRecursively(FLAGS.summaries\_dir)  tf.gfile.MakeDirs(FLAGS.summaries\_dir)  if FLAGS.intermediate\_store\_frequency > 0:  ensure\_dir\_exists(FLAGS.intermediate\_output\_graphs\_dir)  return   def create\_model\_info(architecture):  *"""Given the name of a model architecture, returns information about it.   There are different base image recognition pretrained models that can be  retrained using transfer learning, and this function translates from the name  of a model to the attributes that are needed to download and train with it.   Args:  architecture: Name of a model architecture.   Returns:  Dictionary of information about the model, or None if the name isn't  recognized   Raises:  ValueError: If architecture name is unknown.  """* architecture = architecture.lower()  if architecture == 'inception\_v3':  # pylint: disable=line-too-long  data\_url = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz'  # pylint: enable=line-too-long  bottleneck\_tensor\_name = 'pool\_3/\_reshape:0'  bottleneck\_tensor\_size = 2048  input\_width = 299  input\_height = 299  input\_depth = 3  resized\_input\_tensor\_name = 'Mul:0'  model\_file\_name = 'classify\_image\_graph\_def.pb'  input\_mean = 128  input\_std = 128  elif architecture.startswith('mobilenet\_'):  parts = architecture.split('\_')  if len(parts) != 3 and len(parts) != 4:  tf.logging.error("Couldn't understand architecture name '%s'",  architecture)  return None  version\_string = parts[1]  if (version\_string != '1.0' and version\_string != '0.75' and  version\_string != '0.50' and version\_string != '0.25'):  tf.logging.error(  """"The Mobilenet version should be '1.0', '0.75', '0.50', or '0.25',  but found '%s' for architecture '%s'""",  version\_string, architecture)  return None  size\_string = parts[2]  if (size\_string != '224' and size\_string != '192' and  size\_string != '160' and size\_string != '128'):  tf.logging.error(  """The Mobilenet input size should be '224', '192', '160', or '128',  but found '%s' for architecture '%s'""",  size\_string, architecture)  return None  if len(parts) == 3:  is\_quantized = False  else:  if parts[3] != 'quantized':  tf.logging.error(  "Couldn't understand architecture suffix '%s' for '%s'", parts[3],  architecture)  return None  is\_quantized = True  data\_url = 'http://download.tensorflow.org/models/mobilenet\_v1\_'  data\_url += version\_string + '\_' + size\_string + '\_frozen.tgz'  bottleneck\_tensor\_name = 'MobilenetV1/Predictions/Reshape:0'  bottleneck\_tensor\_size = 1001  input\_width = int(size\_string)  input\_height = int(size\_string)  input\_depth = 3  resized\_input\_tensor\_name = 'input:0'  if is\_quantized:  model\_base\_name = 'quantized\_graph.pb'  else:  model\_base\_name = 'frozen\_graph.pb'  model\_dir\_name = 'mobilenet\_v1\_' + version\_string + '\_' + size\_string  model\_file\_name = os.path.join(model\_dir\_name, model\_base\_name)  input\_mean = 127.5  input\_std = 127.5  else:  tf.logging.error("Couldn't understand architecture name '%s'", architecture)  raise ValueError('Unknown architecture', architecture)   return {  'data\_url': data\_url,  'bottleneck\_tensor\_name': bottleneck\_tensor\_name,  'bottleneck\_tensor\_size': bottleneck\_tensor\_size,  'input\_width': input\_width,  'input\_height': input\_height,  'input\_depth': input\_depth,  'resized\_input\_tensor\_name': resized\_input\_tensor\_name,  'model\_file\_name': model\_file\_name,  'input\_mean': input\_mean,  'input\_std': input\_std,  }   def add\_jpeg\_decoding(input\_width, input\_height, input\_depth, input\_mean,  input\_std):  *"""Adds operations that perform JPEG decoding and resizing to the graph..   