In [1]:

```
###  we copy Retrain.ipynb from internet ,which is written by Radek Bartyzal.
###  What we did is retrain the new model and change some parameters
###  We write the loop ourselves.
```

In [2]:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from datetime import datetime
import glob
import hashlib
import os.path
import random
import re
import sys
import tarfile
import os

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
import struct
```

In [3]:

```
FLAGS = tf.app.flags.FLAGS

# Input and output file flags.
tf.app.flags.DEFINE_string('image_dir', 'images',
                           """Path to folders of labeled images.""")
tf.app.flags.DEFINE_string('output_graph', '/tmp/Final/output_graph.pb',
                           """Where to save the trained graph.""")
tf.app.flags.DEFINE_string('output_labels', '/tmp/Final/output_labels.txt',
                           """Where to save the trained graph's labels.""")
tf.app.flags.DEFINE_string('summaries_dir', '/tmp/Final/retrain_logs',
                          """Where to save summary logs for TensorBoard.""")

# Details of the training configuration.
tf.app.flags.DEFINE_integer('how_many_training_steps', 2000,
                            """How many training steps to run before ending.""")
tf.app.flags.DEFINE_float('learning_rate', 0.01,
                          """How large a learning rate to use when training.""")
tf.app.flags.DEFINE_integer(
    'testing_percentage', 2,
    """What percentage of images to use as a test set.""")
tf.app.flags.DEFINE_integer(
    'validation_percentage', 10,
    """What percentage of images to use as a validation set.""")
tf.app.flags.DEFINE_integer('eval_step_interval', 10,
                            """How often to evaluate the training results.""")
tf.app.flags.DEFINE_integer('train_batch_size', 16,
                            """How many images to train on at a time.""")
tf.app.flags.DEFINE_integer('test_batch_size',16,
                            """How many images to test on at a time. This"""
                            """ test set is only used infrequently to verify"""
                            """ the overall accuracy of the model.""")
tf.app.flags.DEFINE_integer(
    'validation_batch_size', 16,
    """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.""")

# File-system cache locations.
tf.app.flags.DEFINE_string('model_dir', '/tmp/Final/imagenet',
                           """Path to classify_image_graph_def.pb, """
                           """imagenet_synset_to_human_label_map.txt, and """
                           """imagenet_2012_challenge_label_map_proto.pbtxt.""")
tf.app.flags.DEFINE_string(
    'bottleneck_dir', '/tmp/Final/bottleneck',
    """Path to cache bottleneck layer values as files.""")
tf.app.flags.DEFINE_string('final_tensor_name', 'final_result',
                           """The name of the output classification layer in"""
                           """ the retrained graph.""")

# Controls the distortions used during training.
tf.app.flags.DEFINE_boolean(
    'flip_left_right', False,
    """Whether to randomly flip half of the training images horizontally.""")
tf.app.flags.DEFINE_integer(
    'random_crop', 0,
    """A percentage determining how much of a margin to randomly crop off the"""
    """ training images.""")
tf.app.flags.DEFINE_integer(
    'random_scale', 0,
    """A percentage determining how much to randomly scale up the size of the"""
    """ training images by.""")
tf.app.flags.DEFINE_integer(
    'random_brightness', 0,
    """A percentage determining how much to randomly multiply the training"""
    """ image input pixels up or down by.""")

# 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.
# pylint: disable=line-too-long
DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-
```

```
# pylint: enable=line-too-long
BOTTLENECK_TENSOR_NAME = 'pool_3/_reshape:0'
BOTTLENECK_TENSOR_SIZE = 2048
MODEL_INPUT_WIDTH = 400
MODEL_INPUT_HEIGHT = 400
MODEL_INPUT_DEPTH = 3
JPEG_DATA_TENSOR_NAME = 'DecodeJpeg/contents:0'
RESIZED_INPUT_TENSOR_NAME = 'ResizeBilinear:0'
MAX_NUM_IMAGES_PER_CLASS = 2 ** 27 - 1  # ~134M

# Directory containing files with correct image labels for each image.
IMAGE_LABELS_DIR = '/Users/mingjuhe/Desktop/CPE646-Finalprject/image_labels_dir'
# Contains cached ground_truth vectors to prevent calculating them again and again
CACHED_GROUND_TRUTH_VECTORS = {}
# Contains list of all labels, each label is on a separate line, just like in image_
ALL_LABELS_FILE = "/Users/mingjuhe/Desktop/CPE646-Finalprject/labels.txt"


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):
    print("Image directory '" + image_dir + "' not found.")
    return None
  result = {}
  sub_dirs = [x[0] for x in os.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
    print("Looking for images in '" + dir_name + "'")
    for extension in extensions:
      file_glob = os.path.join(image_dir, dir_name, '*.' + extension)
      file_list.extend(glob.glob(file_glob))
    if not file_list:
      print('No files found')
      continue
    if len(file_list) < 20:
      print('WARNING: Folder has less than 20 images, which may cause issues.')
    elif len(file_list) > MAX_NUM_IMAGES_PER_CLASS:
      print('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_labels_path(image_lists, label_name, index, image_labels_dir, category
  """"Returns a path to a file containing correct image labels.

