Tecnológico de Monterrey

Proyecto BD Clasificador de Imágenes

Alejandro Delgado Medrano A01227074

Bases de Datos Avanzadas

GRUPO 01

Profesor Rodolfo Rubén Alvarez González

En el presente proyecto se hará una comparación de imágenes, usando tensorflow para enseñarle a la computadora a distinguirlas a base de carpetas de las mismas. En este caso son 3 carpetas de perros de diferentes razas, al juntar un numero mayor a 50 imágenes en cada carpeta el programa usando machine learning, aprenderá a diferenciarlos. Al igual para probar que si sea cierto se agregaran imágenes nuevas en otra carpeta en la cual el programa las revisara y comparara mostrando al final del proceso cual raza son y cual es su porcentaje de seguridad en ello.

Tensorflow:

Es una librería de código libre para Machine Learning.

TensorFlow fue desarrollado originalmente por investigadores e ingenieros que trabajaban en el equipo de Google Brain Team, dentro del departamento de investigación de Machine Intelligence, con el propósito de llevar a cabo el aprendizaje automático y la investigación de redes neuronales profundas.

TensorFlow es una herramienta de machine learning, popularizada por su eficiencia con redes neuronales de aprendizaje profundo pero que permite la ejecución de procesos distribuidos que no tengan nada que ver con redes neuronales.

Tiene múltiples usos, por ejemplo:

- Puede **reconocer varias palabras** del alfabeto porque relaciona las letras y fonemas.
- **imágenes y textos** que se pueden relacionar entre sí rápidamente gracias a la capacidad de asociación del sistema de redes neuronales.
- Motor de reconocimiento de imágenes **DeepDream**
- mejorar la fotografía de los smartphones
- Para ayudar al diagnóstico médico

La arquitectura flexible de TensorFlow le permite implementar el cálculo a una o más CPU o GPU en equipos de escritorio, servidores o dispositivos móviles con una sola API. Tiene muchas arquitecturas, pero yo use el driver de Spark como gestor de parámetros

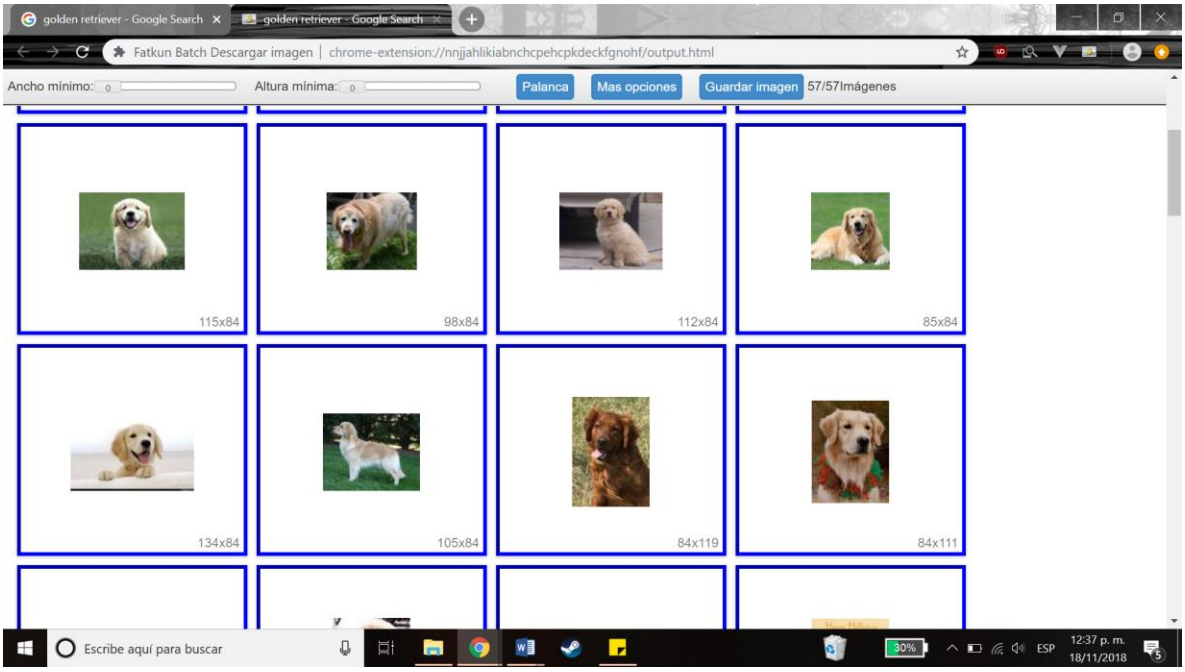
Ventajas:

- Tiene flexibilidad de plataforma.
- Es fácil de entrenar en CPU y GPU para computación distribuida.
- TensorFlow tiene capacidades de diferenciación automática
- Tiene soporte avanzado para subprocesos, cómputo asíncrono y colas.

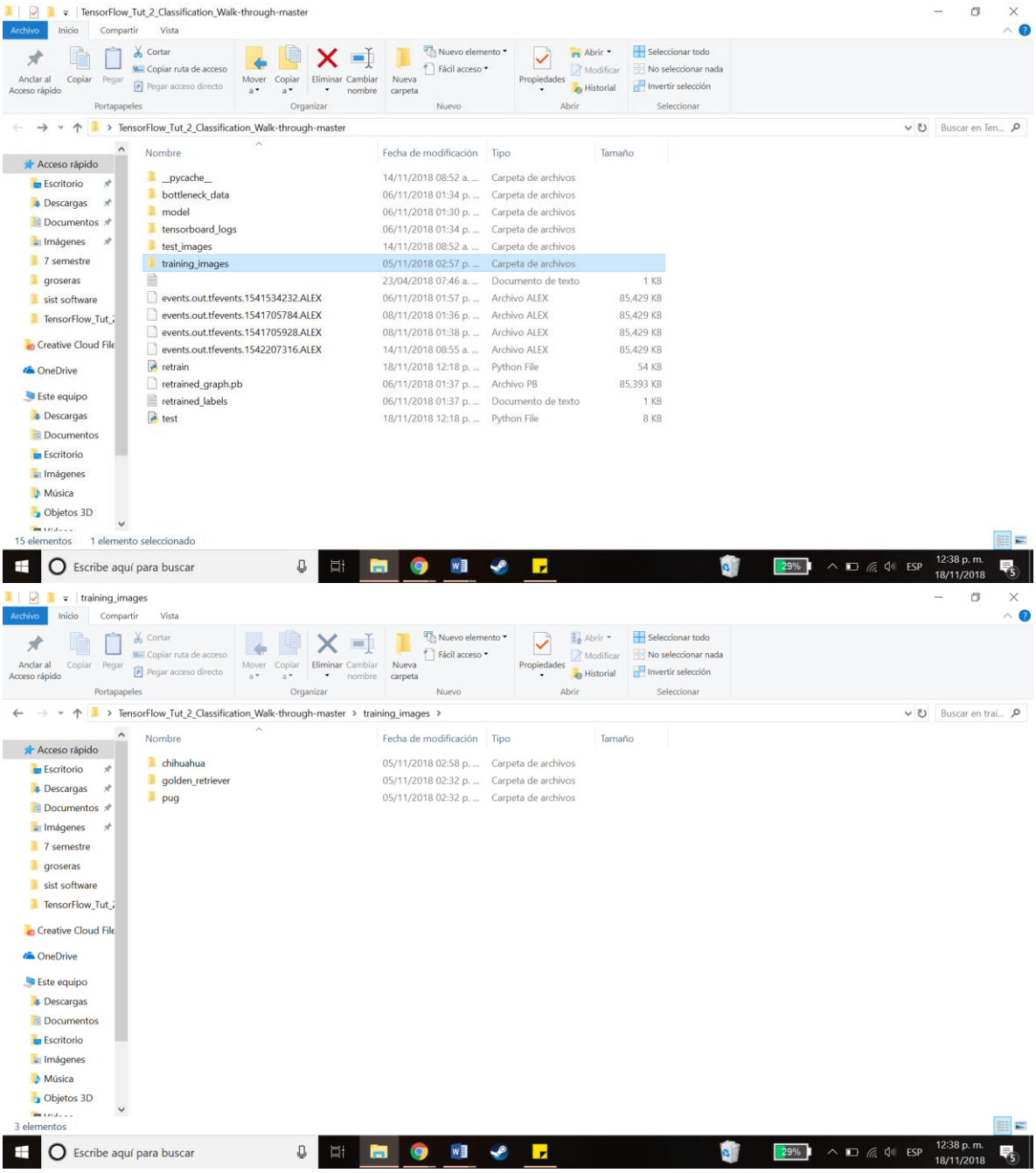
- Es un código personalizable y de código abierto.