Args:  input\_width: Desired width of the image fed into the recognizer graph.  input\_height: Desired width of the image fed into the recognizer graph.  input\_depth: Desired channels of the image fed into the recognizer graph.  input\_mean: Pixel value that should be zero in the image for the graph.  input\_std: How much to divide the pixel values by before recognition.   Returns:  Tensors for the node to feed JPEG data into, and the output of the  preprocessing steps.  """* jpeg\_data = tf.placeholder(tf.string, name='DecodeJPGInput')  decoded\_image = tf.image.decode\_jpeg(jpeg\_data, channels=input\_depth)  decoded\_image\_as\_float = tf.cast(decoded\_image, dtype=tf.float32)  decoded\_image\_4d = tf.expand\_dims(decoded\_image\_as\_float, 0)  resize\_shape = tf.stack([input\_height, input\_width])  resize\_shape\_as\_int = tf.cast(resize\_shape, dtype=tf.int32)  resized\_image = tf.image.resize\_bilinear(decoded\_image\_4d,  resize\_shape\_as\_int)  offset\_image = tf.subtract(resized\_image, input\_mean)  mul\_image = tf.multiply(offset\_image, 1.0 / input\_std)  return jpeg\_data, mul\_image   def main(\_):  # Needed to make sure the logging output is visible.  # See https://github.com/tensorflow/tensorflow/issues/3047  tf.logging.set\_verbosity(tf.logging.INFO)   # Prepare necessary directories that can be used during training  prepare\_file\_system()   # Gather information about the model architecture we'll be using.  model\_info = create\_model\_info(FLAGS.architecture)  if not model\_info:  tf.logging.error('Did not recognize architecture flag')  return -1   # Set up the pre-trained graph.  maybe\_download\_and\_extract(model\_info['data\_url'])  graph, bottleneck\_tensor, resized\_image\_tensor = (  create\_model\_graph(model\_info))   # Look at the folder structure, and create lists of all the images.  image\_lists = create\_image\_lists(FLAGS.image\_dir, FLAGS.testing\_percentage,  FLAGS.validation\_percentage)  class\_count = len(image\_lists.keys())  if class\_count == 0:  tf.logging.error('No valid folders of images found at ' + FLAGS.image\_dir)  return -1  if class\_count == 1:  tf.logging.error('Only one valid folder of images found at ' +  FLAGS.image\_dir +  ' - multiple classes are needed for classification.')  return -1   # See if the command-line flags mean we're applying any distortions.  do\_distort\_images = should\_distort\_images(  FLAGS.flip\_left\_right, FLAGS.random\_crop, FLAGS.random\_scale,  FLAGS.random\_brightness)   with tf.Session(graph=graph) as sess:  # Set up the image decoding sub-graph.  jpeg\_data\_tensor, decoded\_image\_tensor = add\_jpeg\_decoding(  model\_info['input\_width'], model\_info['input\_height'],  model\_info['input\_depth'], model\_info['input\_mean'],  model\_info['input\_std'])   if do\_distort\_images:  # We will be applying distortions, so setup the operations we'll need.  (distorted\_jpeg\_data\_tensor,  distorted\_image\_tensor) = add\_input\_distortions(  FLAGS.flip\_left\_right, FLAGS.random\_crop, FLAGS.random\_scale,  FLAGS.random\_brightness, model\_info['input\_width'],  model\_info['input\_height'], model\_info['input\_depth'],  model\_info['input\_mean'], model\_info['input\_std'])  else:  # We'll make sure we've calculated the 'bottleneck' image summaries and  # cached them on disk.  cache\_bottlenecks(sess, image\_lists, FLAGS.image\_dir,  FLAGS.bottleneck\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_image\_tensor,  bottleneck\_tensor, FLAGS.architecture)   # Add the new layer that we'll be training.  (train\_step, cross\_entropy, bottleneck\_input, ground\_truth\_input,  final\_tensor) = add\_final\_training\_ops(  len(image\_lists.keys()), FLAGS.final\_tensor\_name, bottleneck\_tensor,  model\_info['bottleneck\_tensor\_size'])   # Create the operations we need to evaluate the accuracy of our new layer.  