  This is just slightly edited get_image_path() method.

  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
    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_labels_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]
  full_path = os.path.join(image_labels_dir, base_name)
  full_path += '.txt'
  return full_path

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']
  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):
  """"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.

  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) + '.txt'


def create_inception_graph():
  """"Creates a graph from saved GraphDef file and returns a Graph object.

  Returns:
    Graph holding the trained Inception network, and various tensors we'll be
    manipulating.
  """
  with tf.Session() as sess:
    model_filename = os.path.join(
        FLAGS.model_dir, 'classify_image_graph_def.pb')
    with gfile.FastGFile(model_filename, 'rb') as f:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(f.read())
      bottleneck_tensor, jpeg_data_tensor, resized_input_tensor = (
          tf.import_graph_def(graph_def, name='', return_elements=[
              BOTTLENECK_TENSOR_NAME, JPEG_DATA_TENSOR_NAME,
              RESIZED_INPUT_TENSOR_NAME]))
  return sess.graph, bottleneck_tensor, jpeg_data_tensor, resized_input_tensor


def run_bottleneck_on_image(sess, image_data, image_data_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.
    bottleneck_tensor: Layer before the final softmax.
  Returns:
    Numpy array of bottleneck values.
  """
  bottleneck_values = sess.run(
      bottleneck_tensor,
      {image_data_tensor: image_data})
  bottleneck_values = np.squeeze(bottleneck_values)
  return bottleneck_values


def maybe_download_and_extract():
  """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.
  """
  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)
    print('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)


def write_list_of_floats_to_file(list_of_floats , file_path):
  """Writes a given list of floats to a binary file.

  Args:
    list_of_floats: List of floats we want to write to a file.
    file_path: Path to a file where list of floats will be stored
    file_path: Path to a file where list of floats will be stored.

  """

  s = struct.pack('d' * BOTTLENECK_TENSOR_SIZE, *list_of_floats)
  with open(file_path, 'wb') as f:
    f.write(s)


def read_list_of_floats_from_file(file_path):
  """Reads list of floats from a given file.

  Args:
    file_path: Path to a file where list of floats was stored.
  Returns:
    Array of bottleneck values (list of floats).

  """

  with open(file_path, 'rb') as f:
    s = struct.unpack('d' * BOTTLENECK_TENSOR_SIZE, f.read())
    return list(s)


bottleneck_path_2_bottleneck_values = {}


def get_or_create_bottleneck(sess, image_lists, label_name, index, image_dir,
                             category, bottleneck_dir, jpeg_data_tensor,
                             bottleneck_tensor):
  """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.
    bottleneck_tensor: The output tensor for the bottleneck values.

  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)
  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)
  if not os.path.exists(bottleneck_path):
    print('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()
    bottleneck_values = run_bottleneck_on_image(sess, image_data,
                                                jpeg_data_tensor,
                                                bottleneck_tensor)
    bottleneck_string = ','.join(str(x) for x in bottleneck_values)
    with open(bottleneck_path, 'w') as bottleneck_file:
      bottleneck_file.write(bottleneck_string)

  with open(bottleneck_path, 'r') as bottleneck_file:
    bottleneck_string = bottleneck_file.read()
  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, bottleneck_tensor):
  """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.
    bottleneck_tensor: The penultimate output layer of the graph.

  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,
                                 image_dir, category, bottleneck_dir,
                                 jpeg_data_tensor, bottleneck_tensor)
        how_many_bottlenecks += 1
        if how_many_bottlenecks % 100 == 0:
          print(str(how_many_bottlenecks) + ' bottleneck files created.')

def get_ground_truth(labels_file, labels, class_count):
    if labels_file in CACHED_GROUND_TRUTH_VECTORS.keys():
        ground_truth = CACHED_GROUND_TRUTH_VECTORS[labels_file]
    else:
        with open(labels_file) as f:
            true_labels = f.read().splitlines()
        ground_truth = np.zeros(class_count, dtype=np.float32)

        idx = 0
        for label in labels:
            if label in true_labels:
                ground_truth[idx] = 1.0
            idx += 1
        CACHED_GROUND_TRUTH_VECTORS[labels_file] = ground_truth

    return ground_truth

def get_random_cached_bottlenecks(sess, image_lists, how_many, category,
                                  bottleneck_dir, image_dir, jpeg_data_tensor,
                                  bottleneck_tensor, labels):
  """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: The number of bottleneck values to return.
    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.
    bottleneck_tensor: The bottleneck output layer of the CNN graph.
    labels: All possible labels loaded from file labels.txt.