Demostración:

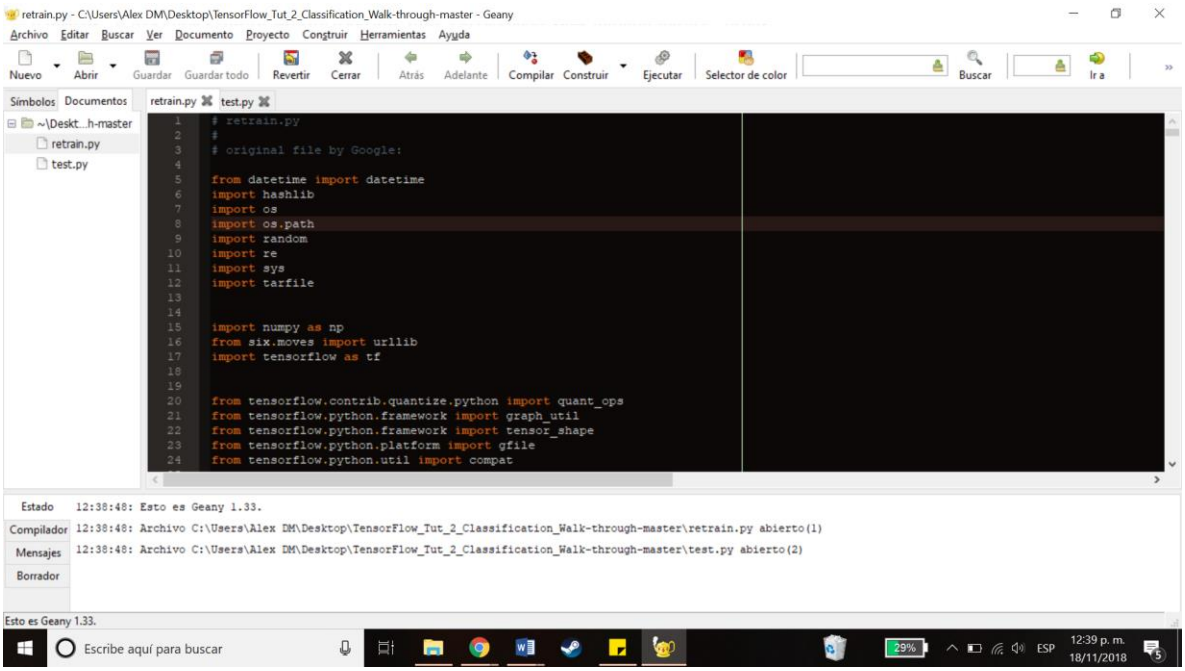
Lo primero que se tiene que hacer es la recolección de imágenes, para esto recomiendo usar una herramienta de Google Chrom llamado “Fatkun” con la cual se selecciona una categoría de imágenes a base de Google y las descarga en la máquina para su uso.



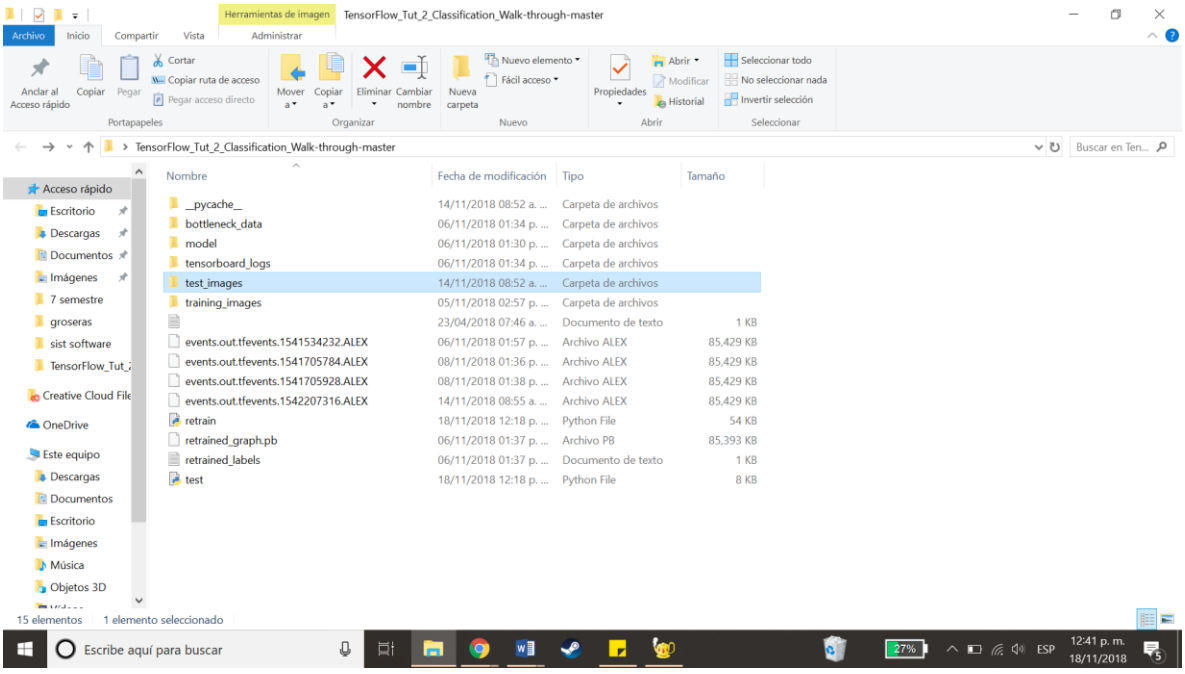
Después de eso se deben agregar todas esas imágenes en la carpeta del proyecto llamado “training_images”

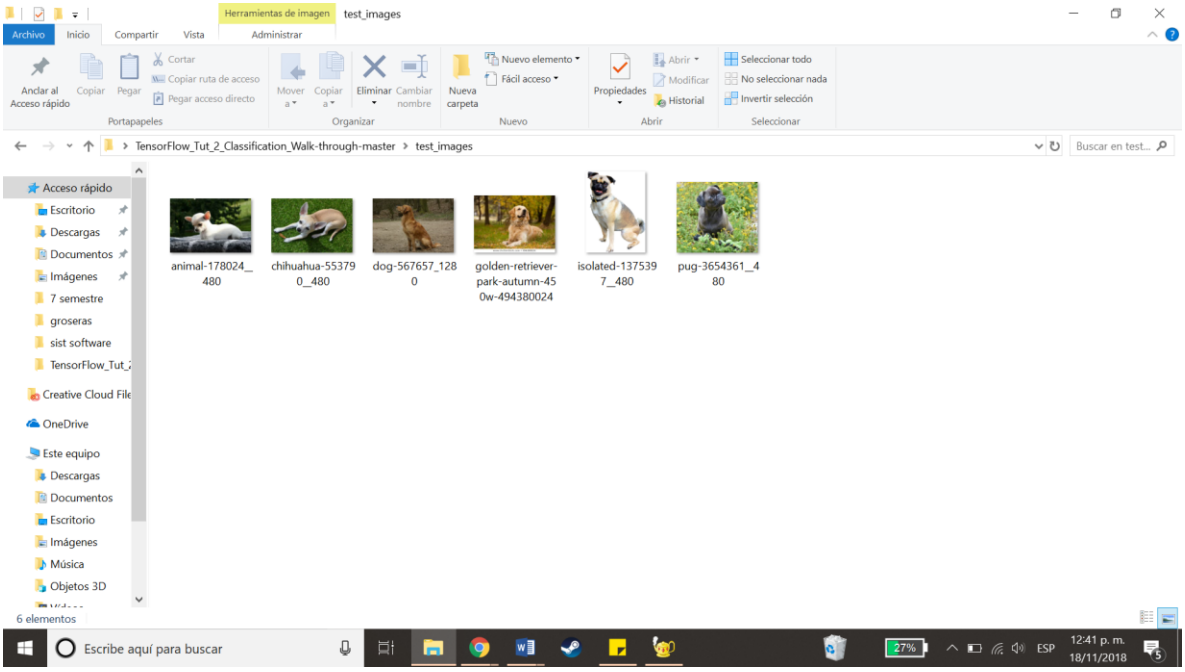


Para el siguiente paso abrimos el código de “retrain” (yo uso Geany por su simpleza y efectividad) el cual es el que se encarga de hacer el machine learning con las imágenes ya descargadas (el proceso puede tardar un rato)

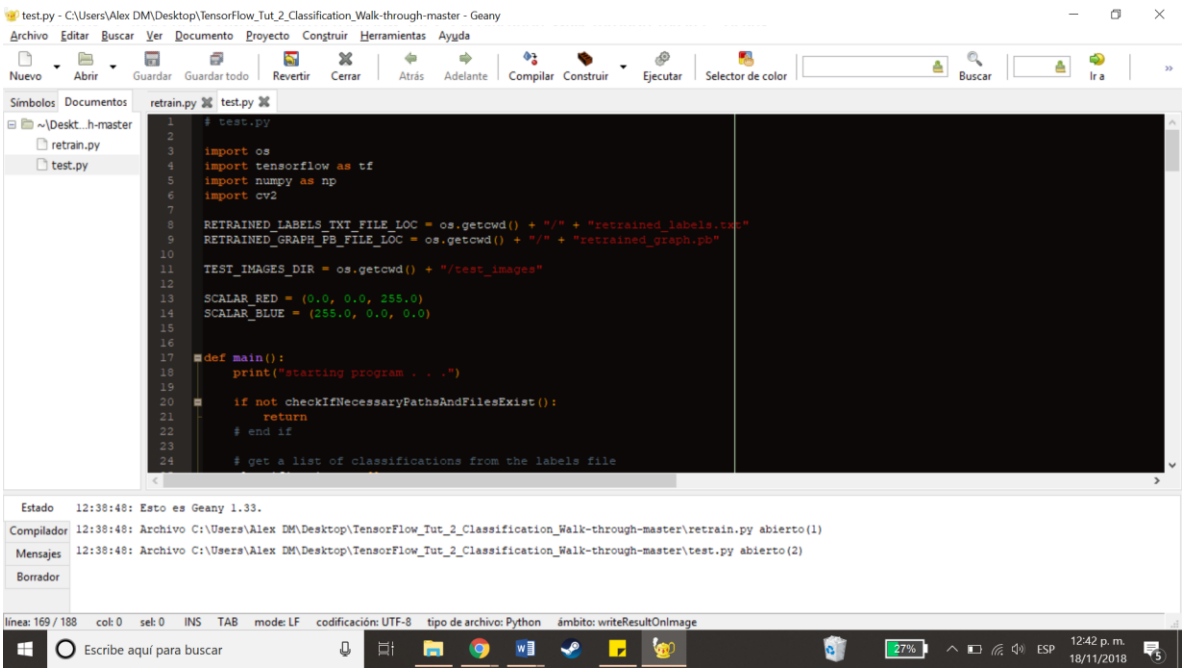


Una vez terminado de correr elegimos nuevas imágenes de las categorías ya descargadas (con 1 o 2 es suficiente) y las insertamos en la carpeta de “test_images”

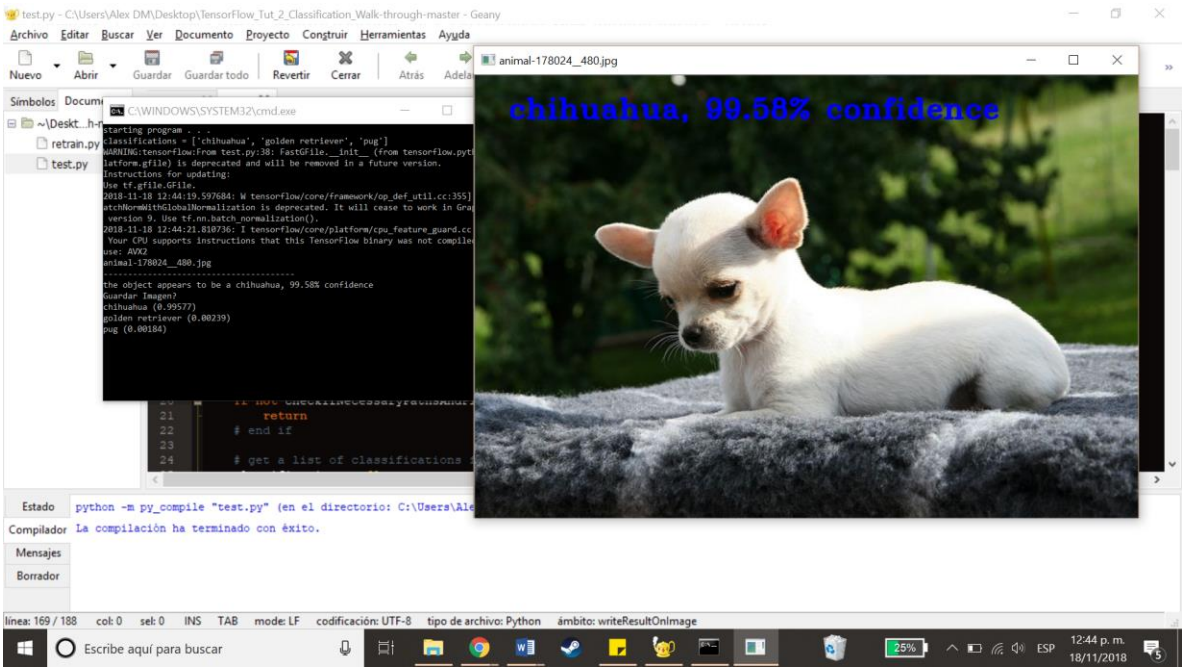




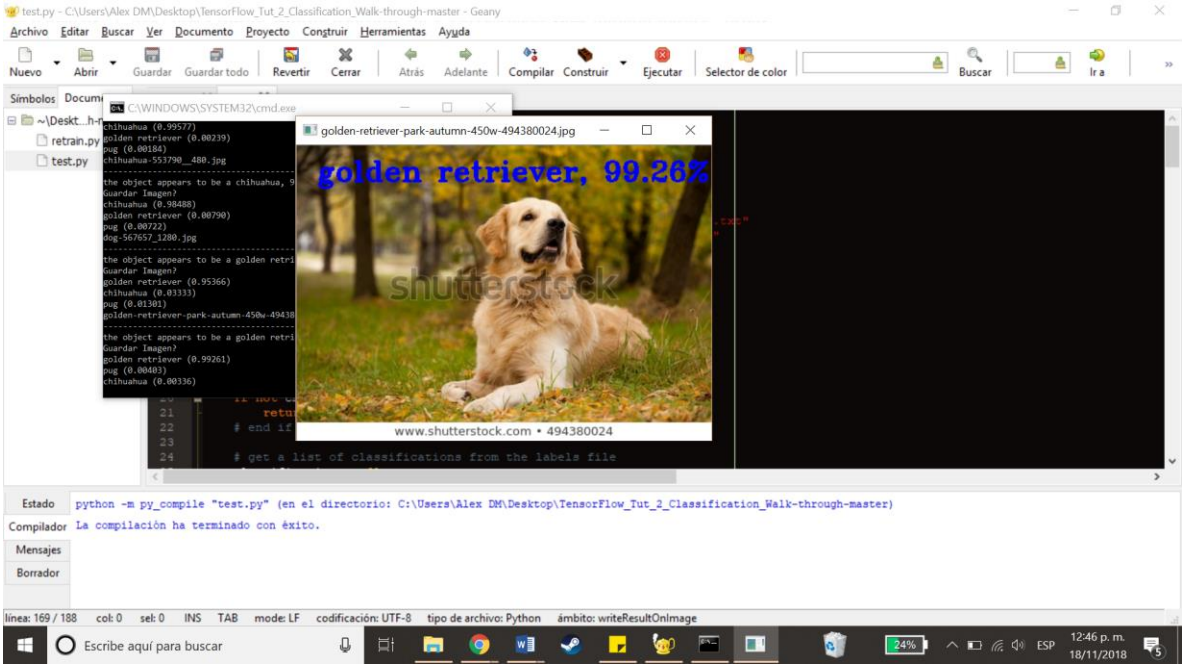
Una ves terminado abrimos el código de “test” en el cual se correra lo mas importante del proyecto para probar que la base de datos si sirva, al igual que la maquina si pueda diferenciar las imágenes.



Al terminar de correr el código se mostrara en pantalla los resultados de las imágenes:



como podemos ver la primera imagen la identifico como un Chihuahua con una seguridad de 99.58%



Al igual nos muestra lo mismo de la siguiente, pero con un poco menos de seguridad.

Una vez terminado el programa y las imágenes en esa carpeta el CMD se cerrara y esta terminado el ejemplo.

Código:

En el presente código se explicaran la función de cada parte de código, usando los comentarios y la explicación de porque fue hecho así.

En el primero tenemos el código de “retrain”, en este código es donde se le enseña a la base de datos a base de Machine learning a aprender a diferenciar las imágenes basándonos en las carpetas establecidas y lograr un rango del 95% al 99% de seguridad.

```
# retrain.py

# original file by Google:

from datetime import datetime

import hashlib

import os

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.contrib.quantize.python import quant_ops

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


# module level variables

MIN_NUM_IMAGES_REQUIRED_FOR_TRAINING = 10

MIN_NUM_IMAGES_SUGGESTED_FOR_TRAINING = 100

MIN_NUM_IMAGES_REQUIRED_FOR_TESTING = 3

MAX_NUM_IMAGES_PER_CLASS = 2 ** 27 - 1  # ~134M


# path to folders of labeled images

TRAINING_IMAGES_DIR = os.getcwd() + '/training_images'


TEST_IMAGES_DIR = os.getcwd() + "/test_images/"


# where to save the trained graph

OUTPUT_GRAPH = os.getcwd() + '/' + 'retrained_graph.pb'


# where to save the intermediate graphs

INTERMEDIATE_OUTPUT_GRAPHS_DIR = os.getcwd() + '/intermediate_graph'


# how many steps to store intermediate graph, if "0" then will not store

INTERMEDIATE_STORE_FREQUENCY = 0


# where to save the trained graph's labels

OUTPUT_LABELS = os.getcwd() + '/' + 'retrained_labels.txt'


# where to save summary logs for TensorBoard

TENSORBOARD_DIR = os.getcwd() + '/' + 'tensorboard_logs'


# how many training steps to run before ending
```

```
# NOTE: original Google default is 4000, use 4000 (or possibly higher) for production grade results

HOW_MANY_TRAINING_STEPS=500


# how large a learning rate to use when training

LEARNING_RATE = 0.01


# what percentage of images to use as a test set

TESTING_PERCENTAGE = 10


# what percentage of images to use as a validation set

VALIDATION_PERCENTAGE = 10


# how often to evaluate the training results

EVAL_STEP_INTERVAL = 10


# how many images to train on at a time

TRAIN_BATCH_SIZE = 100


# 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.

TEST_BATCH_SIZE = -1


# 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.

VALIDATION_BATCH_SIZE = 100


# whether to print out a list of all misclassified test images

PRINT_MISCLASSIFIED_TEST_IMAGES = False


# Path to classify_image_graph_def.pb, imagenet_synset_to_human_label_map.txt, and imagenet_2012_challenge_label_map_proto.pbtxt

MODEL_DIR = os.getcwd() + "/" + "model"


# Path to cache bottleneck layer values as files

BOTTLENECK_DIR = os.getcwd() + '/' + 'bottleneck_data'


# the name of the output classification layer in the retrained graph

FINAL_TENSOR_NAME = 'final_result'


# whether to randomly flip half of the training images horizontally

FLIP_LEFT_RIGHT = False


# a percentage determining how much of a margin to randomly crop off the training images

RANDOM_CROP = 0


# a percentage determining how much to randomly scale up the size of the training images by

RANDOM_SCALE = 0


# a percentage determining how much to randomly multiply the training image input pixels up or down by

RANDOM_BRIGHTNESS = 0


ARCHITECTURE = 'inception_v3'


def main():
```

```
    print("starting program . . .")

    # make sure the logging output is visible, see
https://github.com/tensorflow/tensorflow/issues/
3047

    tf.logging.set_verbosity(tf.logging.INFO)



    if not checkIfNecessaryPathsAndFilesExist():

        return

    # end if



    # 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(ARCHITECTURE)

    if not model_info:

        tf.logging.error('Did not recognize architecture
flag')

        return -1

    # end if



    # download the model if necessary, then create
the model graph

    print("downloading model (if necessary) . . .")


downloadModelIfNotAlreadyPresent(model_info['
data_url'])

    print("creating model graph . . .")

    graph, bottleneck_tensor, resized_image_tensor
= (create_model_graph(model_info))



    # Look at the folder structure, and create lists of
all the images.

    print("creating image lists . . .")

    image_lists =
create_image_lists(TRAINING_IMAGES_DIR,
TESTING_PERCENTAGE,
VALIDATION_PERCENTAGE)

    class_count = len(image_lists.keys())

    if class_count == 0:

        tf.logging.error('No valid folders of images
found at ' + TRAINING_IMAGES_DIR)

        return -1

    # end if

    if class_count == 1:

        tf.logging.error('Only one valid folder of
images found at ' + TRAINING_IMAGES_DIR + ' -
multiple classes are needed for classification.')

        return -1

    # end if



    # determinf if any of the distortion command
line flags have been set

    doDistortImages = False

    if (FLIP_LEFT_RIGHT == True or RANDOM_CROP
!= 0 or RANDOM_SCALE != 0 or
RANDOM_BRIGHTNESS != 0):

        doDistortImages = True

    # end if



    print("starting session . . .")

    with tf.Session(graph=graph) as sess:

        # Set up the image decoding sub-graph.

        print("performing jpeg decoding . . .")

        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'])

        print("caching bottlenecks . . .")
```

```
    distorted_jpeg_data_tensor = None

    distorted_image_tensor = None

    if doDistortImages:

      # We will be applying distortions, so setup
the operations we'll need.

      (distorted_jpeg_data_tensor,
distorted_image_tensor) =
add_input_distortions(FLIP_LEFT_RIGHT,
RANDOM_CROP, RANDOM_SCALE,


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,
TRAINING_IMAGES_DIR, BOTTLENECK_DIR,
jpeg_data_tensor, decoded_image_tensor,

                  resized_image_tensor,
bottleneck_tensor, ARCHITECTURE)

    # end if



    # Add the new layer that we'll be training.

    print("adding final training layer . . .")

    (train_step, cross_entropy, bottleneck_input,
ground_truth_input, final_tensor) =
add_final_training_ops(len(image_lists.keys()),


FINAL_TENSOR_NAME,


bottleneck_tensor,


model_info['bottleneck_tensor_size'],


model_info['quantize_layer'])

    # Create the operations we need to evaluate
the accuracy of our new layer.

    print("adding eval ops for final training layer . .
.")

    evaluation_step, prediction =
add_evaluation_step(final_tensor,
ground_truth_input)



    # Merge all the summaries and write them out
to the tensorboard_dir

    print("writing TensorBoard info . . .")

    merged = tf.summary.merge_all()

    train_writer =
tf.summary.FileWriter(TENSORBOARD_DIR +
'/train', sess.graph)

    validation_writer =
tf.summary.FileWriter(TENSORBOARD_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.

    print("performing training . . .")

    for i in range(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 doDistortImages:

        (train_bottlenecks, train_ground_truth) =
get_random_distorted_bottlenecks(sess,
image_lists, TRAIN_BATCH_SIZE, 'training',


TRAINING_IMAGES_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, TRAIN_BATCH_SIZE, 'training',

BOTTLENECK_DIR, TRAINING_IMAGES_DIR,
jpeg_data_tensor,

decoded_image_tensor, resized_image_tensor,
bottleneck_tensor,

ARCHITECTURE)

        # end if

        # 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 ==
HOW_MANY_TRAINING_STEPS)

        if (i % 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, VALIDATION_BATCH_SIZE, 'validation',

BOTTLENECK_DIR, TRAINING_IMAGES_DIR,
jpeg_data_tensor,

decoded_image_tensor, resized_image_tensor,
bottleneck_tensor,

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_summa
ry, i)

            tf.logging.info('%s: Step %d: Validation
accuracy = %.1f%% (N=%d)' % (datetime.now(), i,
validation_accuracy * 100,
len(validation_bottlenecks)))

        # end if

        # Store intermediate results

        intermediate_frequency =
INTERMEDIATE_STORE_FREQUENCY

        if (intermediate_frequency > 0 and (i %
intermediate_frequency == 0) and i > 0):

            intermediate_file_name =
(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)

        # end if

    # end for

    # We've completed all our training, so run a
final test evaluation on some new images we
haven't used before

    print("running testing . . .")

    test_bottlenecks, test_ground_truth,
test_filenames =
```

```
(get_random_cached_bottlenecks(sess,
image_lists, TEST_BATCH_SIZE, 'testing',
BOTTLENECK_DIR,


TRAINING_IMAGES_DIR, jpeg_data_tensor,
decoded_image_tensor, resized_image_tensor,


bottleneck_tensor, 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 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]:

               tf.logging.info('%70s  %s' %
(test_filename,
list(image_lists.keys())[predictions[i]]))

            # end if

         # end for

      # end if



      # write out the trained graph and labels with
the weights stored as constants

      print("writing trained graph and labbels with
weights")

      save_graph_to_file(sess, graph,
OUTPUT_GRAPH)

      with gfile.FastGFile(OUTPUT_LABELS, 'w') as f:

         f.write('\n'.join(image_lists.keys()) + '\n')

      # end with



      print("done !!")



def checkIfNecessaryPathsAndFilesExist():

   # if the training directory does not exist, show
and error message and bail

   if not os.path.exists(TRAINING_IMAGES_DIR):

      print('')

      print('ERROR: TRAINING_IMAGES_DIR "' +
TRAINING_IMAGES_DIR + '" does not seem to
exist')

      print('Did you set up the training images?')

      print('')

      return False

   # end if



   # nested class

   class TrainingSubDir:

      # constructor

      def __init__(self):

         self.loc = ""

         self.numImages = 0

      # end constructor

   # end class



   # declare a list of training sub-directories

   trainingSubDirs = []



   # populate the training sub-directories

   for dirName in
os.listdir(TRAINING_IMAGES_DIR):

      currentTrainingImagesSubDir =
os.path.join(TRAINING_IMAGES_DIR, dirName)

      if os.path.isdir(currentTrainingImagesSubDir):

         trainingSubDir = TrainingSubDir()

         trainingSubDir.loc =
currentTrainingImagesSubDir

         trainingSubDirs.append(trainingSubDir)

      # end if

   # end for
```

```
    # if no training sub-directories were found, show an error message and return false
    if len(trainingSubDirs) == 0:
        print("ERROR: there don't seem to be any training image sub-directories in " + TRAINING_IMAGES_DIR)
        print("Did you make a separare image sub-directory for each classification type?")
        return False
    # end if

    # populate the number of training images in each training sub-directory
    for trainingSubDir in trainingSubDirs:
        # count how many images are in the current training sub-directory
        for fileName in os.listdir(trainingSubDir.loc):
            if fileName.endswith(".jpg"):
                trainingSubDir.numImages += 1

    # if any training sub-directory has less than the min required number of training images, show an error message and return false
    for trainingSubDir in trainingSubDirs:
        if trainingSubDir.numImages < MIN_NUM_IMAGES_REQUIRED_FOR_TRAINING:
            print("ERROR: there are less than the required " + str(MIN_NUM_IMAGES_REQUIRED_FOR_TRAINING) + " images in " + trainingSubDir.loc)
            print("Did you populate each training sub-directory with images?")
            return False
        # end if
    # end for

    # if any training sub-directory has less than the recommended number of training images, show a warning (but don't return false)
    for trainingSubDir in trainingSubDirs:
        if trainingSubDir.numImages < MIN_NUM_IMAGES_SUGGESTED_FOR_TRAINING:
            print("WARNING: there are less than the suggested " + str(MIN_NUM_IMAGES_SUGGESTED_FOR_TRAINING) + " images in " + trainingSubDir.loc)
            print("More images should be added to this directory for acceptable training results")
            # note we do not return false here b/c this is a warning, not an error

    # if the test images directory does not exist, show and error message and bail
    if not os.path.exists(TEST_IMAGES_DIR):
        print('')
        print('ERROR: TEST_IMAGES_DIR "' + TEST_IMAGES_DIR + '" does not seem to exist')
        print('Did you break out some test images?')
        print('')
        return False
    # end if

    # count how many images are in the test images directory
    numImagesInTestDir = 0
    for fileName in os.listdir(TEST_IMAGES_DIR):
        if fileName.endswith(".jpg"):
            numImagesInTestDir += 1

    # if there are not enough images in the test images directory, show an error and return false
    if numImagesInTestDir < MIN_NUM_IMAGES_REQUIRED_FOR_TESTING:
        print("ERROR: there are not at least " + str(MIN_NUM_IMAGES_REQUIRED_FOR_TESTING) + " images in " + TEST_IMAGES_DIR)
```

```
      print("Did you break out some test images?")
      return False
  return True


def prepare_file_system():
  # Setup the directory we'll write summaries to
for TensorBoard
  if tf.gfile.Exists(TENSORBOARD_DIR):
    tf.gfile.DeleteRecursively(TENSORBOARD_DIR)
  tf.gfile.MakeDirs(TENSORBOARD_DIR)
  if INTERMEDIATE_STORE_FREQUENCY > 0:

makeDirIfDoesNotExist(INTERMEDIATE_OUTPUT_
GRAPHS_DIR)
  return


def makeDirIfDoesNotExist(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 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()
  is_quantized = False
  if architecture == 'inception_v3':
    # pylint: disable=line-too-long
    data_url =
'http://download.tensorflow.org/models/image/i
magenet/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
```

```
    # end if

    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

    # end if

    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



    if is_quantized:

      data_url =
'http://download.tensorflow.org/models/mobilene
t_v1_'

      data_url += version_string + '_' + size_string
+ '_quantized_frozen.tgz'

      bottleneck_tensor_name =
'MobilenetV1/Predictions/Reshape:0'

      resized_input_tensor_name =
'Placeholder:0'

      model_dir_name = ('mobilenet_v1_' +
version_string + '_' + size_string +
'_quantized_frozen')

      model_base_name =
'quantized_frozen_graph.pb'

    else:

      data_url =
'http://download.tensorflow.org/models/mobilene
t_v1_'

      data_url += version_string + '_' + size_string
+ '_frozen.tgz'

      bottleneck_tensor_name =
'MobilenetV1/Predictions/Reshape:0'

      resized_input_tensor_name = 'input:0'

      model_dir_name = 'mobilenet_v1_' +
version_string + '_' + size_string

      model_base_name = 'frozen_graph.pb'



    bottleneck_tensor_size = 1001

    input_width = int(size_string)

    input_height = int(size_string)

    input_depth = 3

    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,
'quantize_layer': is_quantized, }
```

```
def downloadModelIfNotAlreadyPresent(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 = MODEL_DIR
    if not os.path.exists(dest_directory):
        os.makedirs(dest_directory)
    # end if
    filename = data_url.split('/')[-1]
    filepath = os.path.join(dest_directory, filename)
    if not os.path.exists(filepath):
        # nested function
        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()
        # end def

        filepath, _ = urllib.request.urlretrieve(data_url, filepath, _progress)
        print()
        statinfo = os.stat(filepath)
        tf.logging.info('Successfully downloaded ' + str(filename) + ', statinfo.st_size = ' + str(statinfo.st_size) + ' bytes')
        print('Extracting file from ', filepath)
        tarfile.open(filepath, 'r:gz').extractall(dest_directory)
    else:
        print('Not extracting or downloading files, model already present in disk')

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(MODEL_DIR, model_info['model_file_name'])
        print('Model path: ', model_path)
        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 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 the image directory does not exist, log an error and bail

  if not gfile.Exists(image_dir):

    tf.logging.error("Image directory '" + image_dir + "' not found.")

    return None

  # end if



  # create an empty dictionary to store the results

  result = {}



  # get a list of the sub-directories of the image directory

  sub_dirs = [x[0] for x in gfile.Walk(image_dir)]



  # for each directory in the sub-directories list . . .

  is_root_dir = True

  for sub_dir in sub_dirs:

    # if we're on the 1st (root) directory, mark our boolean for that as false for the next time around and go back to the top of the for loop

    if is_root_dir:

      is_root_dir = False

      continue

    # end if



    dir_name = os.path.basename(sub_dir)

    if dir_name == image_dir:

      continue

    # end if



    # ToDo: This section should be refactored.  The right way to do this would be to get a list of the files that are

    # ToDo: there then append (extend) those, not to get the name except the extension, then append an extension,

    # ToDo: this (current) way is error prone of the original file has an upper case or mixed case extension



    extensions = ['jpg', 'jpeg']

    file_list = []

    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))

    # end for



    # if the file list is empty at this point, log a warning and bail
```

```
    if not file_list:
      tf.logging.warning('No files found')
      continue
    # end if

    # if the length of the file list is less than 20 or more than the max number, log an applicable warning (do not return, however)
    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))
    # end if

    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 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 add_input_distortions(flip_left_right,
random_crop, random_scale, random_brightness,
input_width, input_height,

                  input_depth, input_mean,
input_std):




  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

  # end if

  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 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

  makeDirIfDoesNotExist(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)

      # end for

      how_many_bottlenecks += 1

      if how_many_bottlenecks % 100 == 0:

tf.logging.info(str(how_many_bottlenecks) + '
bottleneck files created.')

      # end if



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)

  makeDirIfDoesNotExist(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)

  # end if

  # read in the contents of the bottleneck file as
one big string

  with open(bottleneck_path, 'r') as
bottleneck_file:

    bottleneckBigString = bottleneck_file.read()

  # end with



  bottleneckValues = []

  errorOccurred = False

  try:

    # split the bottleneck file contents read in as
one big string into individual float values

    bottleneckValues = [float(individualString) for
individualString in bottleneckBigString.split(',')]

  except ValueError:

    tf.logging.warning('Invalid float found,
recreating bottleneck')

    errorOccurred = True

  # end try



  if errorOccurred:

    # if an error occurred above, create (or re-
create) the bottleneck file

    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)



    # read in the contents of the newly created
bottleneck file

    with open(bottleneck_path, 'r') as
bottleneck_file:

      bottleneckBigString = bottleneck_file.read()

    # end with



    # split the bottleneck file contents read in as
one big string into individual float values again
```

```
    bottleneckValues = [float(individualString) for individualString in bottleneckBigString.split(',')]

  # end if

  return bottleneckValues



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_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)

  # end if

  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)))

  # end try



  bottleneck_string = ','.join(str(x) for x in bottleneck_values)

  with open(bottleneck_path, 'w') as bottleneck_file:

    bottleneck_file.write(bottleneck_string)



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 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)

   # end if

   label_lists = image_lists[label_name]

   if category not in label_lists:

      tf.logging.fatal('Category does not exist %s.',
category)

   # end if

   category_list = label_lists[category]

   if not category_list:

      tf.logging.fatal('Label %s has no images in the
category %s.', label_name, category)

   # end if

   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 add_final_training_ops(class_count,
final_tensor_name, bottleneck_tensor,
bottleneck_tensor_size, quantize_layer):

   """

   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/tuto
rials/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.

    quantize_layer: Boolean, specifying whether
the newly added layer should be quantized.


  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.int64,
[None], name='GroundTruthInput')

  # end with


  # 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):

    quantized_layer_weights = None

    quantized_layer_biases = None

    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')

      if quantize_layer:

        quantized_layer_weights =
quant_ops.MovingAvgQuantize(layer_weights,
is_training=True)

attachTensorBoardSummaries(quantized_layer_we
ights)

      # end if


      # this comment is necessary to suppress an
unnecessary PyCharm warning

      # noinspection PyTypeChecker

attachTensorBoardSummaries(layer_weights)

    # end with

    with tf.name_scope('biases'):

      layer_biases =
tf.Variable(tf.zeros([class_count]),
name='final_biases')

      if quantize_layer:

        quantized_layer_biases =
quant_ops.MovingAvgQuantize(layer_biases,
is_training=True)

attachTensorBoardSummaries(quantized_layer_bia
ses)

      # end if


      # this comment is necessary to suppress an
unnecessary PyCharm warning

      # noinspection PyTypeChecker

      attachTensorBoardSummaries(layer_biases)

    # end with

    with tf.name_scope('Wx_plus_b'):

      if quantize_layer:
```

```
            logits = tf.matmul(bottleneck_input,
quantized_layer_weights) +
quantized_layer_biases

            logits =
quant_ops.MovingAvgQuantize(logits, init_min=-
32.0, init_max=32.0, is_training=True, num_bits=8,

                                        narrow_range=False,
ema_decay=0.5)

            tf.summary.histogram('pre_activations',
logits)

          else:

            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_mean =
tf.losses.sparse_softmax_cross_entropy(labels=gro
und_truth_input, logits=logits)



    tf.summary.scalar('cross_entropy',
cross_entropy_mean)



    with tf.name_scope('train'):

        optimizer =
tf.train.GradientDescentOptimizer(LEARNING_RAT
E)

        train_step =
optimizer.minimize(cross_entropy_mean)



    return (train_step, cross_entropy_mean,
bottleneck_input, ground_truth_input,
final_tensor)



def attachTensorBoardSummaries(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)))

        # end with

        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_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,
ground_truth_tensor)

        # end with

        with tf.name_scope('accuracy'):
```

```
        evaluation_step =
tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))

    # end with

  tf.summary.scalar('accuracy', evaluation_step)

  return evaluation_step, prediction



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_CLAS
S + 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)

    # end if

    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)

    bottlenecks.append(bottleneck_values)

    ground_truths.append(label_index)

  return bottlenecks, ground_truths
```

```
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_CLAS
S + 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)

      bottlenecks.append(bottleneck)

      ground_truths.append(label_index)

      filenames.append(image_name)

    # end for

  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)

            bottlenecks.append(bottleneck)

            ground_truths.append(label_index)

            filenames.append(image_name)

    return bottlenecks, ground_truths, filenames


def save_graph_to_file(sess, graph,
graph_file_name):

    output_graph_def =
graph_util.convert_variables_to_constants(sess,
graph.as_graph_def(), [FINAL_TENSOR_NAME])

    with gfile.FastGFile(graph_file_name, 'wb') as f:

        f.write(output_graph_def.SerializeToString())

    # end with

    return


if __name__ == '__main__':

    main()
```

En este otro código es donde se prueba la base de datos usando imágenes fuera de las carpetas ya estudiadas por el programa. Al igual que el código anterior esta marcado con comentarios para su mejor comprensión y entendimiento.

```
# test.py

import os

import tensorflow as tf

import numpy as np

import cv2



RETRAINED_LABELS_TXT_FILE_LOC = os.getcwd() +
"/" + "retrained_labels.txt"

RETRAINED_GRAPH_PB_FILE_LOC = os.getcwd() +
"/" + "retrained_graph.pb"


TEST_IMAGES_DIR = os.getcwd() + "/test_images"


SCALAR_RED = (0.0, 0.0, 255.0)

SCALAR_BLUE = (255.0, 0.0, 0.0)




def main():

    print("starting program . . .")


    if not checkIfNecessaryPathsAndFilesExist():

        return

    # end if


    # get a list of classifications from the labels file

    classifications = []

    # for each line in the label file . . .

    for currentLine in
tf.gfile.GFile(RETRAINED_LABELS_TXT_FILE_LOC):

        # remove the carriage return

        classification = currentLine.rstrip()

        # and append to the list

        classifications.append(classification)

    # end for


    # show the classifications to prove out that we
were able to read the label file successfully

    print("classifications = " + str(classifications))


    # load the graph from file

    with
tf.gfile.FastGFile(RETRAINED_GRAPH_PB_FILE_LOC
, 'rb') as retrainedGraphFile:

        # instantiate a GraphDef object

        graphDef = tf.GraphDef()
```

```
    # read in retrained graph into the GraphDef object

graphDef.ParseFromString(retrainedGraphFile.read())

    # import the graph into the current default Graph, note that we don't need to be concerned with the return value

    _ = tf.import_graph_def(graphDef, name='')

  # end with


  # if the test image directory listed above is not valid, show an error message and bail

  if not os.path.isdir(TEST_IMAGES_DIR):

    print("the test image directory does not seem to be a valid directory, check file / directory paths")

    return

  # end if


  with tf.Session() as sess:

    # for each file in the test images directory . . .

    for fileName in os.listdir(TEST_IMAGES_DIR):

      # if the file does not end in .jpg or .jpeg (case-insensitive), continue with the next iteration of the for loop

      if not (fileName.lower().endswith(".jpg") or fileName.lower().endswith(".jpeg")):

        continue

      # end if


      # show the file name on std out

      print(fileName)


      # get the file name and full path of the current image file

      imageFileWithPath = os.path.join(TEST_IMAGES_DIR, fileName)

      # attempt to open the image with OpenCV

      openCVImage = cv2.imread(imageFileWithPath)


      # if we were not able to successfully open the image, continue with the next iteration of the for loop

      if openCVImage is None:

        print("unable to open " + fileName + " as an OpenCV image")

        continue

      # end if


      # get the final tensor from the graph

      finalTensor = sess.graph.get_tensor_by_name('final_result:0')


      # convert the OpenCV image (numpy array) to a TensorFlow image

      tfImage = np.array(openCVImage)[:, :, 0:3]


      # run the network to get the predictions

      predictions = sess.run(finalTensor, {'DecodeJpeg:0': tfImage})


      # sort predictions from most confidence to least confidence

      sortedPredictions = predictions[0].argsort()[-len(predictions[0]):][::-1]


      print("----------------------------------------")


      # keep track of if we're going through the next for loop for the first time so we can show more info about

      # the first prediction, which is the most likely prediction (they were sorted descending above)

      onMostLikelyPrediction = True

      # for each prediction . . .
```

```
        for prediction in sortedPredictions:

            strClassification =
classifications[prediction]



            # if the classification (obtained from the
directory name) ends with the letter "s", remove
the "s" to change from plural to singular

            if strClassification.endswith("s"):

                strClassification = strClassification[:-1]

            # end if



            # get confidence, then get confidence
rounded to 2 places after the decimal

            confidence = predictions[0][prediction]



            # if we're on the first (most likely)
prediction, state what the object appears to be and
show a % confidence to two decimal places

            if onMostLikelyPrediction:

                # get the score as a %

                scoreAsAPercent = confidence * 100.0

                # show the result to std out

                print("the object appears to be a " +
strClassification + ", " +
"{0:.2f}".format(scoreAsAPercent) + "%
confidence")

                # write the result on the image

                writeResultOnImage(openCVImage,
strClassification + ", " +
"{0:.2f}".format(scoreAsAPercent) + "%
confidence")

                # finally we can show the OpenCV
image

                cv2.imshow(fileName, openCVImage)

                print("Guardar Imagen?")

                # mark that we've show the most likely
prediction at this point so the additional
information in

                # this if statement does not show again
for this image

                onMostLikelyPrediction = False

            # end if



            # for any prediction, show the confidence
as a ratio to five decimal places

            print(strClassification + " (" +
"{0:.5f}".format(confidence) + ")")

        # end for



        # pause until a key is pressed so the user can
see the current image (shown above) and the
prediction info

        cv2.waitKey()

        # after a key is pressed, close the current
window to prep for the next time around

        cv2.destroyAllWindows()

    # end for

# end with



# write the graph to file so we can view with
TensorBoard

tfFileWriter =
tf.summary.FileWriter(os.getcwd())

tfFileWriter.add_graph(sess.graph)

tfFileWriter.close()




def checkIfNecessaryPathsAndFilesExist():

    if not os.path.exists(TEST_IMAGES_DIR):

        print('')

        print('ERROR: TEST_IMAGES_DIR "' +
TEST_IMAGES_DIR + '" does not seem to exist')

        print('Did you set up the test images?')

        print('')

        return False

    # end if
```

```
    if not os.path.exists(RETRAINED_LABELS_TXT_FILE_LOC):
        print('ERROR: RETRAINED_LABELS_TXT_FILE_LOC "' + RETRAINED_LABELS_TXT_FILE_LOC + '" does not seem to exist')
        return False
    # end if

    if not os.path.exists(RETRAINED_GRAPH_PB_FILE_LOC):
        print('ERROR: RETRAINED_GRAPH_PB_FILE_LOC "' + RETRAINED_GRAPH_PB_FILE_LOC + '" does not seem to exist')
        return False
    # end if

    return True

def writeResultOnImage(openCVImage, resultText):
    # ToDo: this function may take some further fine-tuning to show the text well given any possible image size

    imageHeight, imageWidth, sceneNumChannels = openCVImage.shape

    # choose a font
    fontFace = cv2.FONT_HERSHEY_TRIPLEX
    # chose the font size and thickness as a fraction of the image size
    fontScale = 1.0
    fontThickness = 2

    # make sure font thickness is an integer, if not, the OpenCV functions that use this may crash
    fontThickness = int(fontThickness)

    upperLeftTextOriginX = int(imageWidth * 0.05)
    upperLeftTextOriginY = int(imageHeight * 0.05)

    textSize, baseline = cv2.getTextSize(resultText, fontFace, fontScale, fontThickness)
    textSizeWidth, textSizeHeight = textSize

    # calculate the lower left origin of the text area based on the text area center, width, and height
    lowerLeftTextOriginX = upperLeftTextOriginX
    lowerLeftTextOriginY = upperLeftTextOriginY + textSizeHeight

    # write the text on the image
    cv2.putText(openCVImage, resultText, (lowerLeftTextOriginX, lowerLeftTextOriginY), fontFace, fontScale, SCALAR_BLUE, fontThickness)

if __name__ == "__main__":
    main()
```

Este proyecto si se implementa con tanta información e imágenes puede llegar a distinguir casi cualquier cosa, al igual que se puede hacer en un app para usar el programa a tiempo real, es por eso de su amplio espectro de escalabilidad. Lo bueno de este proyecto es que solo necesita dos códigos, pero se puede convertir en uno si el usuario sabe hacer todo el proceso antes. El programa se puede personalizar a gustos de los usuarios al igual que este enfocado a un objetivo más directo. Por ultimo este programa puede tener una gran seguridad de usuario al igual que ser anónimo en su uso para la discreción de los clientes.

En conclusión, el proyecto para la clasificación de imágenes es muy simple de usar y efectivo, la único que si se necesita es el tiempo de recolección de imágenes, tener la paciencia para que el programa aprende a diferenciarlos y al probarlo.

Referencias:

https://www.apsl.net/blog/2017/12/05/tensor-flow-para-principiantes-i/

https://www.youtube.com/watch?v=szNPBn_RBfA

https://www.youtube.com/watch?v=90gpNF3KzK8&t=72s