PROJECT VIDEO LINK: https://drive.google.com/file/d/1iPrqf9VPN1fX7Olq9PFFqMCqgcrfNKXi/view?usp=sharing

PROJECT REPORT

Skin Disease Classification Using Convolutional Neural Networks

Submitted for the course

CSE4020 - Machine Learning

Slot: E1

Team members:

Vignesh Vaidyanathan	**-**	**15BCE0076**
Voleti Ravi	**-**	**15BCE0082**
Samudra Pratim Borkakoti	**-**	**15BCE0093**
Mohammed Fardeen Shaik	**-**	**15BCE0180**

Submitted to

Prof. Manoov R, SCOPE

School of Computer Science and Engineering

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

April, 2018

CONTENTS

ABSTRACT:

Skin diseases are very common in people's daily life. Each year, millions of people in American are affected by all kinds of skin disorders. Diagnosis of skin diseases sometimes requires a high-level of expertise due to the variety of their visual aspects. As human judgment are often subjective and hardly reproducible, to achieve a more objective and reliable diagnosis, a computer aided diagnostic system should be considered. In this paper, we investigate the feasibility of constructing a universal skin disease diagnosis system using deep convolutional neural network (CNN). We train the CNN architecture using the 23,000 skin disease images from the Dermnet dataset and test its performance with both the Dermnet and another skin disease dataset which would be a raw image uploaded to the system.

In this project, we have tried to use machine learning models for classification using images as inputs and getting confidence level scores as output based on its classification. The model used is Convolutional Neural Networks (CNN) model for image analysis and classification. The architecture of the model is a standard architecture used in many industries called Inception Model. This model gives improved utilization model of the computing resources inside the network. Our system can achieve an accuracy of 84%.

Introduction:

The use of computer technology in medical decision support is now widespread and pervasive across a wide range of medical area, such as cancer research, dermatology, etc. Data mining is a tremendous opportunity to assist physician deal with this large amount of data. Its methods can help physicians in various ways such as interpreting complex diagnostic tests, combining information from multiple sources (sample movies, images, clinical data, proteomics and scientific knowledge), providing support for differential diagnosis and providing patient-specific prognosis. Skin diseases are one of the most commonly seen infections among people. Due to the disfigurement and associated hardships, skin disorders cause lots of trouble to the sufferers [1]. In each year, the number of new cases of skin cancer is more than the number of the new incidence of cancers of the breast, prostate, lung and colon in combined [2].

However, the diagnosis of skin disease is challenging. To diagnose a skin disease, a variety of visual clues may been used such as the individual lesional morphology, the body site distribution, colour, scaling and arrangement of lesions. When the individual components are analysed separately, the recognition process can be quite complex [3]. For example, the well-studied skin cancer, melanoma, has four major clinical diagnosis methods: ABCD rules, pattern analysis, Menzies method and 7-Point Checklist. To use these methods and achieve a good diagnostic accuracy, a high level of expertise is required as the differentiation of skin lesions need a great deal of experience [4].

Software requirements:

- Python 3.0
- Library packages
    - CNN module
    - Inception module
    - OpenCV
    - Tensor flow

Literature Review

Looking into the current situations of computerized skin disease diagnosis systems, there are few solutions available which are still under research developments. Certain limitations and drawbacks are identified in those hence this solution tries to overcome the existing problems with different approach. [6][7]

In the paper "Detection of Malignant Skin Diseases Based on the Lesion Segmentation" by Jyothilakshmi K. K, Jeeva J. B, a novel technique to automatically detect the malignancy of skin diseases using conventional camera images is proposed. The procedure used would be of great advantage to the dermatologists as a pre-screening system for early diagnosis in situations where the dermoscopes are not accessible. This algorithm mainly aims at an early diagnosis of the malignant diseases since they can be cured if detected early. The proposed method works on colour images by taking the HSV component and pre-processing was performed. A robust segmentation procedure is performed for the accurate detection of the lesion. For detection purpose, the morphological features like asymmetry, border irregularity, colour variation and diameter are used. These extracted features help to identify the malignant lesions from the non-malignant ones [7].

In the paper, Skin disease classification using texture analysis (ANN), this research describes skin disease recognition by using neural network which based on the texture analysis. There are many skin diseases which have a lot of similarities in their symptoms, such as Measles (rubeola), German measles (rubella), and Chickenpox etc. In general, these diseases have similarities in pattern of infection and symptoms such as redness and rash. Diagnosis and recognition of skin disease take a very long term process because it requires patient's history, physical examination and proper laboratory diagnostic tests. Not only that, it also requires large number of features clinical as well as histopathological for analysis and to provide further treatment. The disease diagnosis and recognition becomes difficult as the complexity and number of features of the disease increases. Hence, a computer aided diagnosis and recognition system is introduced. Computer algorithm which contains few steps that involves image processing, image feature extraction and classification of data have been implemented with the help of classifier such as artificial neural network (ANN). The ANN can learn patterns of symptoms of particular diseases and provides faster diagnosis and recognition than a human physician. Thus, the patients can do the treatment for the skin disease faced immediately based on the symptoms detected.

In the paper "Survey of Texture Based Feature Extraction for Skin Disease Detection" by Seema Kolkur, D.R. Kalbande, [5] an approach of texture based feature extraction for detection of skin diseases has been presented to resolve issues. In statistical texture analysis, texture features are computed from the statistical distribution of observed combinations of intensities at specified positions relative to each other in the image. According to the number of intensity points (pixels) in each combination, statistics are classified into first-order, second-order and higher-order statistics. GLCM method is a way of extracting second order statistical texture features. Third and higher order textures consider the relationships among three or more pixels. These are theoretically possible but not commonly implemented due to calculation time and interpretation difficulty. In this paper, work on texture based features derived from GLCM matrix used for the detection of skin diseases is discussed and consolidated. Many researchers have used additional features along with texture based features to improve accuracy of classification.

In the paper "An Intelligent Decision Support System for Skin Cancer Detection from Dermoscopic Images", [10] the proposed system employs pre-processing such as dull razors and median filters to remove hair and other noise. Then, the images were segmented using a pixel limitation technique to separate lesions from image background. Feature extraction is subsequently conducted. The features extracted by the system reflect the well-known asymmetry, border irregularity, color variegation and diameter (ABCD) of dermatology and the epiluminescence microscopy (ELM) criteria. They focus primarily on the size, shape, color and local parameters of lesions with some additional consideration of the lesion edges. Evaluated with 100 images from the Edinburgh Research and Innovation (Dermofit) dataset, this work achieves an average accuracy of 92% and 84% for benign and malignant skin lesion classification.

Most of the work is carried out detection of skin cancer but other diseases like psoriasis, warts, moles, eczema are also considered in some of the works. Most the research shows overall accuracy around or above 90%. Top five features used in all this work are Contrast, Correlation, Energy, Entropy and homogeneity. Skin Disease Detection System based on these findings is proposed.

Drawback of existing systems:

Most of the existing systems do classify on a neural network basis where the AI is made to think and decide with the artificial neurons provided but rather uses a probability measure such as NAÏVE BAYES and assumes that is the correct one. The pre-processing of the image is also becomes an issue where suppose in the case of skin cancer, a grayscale pre-processing is completely unrequired and irrelevant.

Developed model:

In this proposed system, we are considering a train of images that will be obtained from the user and pre-processing and segmentation will be performed on each image. Then feature extraction is done on each image to extract features that can be used to create classification model. The system then provides a confidence score as to how likely it is which disease a person has got.

Dataset

We build our skin disease dataset from Dermnet. Dermnet is one of the largest photo dermatology source that available publicly. It has more than 23,000 skin disease images on a wide variety of skin conditions, out of which we have chosen 2 of them. One being melanoma skin cancer and the other being herpes. Dermnet organizes the skin diseases biologically in a two-level taxonomy. The bottom-level contains more than 600 skin diseases in a fine-grained granularity.

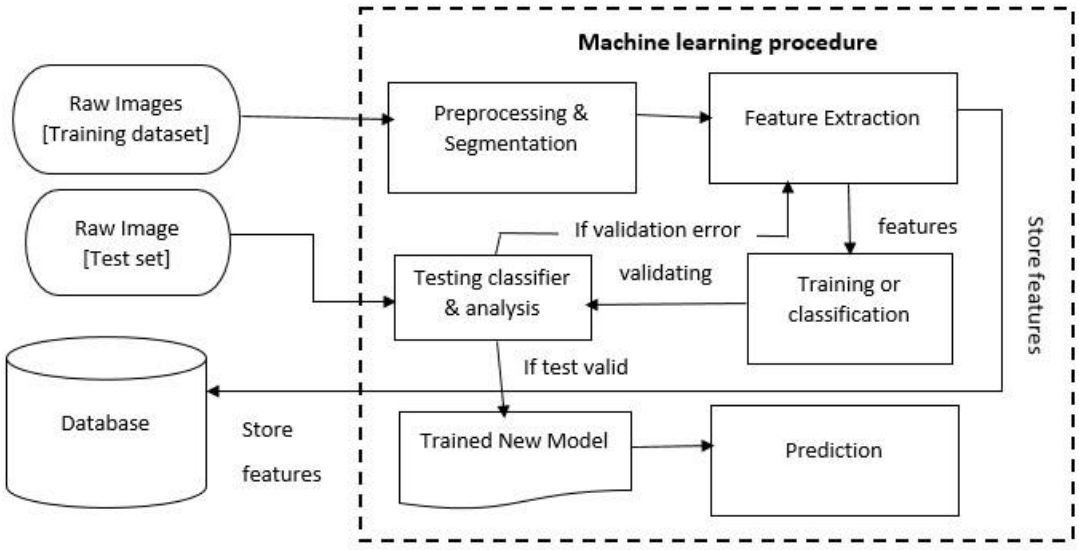

Figure 1. Basic system architecture

CNN Architecture:

It has been shown that in many cases transfer learning can be used to efficiently train a deep CNN. In transfer learning, instead of training the network from randomly initialized parameters, people takes a pretrained network and fine-tunes its weights by continuing the back- propagation. This works for the reason that the output of the early layers of a well-trained network usually contains some generic features. Those generic features such as edges, blobs can be very useful in many tasks. For a new dataset, those features can be applied directly. In our approach, we do transfer learning by fine-tuning ImageNet pre-trained models with inception a deep learning framework that supports expressive and efficient deep CNN training. The primary dataset was prepared for analysis by scaling all features to values between 0 and 1 and converting the set to a classification dataset format. The input layer of the network consisted of 22 neurons, one for each of the scaled features. There was one linear hidden layer, with 13 neurons. The output layer was a single classification neuron with a sigmoid activation function. A back-propagation trainer was created with a learning rate and momentum that were varied from 0.01 to 1 in increments of
0.01 to determine optimal parameter values.

Pre-processing:

Image pre-processing is an essential step of detection in order to remove noise such as hair clothing and other artifacts and enhance the quality of original image. The main purpose of this step is to improve the quality of skin image by removing unrelated and surplus parts in the back ground of image for further processing. Good selection of pre-processing techniques can greatly improve the accuracy of the system. The objective of the pre-processing stage can be achieved through three process stages of image enhancement, image restoration and hair removal. We are pre-processing the image by converting it to grayscale, reducing its dimension and cropping it to a standard 100x100 image irrespective of its original size.

Inception model:

The module basically acts as multiple convolution filters that are applied to the same input, with some pooling. The results are then concatenated. This allows the model to take advantage of multi-level feature extraction. For instance, it extracts general (5x5) and local (1x1) features at the same time. This means instead of adding a particular filter size layer, we add all 1x1, 3x3, 5x5 filters and perform convolution on the output from the previous layers. Since pooling has been essential for the success of current CNNs, the inception module also includes an additional pooling path. The output of all the filters are concatenated and passed on as input to the next layer.

So the inception model helps in pre-processing as well by reducing the dimensions and also rendering the need for individual neurons needless.

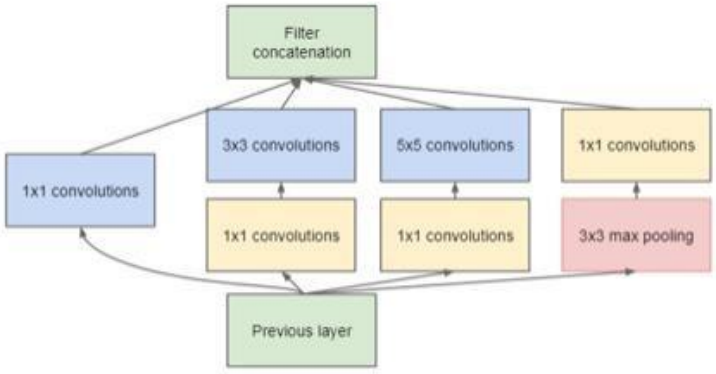

Figure 2. Inception model for dimension reduction architecture

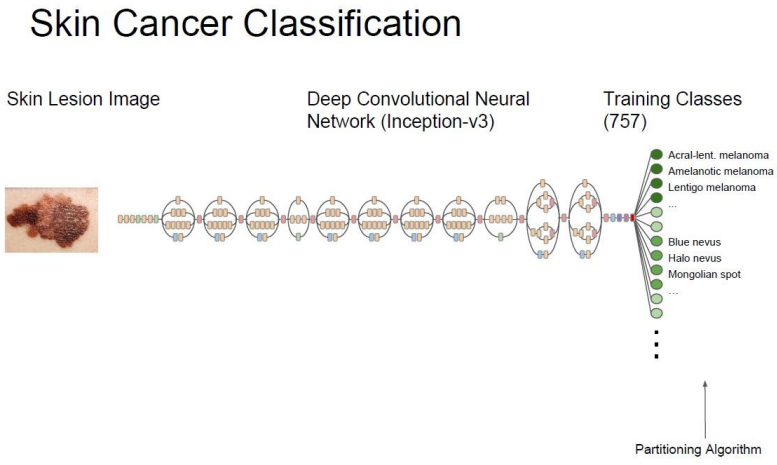

Figure 3. Another representation of inception model

Implementation:

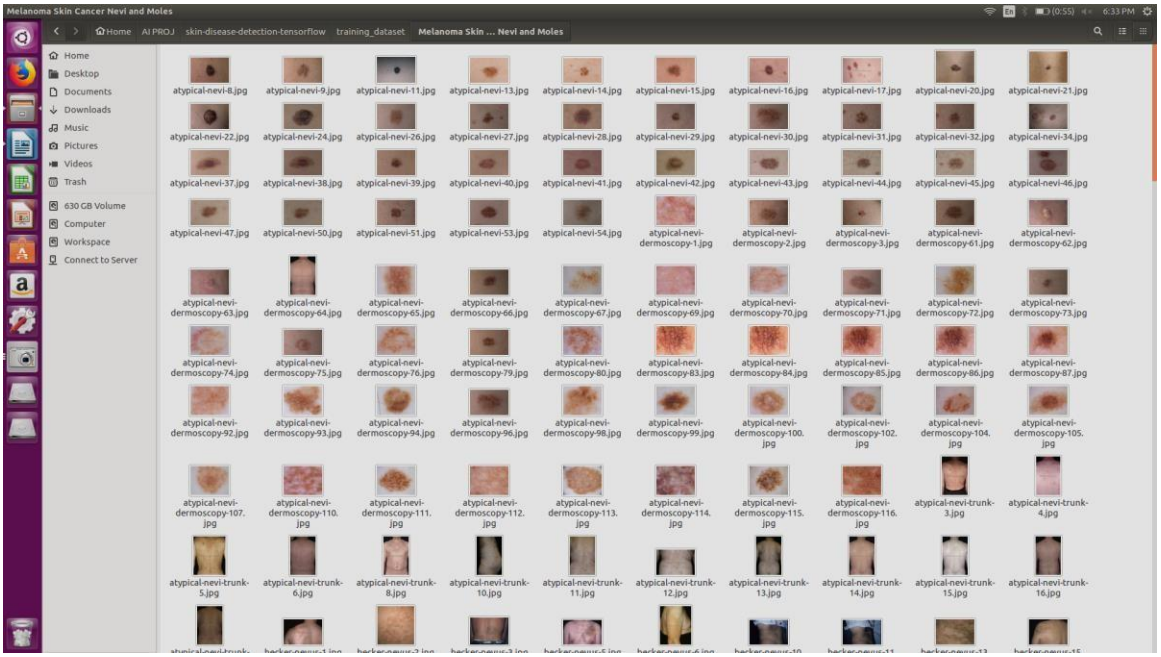

Figure 3. Melanoma training and testing dataset

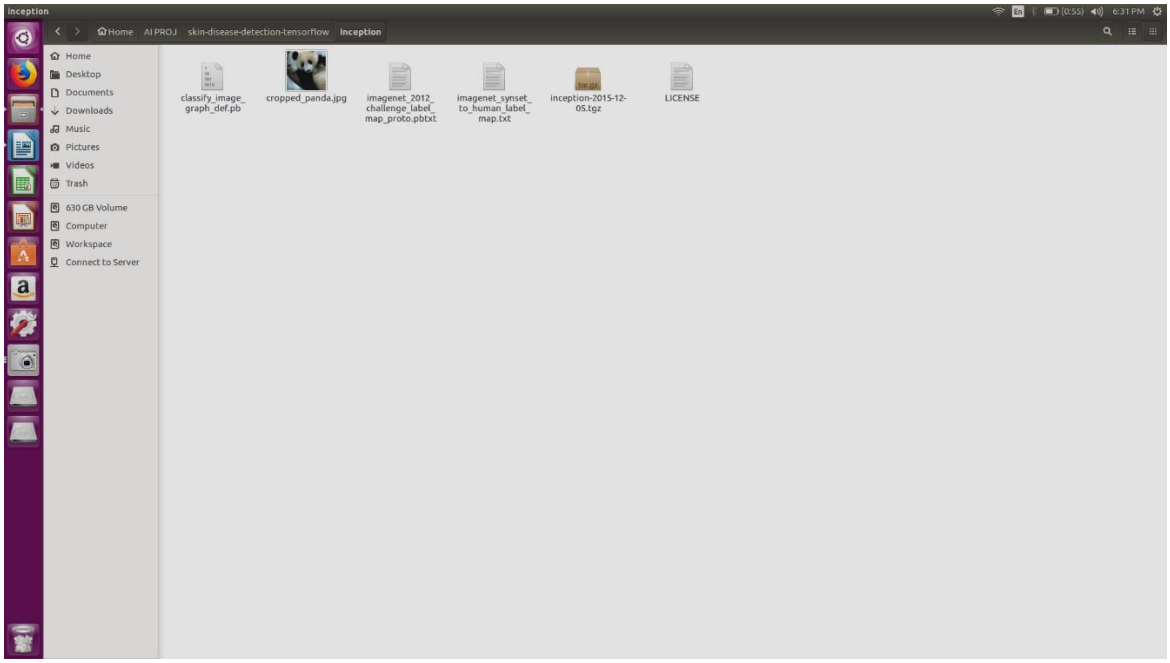

Figure 4. Inception model and creation of bottleneck for each image

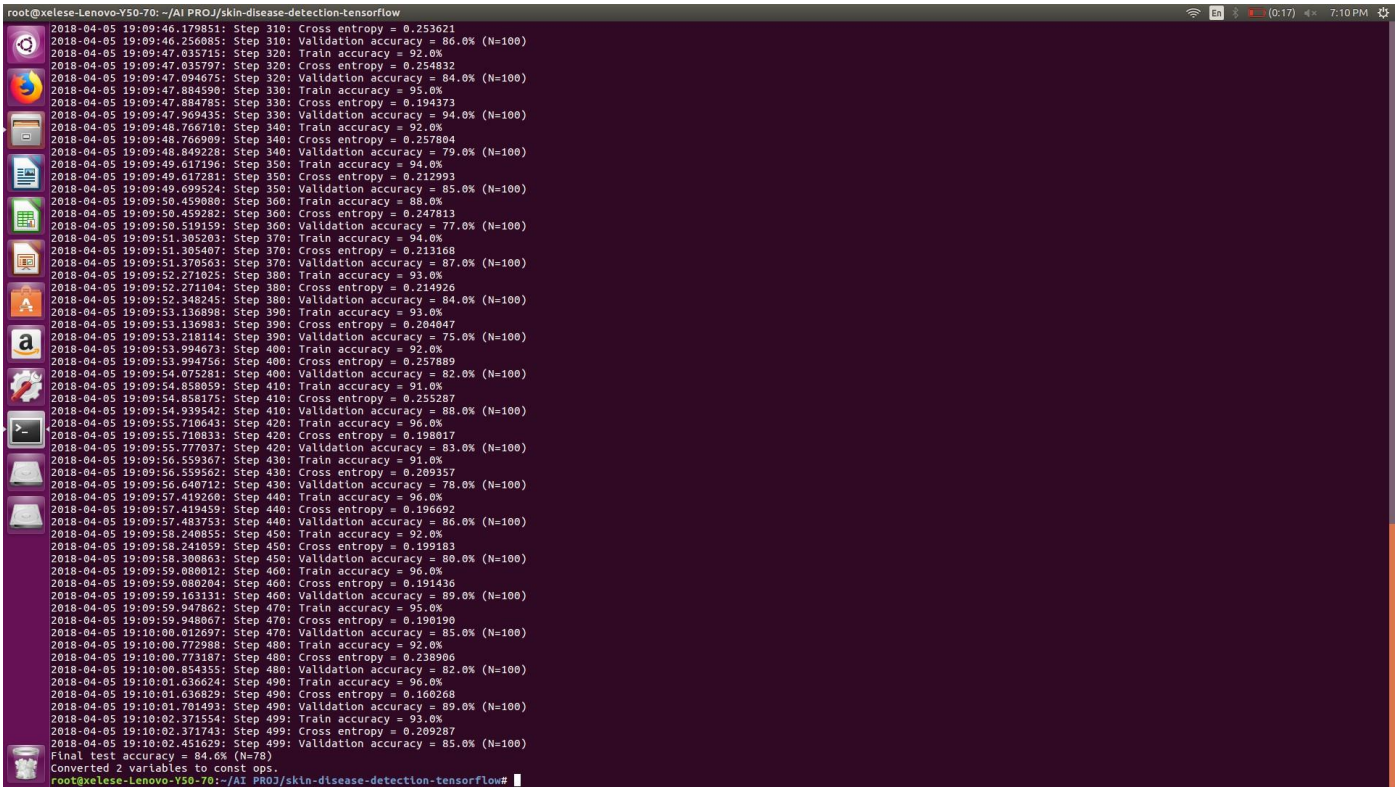

Figure 5. Training and testing of CNN with a result of 84.6% accuracy

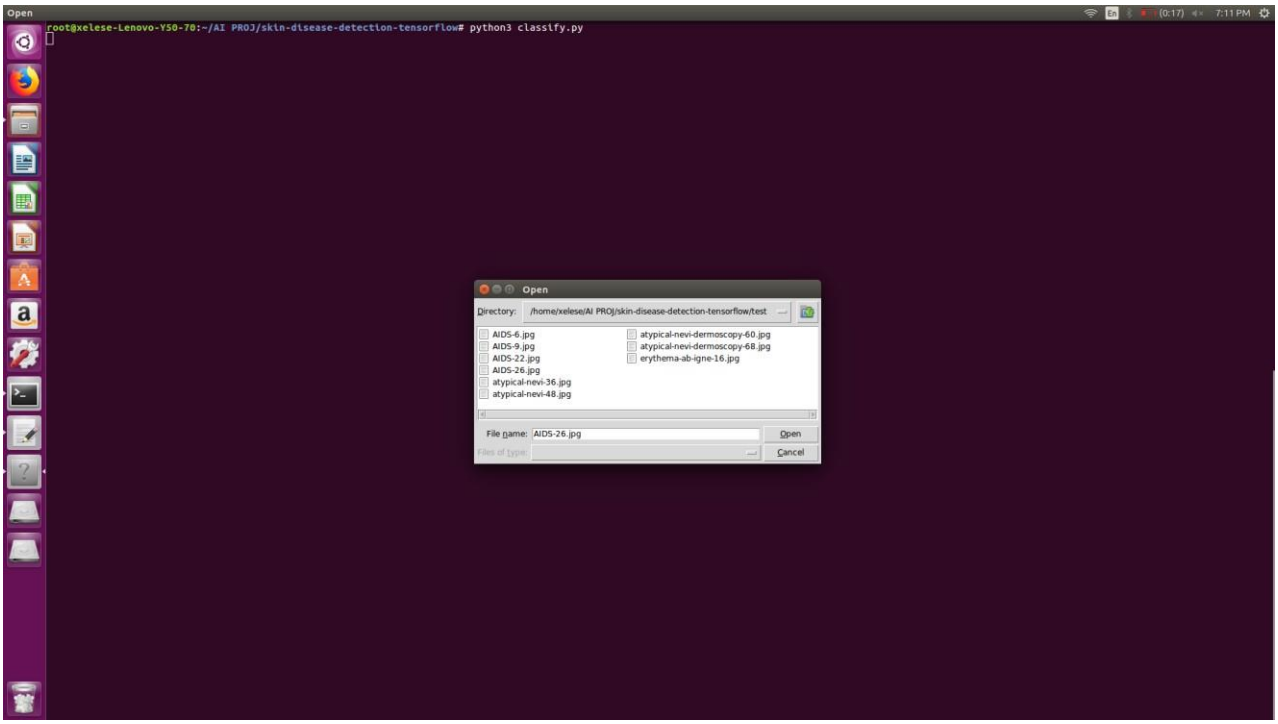

Figure 6. Testing with raw image(input image is herpes)

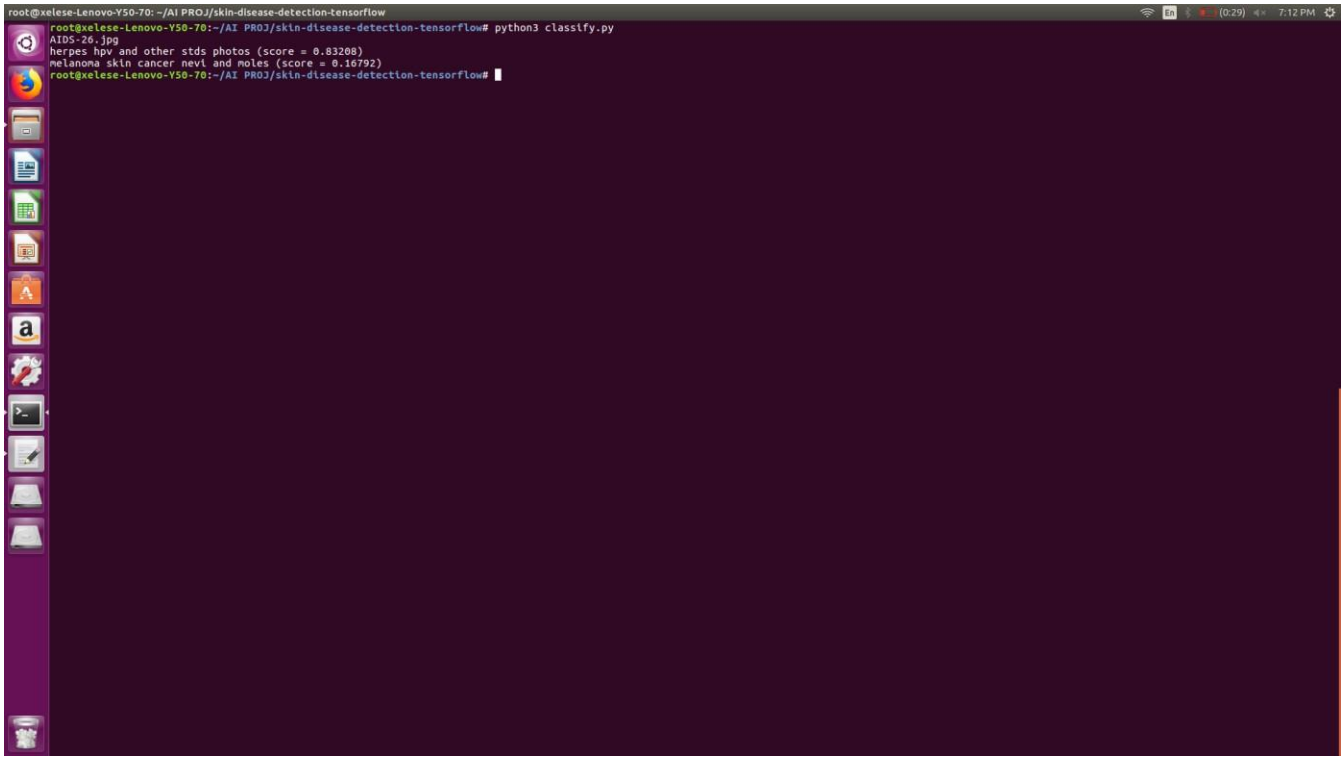

Figure 7. The raw image is classified appropriately with 83% confidence level thereby adhering to the accuracy of training value

Result and Discussion:

The images are split into training and testing set randomly. These pictures in these folders are put on the inception model which is our CNN architecture and trained for 500 epochs. After training, the model itself tests the accuracy of its prediction and gives out the training accuracy score.

As you can observe the above, after the initial training and testing phases we observed an accuracy of around 84% which is more than the average train-test phase in general. Then a raw image is uploaded for external testing has classified appropriately with a staggering confidence level of 98%. The other skin disease is seen to have a 2% rate though it is not the disease is because the total confidence level is expected to be 100% and it is impossible to get 100% confidence result for one particular disease.

In this study, an automated skin disease detection system is proposed which will help the medical society for the early detection of the skin diseases. This study is also highlighted to help the dermatologists in improving the diagnosis time and the accuracy of their intervention. Also with the usage and the demand of the system we can expand the number of diseases which can be recognized by the system into considerable amount.

In future research, we hope this work can be pushed much further and get a better accuracy. There are several places we can improve this work. First, since Inception model is not specialized for skin data, the pre-trained models may not be the best choice for skin disease classification. Thus, we should train a CNN model from scratch and test its performance. Second, the Dermnet images are organized using a biological taxonomy which is not the best choice for computer vision applications. We can work with a dermatologist to design a visually organized taxonomy and apply it to the classifier. Third, as the experimental results suggest that more variance in the training set would lead to a better accuracy, we should increase the size of our training set. Also note that the images retrieved by the networks are closely related to the ground truth. We may need to design a hierarchical classification algorithm using the retrieved images to improve the accuracy.

Conclusion:

As knowledge underlying health and illness is improving, the available data are becoming more numerous and complex, so many demands are arrived to process these data. We have investigated the feasibility of building a universal skin disease classification system using deep CNN. We have tackled this problem by appropriately pre-processing the image as required for the skin diseases classification. Our experiments show that the current state-of-art CNN models can achieve as high as 84.3% when testing on the Dermnet dataset.

The problem is the dataset used is done only to 2 classes of images, the model compares with those 2 alone. So a more sophisticated classification design has to be incorporated or invented to classify all 26 different types of skin diseases as described in Dermnet and classify them appropriately and give a single output with its confidence score of which class of skin disease it belongs too.

As a future work, this method can also be tested on a large dataset consisting of various complexities to improve the efficiency of the algorithm. Also with the usage and the demand of the system we can expand the number of diseases which can be recognized by the system into considerable amount.

References:

[1] G. K. Jana, A. Gupta, A. Das, R. Tripathy, and P. Sahoo. Herbal treatment to skin diseases: A global approach. *Drug Invention Today*, 2(8):381–384, August 2010.

[2] R. L. Siegel, K. D. Miller, and A. Jemal. Cancer statistics, 2015. *CA: a cancer journal for clinicians*, 65(1):5–29, 2015.

[3] H. Chang, Y. Zhou, A. Borowsky, K. Barner, P. Spellman, and B. Parvin. Stacked predictive sparse decomposition for classification of histology sections. *International Journal of Computer Vision*, 113(1):3–18, 2014.

[4] J. D. Whited and J. M. Grichnik. Does this patient have a mole or a melanoma? *Jama*, 279(9):696–701, 2016.

[5] Seema Kolkur, D.R. Kalbande, "Survey of Texture Based Feature Extraction for Skin Disease Detection", IEEE,2016.

[6] Seema Kolkur, D.R. Kalbande, "Survey of Texture Based Feature Extraction for Skin Disease Detection", IEEE,2016.

[7] Jyothilakshmi K. K, Jeeva J. B, "Detection of Malignant Skin Diseases Based on the Lesion Segmentation", International Conference on Communication and Signal Processing, April 3-5, 2014, India

[8] A. Masood and A. Ali Al-Jumaily. Computer aided diagnos- tic support system for skin cancer: A review of techniques and algorithms. *International Journal of Biomedical Imag- ing*, 2013, 2013.

[9] J. Arroyo and B. Zapirain. Automated detection of melanoma in dermoscopic images. In J. Scharcanski and M. E. Celebi, editors, *Computer Vision Techniques for the Diagnosis of Skin Cancer*, Springer Berlin Heidelberg, 2014.

[10] Teck Yan Tan, Li Zhang, Ming Jiang, "An Intelligent Decision Support System for Skin Cancer Detection from Dermoscopic Images", 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)

[11] Charu Gupta, "IMPLEMENTATION OF BACK PROPAGATION ALGORITHM (of neural networks) IN VHDL," Department Of Electronics and Communication Engineering THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGYPatiala, India, Master thesis 2016.

Website: for dataset repository http://www.dermnet.com/dermatology-pictures-skin-disease-pictures

Appendix:

code:

train.py (train, validate, test) from

__future__ import absolute_import from

__future__ import division from

__future__ import print_function

import argparse from datetime

import datetime import

hashlib import os.path import

random import re import

struct import sys import tarfile

import numpy as np from

six.moves import urllib import

tensorflow as tf

from tensorflow.python.framework import graph_util

from tensorflow.python.framework import tensor_shape

from tensorflow.python.platform import gfile from

tensorflow.python.util import compat

FLAGS = None

These are all parameters that are tied to the particular model architecture we're using for Inception v3. These include things like tensor names and their sizes. If you want to adapt this script to work with another model, you will need to update these to reflect the values in the network you're using.

pylint: disable=line-too-long

DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz' #

pylint: enable=line-too-long

BOTTLENECK_TENSOR_NAME = 'pool_3/_reshape:0'

BOTTLENECK_TENSOR_SIZE = 2048

MODEL_INPUT_WIDTH = 299

MODEL_INPUT_HEIGHT = 299

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

```
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.

  “””     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_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.Graph().as_default() as graph:

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 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 create_bottleneck_file(bottleneck_path, image_lists, label_name, index, image_dir, category, sess, jpeg_data_tensor, bottleneck_tensor):
  """Create a single bottleneck file."""  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()
  try:
    bottleneck_values = run_bottleneck_on_image(sess, image_data, jpeg_data_tensor, bottleneck_tensor)
except:
    raise RuntimeError('Error during processing file %s' % image_path)

  bottleneck_string = ','.join(str(x) for x in bottleneck_values)
with open(bottleneck_path, 'w') as bottleneck_file:
    bottleneck_file.write(bottleneck_string)


def get_or_create_bottleneck(sess, image_lists, label_name, index, image_dir, category, bottleneck_dir, jpeg_data_tensor, 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)  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):
    create_bottleneck_file(bottleneck_path, image_lists, label_name, index, image_dir, category, sess, jpeg_data_tensor, bottleneck_tensor)
with open(bottleneck_path, 'r') as bottleneck_file:
    bottleneck_string = bottleneck_file.read()
did_hit_error = False
  try:
```

```
    bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
except ValueError:     print('Invalid float found, recreating
bottleneck')     did_hit_error = True   if did_hit_error:
    create_bottleneck_file(bottleneck_path, image_lists, label_name, index, image_dir, category, sess, jpeg_data_tensor, bottleneck_tensor)
with open(bottleneck_path, 'r') as bottleneck_file:
      bottleneck_string = bottleneck_file.read()
    # Allow exceptions to propagate here, since they shouldn't happen after a fresh creation
bottleneck_values = [float(x) for x in bottleneck_string.split(',')]   return
bottleneck_values


def cache_bottlenecks(sess, image_lists, image_dir, bottleneck_dir, jpeg_data_tensor, 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, 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_random_cached_bottlenecks(sess, image_lists, how_many, category, bottleneck_dir, image_dir, jpeg_data_tensor, bottleneck_tensor):
  """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.
bottleneck_tensor: The bottleneck output layer of the CNN graph.
  Returns:
    List of bottleneck arrays, their corresponding ground truths, and the relevant filenames.
  """   class_count =
len(image_lists.keys())   bottlenecks =
[]   ground_truths = []   filenames = []
if how_many >= 0:
    # Retrieve a random sample of bottlenecks.
for unused_i in range(how_many):
      label_index = random.randrange(class_count)
label_name = list(image_lists.keys())[label_index]
      image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
image_name = get_image_path(image_lists, label_name, image_index, image_dir, category)
      bottleneck = get_or_create_bottleneck(sess, image_lists, label_name, image_index, image_dir, category, bottleneck_dir, jpeg_data_tensor,
bottleneck_tensor)       ground_truth = np.zeros(class_count, dtype=np.float32)       ground_truth[label_index] = 1.0
bottlenecks.append(bottleneck)       ground_truths.append(ground_truth)       filenames.append(image_name)
  else:

    # Retrieve all bottlenecks.     for label_index, label_name in
enumerate(image_lists.keys()):       for image_index, image_name in
enumerate(image_lists[label_name][category]):
        image_name = get_image_path(image_lists, label_name, image_index, image_dir, category)
        bottleneck = get_or_create_bottleneck(sess, image_lists, label_name, image_index, image_dir, category, bottleneck_dir, jpeg_data_tensor,
bottleneck_tensor)         ground_truth = np.zeros(class_count, dtype=np.float32)         ground_truth[label_index] = 1.0
```

```
bottlenecks.append(bottleneck)  ground_truths.append(ground_truth)  filenames.append(image_name)  return bottlenecks,
ground_truths, filenames


def get_random_distorted_bottlenecks(sess, image_lists, how_many, category, image_dir, input_jpeg_tensor, distorted_image, resized_input_tensor,
bottleneck_tensor):
  """Retrieves bottleneck values for training images, after distortions.
  If we're training with distortions like crops, scales, or flips, we have to recalculate the full model for every image, and so we can't use cached bottleneck
values. Instead we find random images for the requested category, run them through the distortion graph, and then the full graph to get the bottleneck results for
each.  Args:
    sess: Current TensorFlow Session.  image_lists: Dictionary of training images for
each label.  how_many: The integer number of bottleneck values to return.
category: Name string of which set of images to fetch - training, testing, or validation.
image_dir: Root folder string of the subfolders containing the training images.
    input_jpeg_tensor: The input layer we feed the image data to.
    distorted_image: The output node of the distortion graph.
resized_input_tensor: The input node of the recognition graph.
bottleneck_tensor: The bottleneck output layer of the CNN graph.


  Returns:
    List of bottleneck arrays and their corresponding ground truths.
  """  class_count =
len(image_lists.keys())  bottlenecks =
[]  ground_truths = []  for unused_i
in range(how_many):
    label_index = random.randrange(class_count)
    label_name = list(image_lists.keys())[label_index]
    image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
image_path = get_image_path(image_lists, label_name, image_index, image_dir, category)
if not gfile.Exists(image_path):
      tf.logging.fatal('File does not exist %s', image_path)
jpeg_data = gfile.FastGFile(image_path, 'rb').read()
    # Note that we materialize the distorted_image_data as a numpy array before sending running inference on the image. This involves 2 memory copies and
might be optimized in other implementations.
    distorted_image_data = sess.run(distorted_image, {input_jpeg_tensor: jpeg_data})  bottleneck =
run_bottleneck_on_image(sess, distorted_image_data, resized_input_tensor, bottleneck_tensor)
ground_truth = np.zeros(class_count, dtype=np.float32)  ground_truth[label_index] = 1.0
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:

  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 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.
  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'):
initial_value = tf.truncated_normal([BOTTLENECK_TENSOR_SIZE, class_count], stddev=0.001)


      layer_weights = tf.Variable(initial_value, name='final_weights')


      variable_summaries(layer_weights)
with tf.name_scope('biases'):
      layer_biases = tf.Variable(tf.zeros([class_count]), name='final_biases')
      variable_summaries(layer_biases)
with tf.name_scope('Wx_plus_b'):
```

```
    logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
tf.summary.histogram('pre_activations', logits)

 final_tensor = tf.nn.softmax(logits, name=final_tensor_name)
tf.summary.histogram('activations', final_tensor)

 with tf.name_scope('cross_entropy'):
   cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=ground_truth_input, logits=logits)
with tf.name_scope('total'):
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
tf.summary.scalar('cross_entropy', cross_entropy_mean)   with
tf.name_scope('train'):
   optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)    train_step =
optimizer.minimize(cross_entropy_mean)   return (train_step, cross_entropy_mean,
bottleneck_input, ground_truth_input, final_tensor)

def add_evaluation_step(result_tensor, ground_truth_tensor):
 """Inserts the operations we need to evaluate the accuracy of our results.
 Args:
   result_tensor: The new final node that produces results.
ground_truth_tensor: The node we feed ground truth data into.
 Returns:
   Tuple of (evaluation step, prediction).
 """   with tf.name_scope('accuracy'):    with
tf.name_scope('correct_prediction'):      prediction = tf.argmax(result_tensor,
1)      correct_prediction = tf.equal(prediction, tf.argmax(ground_truth_tensor,
1))    with tf.name_scope('accuracy'):
    evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', evaluation_step)   return evaluation_step,
prediction def 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)
class_count = len(image_lists.keys())   if class_count == 0:
    print('No valid folders of images found at ' + FLAGS.image_dir)
return -1   if class_count == 1:
    print('Only one valid folder of images found at ' + FLAGS.image_dir + ' - multiple classes are needed for classification.')
return -1

  # See if the command-line flags mean we're applying any distortions.   do_distort_images = should_distort_images(FLAGS.flip_left_right,
FLAGS.random_crop, FLAGS.random_scale, FLAGS.random_brightness)

  with tf.Session(graph=graph) as sess:
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(len(image_lists.keys()), FLAGS.final_tensor_name,
bottleneck_tensor)

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

    # Merge all the summaries and write them out to the summaries_dir      merged =
tf.summary.merge_all()      train_writer =
tf.summary.FileWriter(FLAGS.summaries_dir + '/train', sess.graph)
validation_writer = tf.summary.FileWriter(FLAGS.summaries_dir + '/validation')
```

```
  # Set up all our weights to their initial default values.
init = tf.global_variables_initializer()     sess.run(init)


  # Run the training for as many cycles as requested on the command line.
for i in range(FLAGS.how_many_training_steps):
    # Get a batch of input bottleneck values, either calculated fresh every time with distortions applied, or from the cache stored on disk.
if do_distort_images:
      (train_bottlenecks, train_ground_truth) = get_random_distorted_bottlenecks(sess, image_lists, FLAGS.train_batch_size, 'training', FLAGS.image_dir, distorted_jpeg_data_tensor, distorted_image_tensor, resized_image_tensor, bottleneck_tensor)
    else:
      (train_bottlenecks, train_ground_truth, _) = get_random_cached_bottlenecks(sess, image_lists, FLAGS.train_batch_size, 'training', FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor, bottleneck_tensor)
    # 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))




  # 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%% (N=%d)' % (datetime.now(), i, validation_accuracy * 100, len(validation_bottlenecks)))


  # We've completed all our training, so run a final test evaluation on some new images we haven't used before.
  test_bottlenecks, test_ground_truth, test_filenames = (get_random_cached_bottlenecks(sess, image_lists, FLAGS.test_batch_size, 'testing', FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor, bottleneck_tensor))     test_accuracy, predictions = sess.run([evaluation_step, prediction], feed_dict={bottleneck_input: test_bottlenecks, ground_truth_input: test_ground_truth})     print('Final test accuracy = %.1f%% (N=%d)' % (test_accuracy * 100, len(test_bottlenecks)))     if FLAGS.print_misclassified_test_images:
    print('=== MISCLASSIFIED TEST IMAGES ===')
```

```
      for i, test_filename in enumerate(test_filenames):
if predictions[i] != test_ground_truth[i].argmax():
          print('%70s  %s' % (test_filename, list(image_lists.keys())[predictions[i]]))


    # 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__':
  parser = argparse.ArgumentParser()
parser.add_argument(
      '--image_dir',       type=str,
default='',       help='Path to folders of
labeled images.'
  )
  parser.add_argument(       '--output_graph',
type=str,       default='/tmp/output_graph.pb',
help='Where to save the trained graph.'  )
parser.add_argument(

      '--output_labels',       type=str,
default='/tmp/output_labels.txt',       help='Where to save
the trained graph\'s labels.' )   parser.add_argument(
'--summaries_dir',       type=str,
default='/tmp/retrain_logs',       help='Where to save
summary logs for TensorBoard.'
  )
  parser.add_argument(       '--how_many_training_steps',
type=int,       default=4000,       help='How many
training steps to run before ending.'
  )
  parser.add_argument(
      '--learning_rate',       type=float,       default=0.01,
help='How large a learning rate to use when training.'
```

```
  )
  parser.add_argument(      '--testing_percentage',
type=int,      default=10,      help='What percentage of
images to use as a test set.'
  )
  parser.add_argument(
      '--validation_percentage',      type=int,      default=10,
help='What percentage of images to use as a validation set.'
  )

  parser.add_argument(      '--eval_step_interval',
type=int,      default=10,      help='How often to
evaluate the training results.'
  )
  parser.add_argument(      '--train_batch_size',
type=int,      default=100,      help='How many
images to train on at a time.'
  )
  parser.add_argument(
      '--test_batch_size',
type=int,      default=-1,
help="""\
      How many images to test on. This test set is only used once, to evaluate the final accuracy of the model after training completes.
      A value of -1 causes the entire test set to be used, which leads to more stable results across runs.\
      """
  )
  parser.add_argument(
      '--validation_batch_size',
type=int,      default=100,
help="""\
      How many images to use in an evaluation batch. This validation set is used much more often than the test set, and is an early indicator of how accurate the model is during training.
      A value of -1 causes the entire validation set to be used, which leads to more stable results across training iterations, but may be slower on large training sets.\
      """
  )
  parser.add_argument(
```

```
    '--print_misclassified_test_images',
default=False,      help="""\
    Whether to print out a list of all misclassified test images.\
    """,
action='store_true'
 )




 parser.add_argument(
    '--model_dir',
type=str,
default='/tmp/imagenet',
help="""\
    Path to classify_image_graph_def.pb, imagenet_synset_to_human_label_map.txt, and imagenet_2012_challenge_label_map_proto.pbtxt.\
    """
 )
 parser.add_argument(
    '--bottleneck_dir',      type=str,
default='/tmp/bottleneck',      help='Path to cache
bottleneck layer values as files.'
 )
 parser.add_argument(      '--
final_tensor_name',      type=str,
default='final_result',
help="""\
    The name of the output classification layer in the retrained graph.\
    """
 )
 parser.add_argument(
    '--flip_left_right',
default=False,      help="""\
    Whether to randomly flip half of the training images horizontally.\
    """,
action='store_true'
 )
```

```
  parser.add_argument(
'--random_crop',
type=int,       default=0,
help="""\
      A percentage determining how much of a margin to randomly crop off the training images.\
      """
  )



  parser.add_argument(
'--random_scale',
type=int,       default=0,
help="""\
      A percentage determining how much to randomly scale up the size of the training images by.\
      """
  )
  parser.add_argument(
'--random_brightness',
type=int,       default=0,
help="""\
      A percentage determining how much to randomly multiply the training image input pixels up or down by.\
      """
  )
  FLAGS, unparsed = parser.parse_known_args()
tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```