  Returns:
    List of bottleneck arrays and their corresponding ground truths.
  """
  # class_count = len(image_lists.keys())
  class_count = len(labels)
  bottlenecks = []
  ground_truths = []
  for unused_i in range(how_many):
    # label_index = random.randrange(class_count)
    label_index = 0 # there is only one folder with images = 'multi-label'
    label_name = list(image_lists.keys())[label_index]
    image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
    bottleneck = get_or_create_bottleneck(sess, image_lists, label_name,
                                          image_index, image_dir, category,
                                          bottleneck_dir, jpeg_data_tensor,
                                          bottleneck_tensor)

    labels_file = get_image_labels_path(image_lists, label_name, image_index, IMAGE_
    ground_truth = get_ground_truth(labels_file, labels, class_count)

    bottlenecks.append(bottleneck)
    ground_truths.append(ground_truth)
  return bottlenecks, ground_truths


def get_random_distorted_bottlenecks(
    sess, image_lists, how_many, category, image_dir, input_jpeg_tensor,
    distorted_image, resized_input_tensor, bottleneck_tensor, labels):
  """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.
    labels: All possible labels loaded from file labels.txt.

  Returns:
    List of bottleneck arrays and their corresponding ground truths.
  """
  class_count = len(labels)
  bottlenecks = []
  ground_truths = []
  for unused_i in range(how_many):
    label_index = 0 # there is only one folder with images = 'multi-label'
    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)
      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 = run_bottleneck_on_image(sess, distorted_image_data,
                                         resized_input_tensor,
                                         bottleneck_tensor)

    labels_file = get_image_labels_path(image_lists, label_name, image_index, IMAGE_
    ground_truth = get_ground_truth(labels_file, labels, class_count)

    bottlenecks.append(bottleneck)
    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):
  """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.

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=MODEL_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, MODEL_INPUT_WIDTH)
precrop_height = tf.multiply(scale_value, MODEL_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,
                                 [MODEL_INPUT_HEIGHT, MODEL_INPUT_WIDTH,
                                  MODEL_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)
  distort_result = tf.expand_dims(brightened_image, 0, name='DistortResult')
  return jpeg_data, distort_result


def variable_summaries(var, name):
  """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/' + name, mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev/' + name, stddev)
    tf.summary.scalar('max/' + name, tf.reduce_max(var))
    tf.summary.scalar('min/' + name, tf.reduce_min(var))
    tf.summary.histogram(name, var)


def add_final_training_ops(class_count, final_tensor_name, bottleneck_tensor):
  """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://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.

  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'):
      layer_weights = tf.Variable(tf.truncated_normal([BOTTLENECK_TENSOR_SIZE, class
      variable_summaries(layer_weights, layer_name + '/weights')
    with tf.name_scope('biases'):
      layer_biases = tf.Variable(tf.zeros([class_count]), name='final_biases')
      variable_summaries(layer_biases, layer_name + '/biases')
    with tf.name_scope('Wx_plus_b'):
      logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
      tf.summary.histogram(layer_name + '/pre_activations', logits)

  final_tensor = tf.nn.sigmoid(logits, name=final_tensor_name)
  tf.summary.histogram(final_tensor_name + '/activations', final_tensor)

  with tf.name_scope('cross_entropy'):
    cross_entropy = tf.nn.sigmoid_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'):
    train_step = tf.train.GradientDescentOptimizer(FLAGS.learning_rate).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:
    Nothing.
  """
  with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
      # tf.argmax(result_tensor, 1) = return index of maximal value (= 1 in a 1-of-N
      # But we have more ones (indicating multiple labels) in one row of result_tens
      # correct_prediction = tf.equal(tf.argmax(result_tensor, 1), \
      #   tf.argmax(ground_truth_tensor, 1))

      # ground_truth is not a binary tensor, it contains the probabilities of each
      # to acquire a binary tensor allowing comparison by tf.equal()
      # See: http://stackoverflow.com/questions/39219414/in-tensorflow-how-can-i-ge

      correct_prediction = tf.equal(tf.round(result_tensor), ground_truth_tensor)
    with tf.name_scope('accuracy'):
      # Mean accuracy over all labels:
      # http://stackoverflow.com/questions/37746670/tensorflow-multi-label-accuracy-
      evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy', evaluation_step)
  return evaluation_step


def main(_):
  # 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)