evaluation\_step, prediction = add\_evaluation\_step(  final\_tensor, ground\_truth\_input)   # Merge all the summaries and write them out to the summaries\_dir  merged = tf.summary.merge\_all()  train\_writer = tf.summary.FileWriter(FLAGS.summaries\_dir + '/train',  sess.graph)   validation\_writer = tf.summary.FileWriter(  FLAGS.summaries\_dir + '/validation')   # Set up all our weights to their initial default values.  init = tf.global\_variables\_initializer()  sess.run(init)   # Run the training for as many cycles as requested on the command line.  for i in range(FLAGS.how\_many\_training\_steps):  # Get a batch of input bottleneck values, either calculated fresh every  # time with distortions applied, or from the cache stored on disk.  if do\_distort\_images:  (train\_bottlenecks,  train\_ground\_truth) = get\_random\_distorted\_bottlenecks(  sess, image\_lists, FLAGS.train\_batch\_size, 'training',  FLAGS.image\_dir, distorted\_jpeg\_data\_tensor,  distorted\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor)  else:  (train\_bottlenecks,  train\_ground\_truth, \_) = get\_random\_cached\_bottlenecks(  sess, image\_lists, FLAGS.train\_batch\_size, 'training',  FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor,  FLAGS.architecture)  # Feed the bottlenecks and ground truth into the graph, and run a training  # step. Capture training summaries for TensorBoard with the `merged` op.  train\_summary, \_ = sess.run(  [merged, train\_step],  feed\_dict={bottleneck\_input: train\_bottlenecks,  ground\_truth\_input: train\_ground\_truth})  train\_writer.add\_summary(train\_summary, i)   # Every so often, print out how well the graph is training.  is\_last\_step = (i + 1 == FLAGS.how\_many\_training\_steps)  if (i % FLAGS.eval\_step\_interval) == 0 or is\_last\_step:  train\_accuracy, cross\_entropy\_value = sess.run(  [evaluation\_step, cross\_entropy],  feed\_dict={bottleneck\_input: train\_bottlenecks,  ground\_truth\_input: train\_ground\_truth})  tf.logging.info('%s: Step %d: Train accuracy = %.1f%%' %  (datetime.now(), i, train\_accuracy \* 100))  tf.logging.info('%s: Step %d: Cross entropy = %f' %  (datetime.now(), i, cross\_entropy\_value))  validation\_bottlenecks, validation\_ground\_truth, \_ = (  get\_random\_cached\_bottlenecks(  sess, image\_lists, FLAGS.validation\_batch\_size, 'validation',  FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor,  FLAGS.architecture))  # Run a validation step and capture training summaries for TensorBoard  # with the `merged` op.  validation\_summary, validation\_accuracy = sess.run(  [merged, evaluation\_step],  feed\_dict={bottleneck\_input: validation\_bottlenecks,  ground\_truth\_input: validation\_ground\_truth})  validation\_writer.add\_summary(validation\_summary, i)  tf.logging.info('%s: Step %d: Validation accuracy = %.1f%% (N=%d)' %  (datetime.now(), i, validation\_accuracy \* 100,  len(validation\_bottlenecks)))   # Store intermediate results  intermediate\_frequency = FLAGS.intermediate\_store\_frequency   if (intermediate\_frequency > 0 and (i % intermediate\_frequency == 0)  and i > 0):  intermediate\_file\_name = (FLAGS.intermediate\_output\_graphs\_dir +  'intermediate\_' + str(i) + '.pb')  tf.logging.info('Save intermediate result to : ' +  intermediate\_file\_name)  save\_graph\_to\_file(sess, graph, intermediate\_file\_name)   # We've completed all our training, so run a final test evaluation on  # some new images we haven't used before.  test\_bottlenecks, test\_ground\_truth, test\_filenames = (  get\_random\_cached\_bottlenecks(  sess, image\_lists, FLAGS.test\_batch\_size, 'testing',  FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor,  decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor,  FLAGS.architecture))  test\_accuracy, predictions = sess.run(  [evaluation\_step, prediction],  feed\_dict={bottleneck\_input: test\_bottlenecks,  ground\_truth\_input: test\_ground\_truth})  tf.logging.info('Final test accuracy = %.1f%% (N=%d)' %  (test\_accuracy \* 100, len(test\_bottlenecks)))   if FLAGS.print\_misclassified\_test\_images:  tf.logging.info('=== MISCLASSIFIED TEST IMAGES ===')  for i, test\_filename in enumerate(test\_filenames):  if predictions[i] != test\_ground\_truth[i].argmax():  tf.logging.info('%70s %s' %  (test\_filename,  list(image\_lists.keys())[predictions[i]]))   # Write out the trained graph and labels with the weights stored as  # constants.  save\_graph\_to\_file(sess, graph, FLAGS.output\_graph)  with gfile.FastGFile(FLAGS.output\_labels, 'w') as f:  f.write('\n'.join(image\_lists.keys()) + '\n')   if \_\_name\_\_ == '\_\_main\_\_':  parser = argparse.ArgumentParser()  parser.add\_argument(  '--image\_dir',  type=str,  default='/Users/qbq\_wzk/Desktop/shuiguo',  help='Path to folders of labeled images.'  )  parser.add\_argument(  '--output\_graph',  type=str,  default='tmp/output\_graph.pb',  help='Where to save the trained graph.'  )  parser.add\_argument(  '--intermediate\_output\_graphs\_dir',  type=str,  default='tmp/intermediate\_graph/',  help='Where to save the intermediate graphs.'  )  parser.add\_argument(  '--intermediate\_store\_frequency',  type=int,  default=0,  help="""\  How many steps to store intermediate graph. If "0" then will not  store.\  """  )  parser.add\_argument(  '--output\_labels',  type=str,  default='tmp/output\_labels.txt',  help='Where to save the trained graph\'s labels.'  )  parser.add\_argument(  '--summaries\_dir',  type=str,  default='tmp/retrain\_logs',  help='Where to save summary logs for TensorBoard.'  )  parser.add\_argument(  '--how\_many\_training\_steps',  type=int,  default=200,  help='How many training steps to run before ending.'  )  parser.add\_argument(  '--learning\_rate',  type=float,  default=0.01,  help='How large a learning rate to use when training.'  )  parser.add\_argument(  '--testing\_percentage',  type=int,  default=10,  help='What percentage of images to use as a test set.'  )  parser.add\_argument(  '--validation\_percentage',  type=int,  default=10,  help='What percentage of images to use as a validation set.'  )  parser.add\_argument(  '--eval\_step\_interval',  type=int,  default=10,  help='How often to evaluate the training results.'  )  parser.add\_argument(  '--train\_batch\_size',  type=int,  default=100,  help='How many images to train on at a time.'  )  parser.add\_argument(  '--test\_batch\_size',  type=int,  default=-1,  help="""\  How many images to test on. This test set is only used once, to evaluate  the final accuracy of the model after training completes.  A value of -1 causes the entire test set to be used, which leads to more  stable results across runs.\  """  )  parser.add\_argument(  '--validation\_batch\_size',  type=int,  default=100,  help="""\  How many images to use in an evaluation batch. This validation set is  used much more often than the test set, and is an early indicator of how  accurate the model is during training.  A value of -1 causes the entire validation set to be used, which leads to  more stable results across training iterations, but may be slower on large  training sets.\  """  )  parser.add\_argument(  '--print\_misclassified\_test\_images',  default=False,  help="""\  Whether to print out a list of all misclassified test images.\  """,  action='store\_true'  )  parser.add\_argument(  '--model\_dir',  type=str,  default='tmp/imagenet',  help="""\  Path to classify\_image\_graph\_def.pb,  imagenet\_synset\_to\_human\_label\_map.txt, and  imagenet\_2012\_challenge\_label\_map\_proto.pbtxt.\  """  )  parser.add\_argument(  '--bottleneck\_dir',  type=str,  default='tmp/bottleneck',  help='Path to cache bottleneck layer values as files.'  )  parser.add\_argument(  '--final\_tensor\_name',  type=str,  default='final\_result',  help="""\  The name of the output classification layer in the retrained graph.\  """  )  parser.add\_argument(  '--flip\_left\_right',  default=False,  help="""\  Whether to randomly flip half of the training images horizontally.\  """,  action='store\_true'  )  parser.add\_argument(  '--random\_crop',  type=int,  default=0,  help="""\  A percentage determining how much of a margin to randomly crop off the  training images.\  """  )  parser.add\_argument(  '--random\_scale',  type=int,  default=0,  help="""\  A percentage determining how much to randomly scale up the size of the  training images by.\  """  )  parser.add\_argument(  '--random\_brightness',  type=int,  default=0,  help="""\  A percentage determining how much to randomly multiply the training image  input pixels up or down by.\  """  )  parser.add\_argument(  '--architecture',  type=str,  default='inception\_v3',  help="""\  Which model architecture to use. 'inception\_v3' is the most accurate, but  also the slowest. For faster or smaller models, chose a MobileNet with the  form 'mobilenet\_<parameter size>\_<input\_size>[\_quantized]'. For example,  'mobilenet\_1.0\_224' will pick a model that is 17 MB in size and takes 224  pixel input images, while 'mobilenet\_0.25\_128\_quantized' will choose a much  less accurate, but smaller and faster network that's 920 KB on disk and  takes 128x128 images. See https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html  for more information on Mobilenet.\  """)  FLAGS, unparsed = parser.parse\_known\_args()  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)  parser.add\_argument(  '--test\_batch\_size',  type=int,  default=-1,  help="""\  How many images to test on. This test set is only used once, to evaluate  the final accuracy of the model after training completes.  A value of -1 causes the entire test set to be used, which leads to more  stable results across runs.\  """  )  parser.add\_argument(  '--validation\_batch\_size',  type=int,  default=100,  help="""\  How many images to use in an evaluation batch. This validation set is  used much more often than the test set, and is an early indicator of how  accurate the model is during training.  A value of -1 causes the entire validation set to be used, which leads to  more stable results across training iterations, but may be slower on large  training sets.\  """  )  parser.add\_argument(  '--print\_misclassified\_test\_images',  default=False,  help="""\  Whether to print out a list of all misclassified test images.\  """,  action='store\_true'  )  parser.add\_argument(  '--model\_dir',  type=str,  default='tmp/imagenet',  help="""\  Path to classify\_image\_graph\_def.pb,  imagenet\_synset\_to\_human\_label\_map.txt, and  imagenet\_2012\_challenge\_label\_map\_proto.pbtxt.\  """  )  parser.add\_argument(  '--bottleneck\_dir',  type=str,  default='tmp/bottleneck',  help='Path to cache bottleneck layer values as files.'  )  parser.add\_argument(  '--final\_tensor\_name',  type=str,  default='final\_result',  help="""\  The name of the output classification layer in the retrained graph.\  """  )  parser.add\_argument(  '--flip\_left\_right',  default=False,  help="""\  Whether to randomly flip half of the training images horizontally.\  """,  action='store\_true'  )  parser.add\_argument(  '--random\_crop',  type=int,  default=0,  help="""\  A percentage determining how much of a margin to randomly crop off the  training images.\  """  )  parser.add\_argument(  '--random\_scale',  type=int,  default=0,  help="""\  A percentage determining how much to randomly scale up the size of the  training images by.\  """  )  parser.add\_argument(  '--random\_brightness',  type=int,  default=0,  help="""\  A percentage determining how much to randomly multiply the training image  input pixels up or down by.\  """  )  parser.add\_argument(  '--architecture',  type=str,  default='inception\_v3',  help="""\  Which model architecture to use. 'inception\_v3' is the most accurate, but  also the slowest. For faster or smaller models, chose a MobileNet with the  form 'mobilenet\_<parameter size>\_<input\_size>[\_quantized]'. For example,  'mobilenet\_1.0\_224' will pick a model that is 17 MB in size and takes 224  pixel input images, while 'mobilenet\_0.25\_128\_quantized' will choose a much  less accurate, but smaller and faster network that's 920 KB on disk and  takes 128x128 images. See https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html  for more information on Mobilenet.\  """)  FLAGS, unparsed = parser.parse\_known\_args()  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)  以上代码没有使用训练模型的下载地址，直接使用的模型名，由代码下载。下面是测试代码：  # -\*- coding: utf-8 -\*- *""" Created on Fri Oct 13 16:15:16 2017 use\_output\_graph 使用retrain所训练的迁移后的inception模型来测试 @author: Dexter """* import os os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' import tensorflow as tf import numpy as np import os from PIL import Image import matplotlib.pyplot as plt  model\_name = 'tmp/output\_graph.pb' image\_dir = 'data/validation' label\_filename = 'tmp/output\_labels.txt'   # 读取并创建一个图graph来存放Google训练好的Inception\_v3模型（函数） def create\_graph():  with tf.gfile.FastGFile(model\_name, 'rb') as f:  # 使用tf.GraphDef()定义一个空的Graph  graph\_def = tf.GraphDef()  graph\_def.ParseFromString(f.read())  # Imports the graph from graph\_def into the current default Graph.  tf.import\_graph\_def(graph\_def, name='')   # 读取标签labels def load\_labels(label\_file\_dir):  if not tf.gfile.Exists(label\_file\_dir):  # 预先检测地址是否存在  tf.logging.fatal('File does not exist %s', label\_file\_dir)  else:  # 读取所有的标签返并回一个list  labels = tf.gfile.GFile(label\_file\_dir).readlines()  for i in range(len(labels)):  labels[i] = labels[i].strip('\n')  return labels   # 创建graph create\_graph()  # 创建会话，因为是从已有的Inception\_v3模型中恢复，所以无需初始化 with tf.Session() as sess:  # Inception\_v3模型的最后一层final\_result:0的输出  softmax\_tensor = sess.graph.get\_tensor\_by\_name('final\_result:0')   # 遍历目录  for root, dirs, files in os.walk(image\_dir):  for file in files:  # 载入图片  image\_data = tf.gfile.FastGFile(os.path.join(root, file), 'rb').read()  # 输入图像（jpg格式）数据，得到softmax概率值（一个shape=(1,1008)的向量）  predictions = sess.run(softmax\_tensor, {'DecodeJpeg/contents:0': image\_data})  # 将结果转为1维数据  predictions = np.squeeze(predictions)   # 打印图片路径及名称  image\_path = os.path.join(root, file)  print(image\_path)  # 显示图片  img = Image.open(image\_path)  plt.imshow(img)  plt.axis('off')  plt.show()   # 排序，取出前5个概率最大的值（top-5),本数据集一共就5个  # argsort()返回的是数组值从小到大排列所对应的索引值  top\_5 = predictions.argsort()[-5:][::-1]  for label\_index in top\_5:  # 获取分类名称  label\_name = load\_labels(label\_filename)[label\_index]  # 获取该分类的置信度  label\_score = predictions[label\_index]  print('%s (score = %.5f)' % (label\_name, label\_score))  print()  至此可以训练完成，但训练完的模型在手机端还是无法识别，报错：  dealPics: java.lang.RuntimeException: Node 'output' does not exist in model '  查找模型无法使用原因 |