  # Set up the pre-trained graph.
  maybe_download_and_extract()
  graph, bottleneck_tensor, jpeg_data_tensor, resized_image_tensor = (
      create_inception_graph())

  # 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)

  if len(image_lists.keys()) == 0:

      print('Folder containing training images has not been found inside {} directo
            'Put all the training images into '
            'one folder inside {} directory and delete everything else inside the {
            .format(FLAGS.image_dir, FLAGS.image_dir, FLAGS.image_dir))
      return -1

  if len(image_lists.keys()) > 1:
      print('More than one folder found inside {} directory. \n'
            'In order to prevent validation issues, put all the training images into
            'one folder inside {} directory and delete everything else inside the {
            .format(FLAGS.image_dir, FLAGS.image_dir, FLAGS.image_dir))
      return -1

  if not os.path.isfile(ALL_LABELS_FILE):
      print('File {} containing all possible labels (= classes) does not exist.\n'
            'Create it in project root and put each possible label on new line, '
            'it is exactly the same as creating an image_label file for image '
            'that is in all the possible classes.'.format(ALL_LABELS_FILE))
      return -1

  with open(ALL_LABELS_FILE) as f:
      labels = f.read().splitlines()
```

```
class_count = len(labels)

if class_count == 0:
  print('No valid labels inside file {} that should contain all possible labels (
  return -1
if class_count == 1:
  print('Only one valid label found inside {} - multiple classes are needed for c
  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)
sess = tf.Session()

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)
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, bottleneck_tensor)

# Add the new layer that we'll be training.
(train_step, cross_entropy, bottleneck_input, ground_truth_input,
 final_tensor) = add_final_training_ops(class_count,
                                        FLAGS.final_tensor_name,
                                        bottleneck_tensor)

# Create the operations we need to evaluate the accuracy of our new layer.
evaluation_step = add_evaluation_step(final_tensor, ground_truth_input)

# Merge all the summaries and write them out to /tmp/retrain_logs (by default)
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,
        bottleneck_tensor, labels)
  # 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})
    print('%s: Step %d: Train accuracy = %.1f%%' % (datetime.now(), i,
                                                    train_accuracy * 100))
    print('%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,
            bottleneck_tensor, labels))
    # 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)
    print('%s: Step %d: Validation accuracy = %.1f%%' %
           (datetime.now(), i, validation_accuracy * 100))

# 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 = get_random_cached_bottlenecks(
    sess, image_lists, FLAGS.test_batch_size, 'testing',
    FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
    bottleneck_tensor, labels)
test_accuracy = sess.run(
    evaluation_step,
    feed_dict={bottleneck_input: test_bottlenecks,
               ground_truth_input: test_ground_truth})
print('Final test accuracy = %.1f%%' % (test_accuracy * 100))

# Write out the trained graph and labels with the weights stored as constants.
output_graph_def = graph_util.convert_variables_to_constants(
    sess, graph.as_graph_def(), [FLAGS.final_tensor_name])
with gfile.FastGFile(FLAGS.output_graph, 'wb') as f:
    f.write(output_graph_def.SerializeToString())
  with gfile.FastGFile(FLAGS.output_labels, 'w') as f:
    f.write('\n'.join(image_lists.keys()) + '\n')


if __name__ == '__main__':
  tf.app.run()
```

```
Looking for images in 'multi-label'
100 bottleneck files created.
200 bottleneck files created.
300 bottleneck files created.
400 bottleneck files created.
500 bottleneck files created.
600 bottleneck files created.
700 bottleneck files created.
800 bottleneck files created.
INFO:tensorflow:Summary name cross entropy is illegal; using cross_en
tropy instead.
2017-12-10 22:47:51.813554: Step 0: Train accuracy = 75.0%
2017-12-10 22:47:51.814116: Step 0: Cross entropy = 0.611278
2017-12-10 22:47:51.837131: Step 0: Validation accuracy = 62.5%
2017-12-10 22:47:52.028879: Step 10: Train accuracy = 68.8%
2017-12-10 22:47:52.029005: Step 10: Cross entropy = 0.491507
2017-12-10 22:47:52.046030: Step 10: Validation accuracy = 62.5%
2017-12-10 22:47:52.226428: Step 20: Train accuracy = 78.1%
2017-12-10 22:47:52.226549: Step 20: Cross entropy = 0.491291
2017-12-10 22:47:52.243071: Step 20: Validation accuracy = 84.4%
```

In [ ]: