**Chapter 3**

**Developing The Deep Net Model**

**3.1 Data Collection and Gathering**

The first place to start is by looking at the images gathered, since the most common issues with training come from the data that's being fed in.

For training to work well, at least a hundred photos of each kind of object which is to be recognized needs to be gathered. The more photos gathered, the better the accuracy of the trained model is likely to be. It is needed to make sure that the photos are a good representation of what the application will actually encounter. For example, if all photos are indoors against a blank wall and the users are trying to recognize objects outdoors, we won't see good results when deployed.

Another pitfall to avoid is that the learning process will pick up on anything that the labeled images have in common with each other, and if we are not careful that might be something that's not useful. For example if we photograph one kind of object in a blue room, and another in a green one, then the model will end up basing its prediction on the background color, not the features of the object. To avoid this, pictures are to be taken in as wide a variety of situations as available, at different times, and with different devices.

The categories need to be carefully selected. It might be worth splitting big categories that cover a lot of different physical forms into smaller ones that are more visually distinct. For example instead of 'vehicle' we might use 'car', 'motorbike', and 'truck'. It's also worth thinking about whether you have a 'closed world' or an 'open world' problem. In a closed world, the only things we'll ever be asked to categorize are the classes of object you know about. This might apply to a plant recognition app where the user is likely to be taking a picture of a flower, so all we have to do is decide which species. By contrast a roaming robot might see all sorts of different things through its camera as it wanders around the world. In that case we’d want the classifier to report if it wasn't sure what it was seeing. This can be hard to do well, but often if when a large number of typical 'background' photos with no relevant objects in them are collected, we can add them to an extra 'unknown' class in the image folders.

It's also worth checking to make sure that all of our images are labeled correctly. Often user-generated tags are unreliable for our purposes. For example: pictures tagged #daisy might also include people and characters named Daisy. Correct Labels can do wonders to the overall accuracy of our model.

At least 20 images of each Categories were downloaded which were distinct and contains only the specified Object image for Proper training. The Similar images were grouped into folder, The folder names are used as Label Names. The images were downloaded from

* various non-copyrighted and open source images such as Google’s Image Net.
* from Kaggle.com the dataset named “Natural Images” by Prasun Roy.

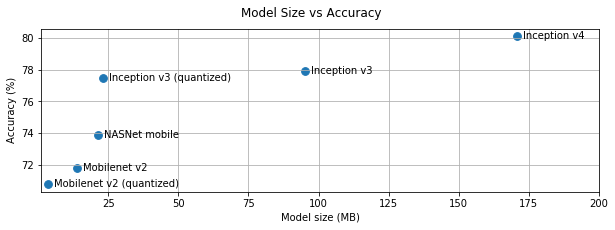
A common way of improving the results of image training is by deforming, cropping, or brightening the training inputs in random ways. This has the advantage of expanding the effective size of the training data thanks to all the possible variations of the same images, and tends to help the network learn to cope with all the distortions that will occur in real-life uses of the classifier. The biggest disadvantage of enabling these distortions in our script is that the bottleneck caching is no longer useful, since input images are never reused exactly. This means the training process takes a lot longer (many hours), so it's recommended to try this as a way of polishing the model only after the model works reasonably well.

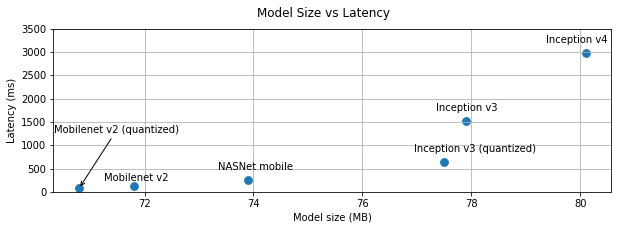
**3.2 Model Selection for Transfer Learning:-**

Quantized image classification models offer the smallest model size and fastest performance, at the expense of accuracy.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model Name** | **Paper and Model** | **Model Size** | **Top-1 accuracy** | **Top-5 accuracy** | **TF Lite Performance** |
| Mobilenet\_V1\_0.25\_128\_quant | paper, tflite&pb | 0.5 Mb | 39.5% | 64.4% | 3.7 ms |
| Mobilenet\_V1\_0.25\_160\_quant | paper, tflite&pb | 0.5 Mb | 42.8% | 68.1% | 5.5 ms |
| Mobilenet\_V1\_0.25\_192\_quant | paper, tflite&pb | 0.5 Mb | 45.7% | 70.8% | 7.9 ms |
| Mobilenet\_V1\_0.25\_224\_quant | paper, tflite&pb | 0.5 Mb | 48.2% | 72.8% | 10.4 ms |
| Mobilenet\_V1\_0.50\_128\_quant | paper, tflite&pb | 1.4 Mb | 54.9% | 78.1% | 8.8 ms |
| Mobilenet\_V1\_0.50\_160\_quant | paper, tflite&pb | 1.4 Mb | 57.2% | 80.5% | 13.0 ms |
| Mobilenet\_V1\_0.50\_192\_quant | paper, tflite&pb | 1.4 Mb | 59.9% | 82.1% | 18.3 ms |
| Mobilenet\_V1\_0.50\_224\_quant | paper, tflite&pb | 1.4 Mb | 61.2% | 83.2% | 24.7 ms |
| Mobilenet\_V1\_0.75\_128\_quant | paper, tflite&pb | 2.6 Mb | 55.9% | 79.1% | 16.2 ms |
| Mobilenet\_V1\_0.75\_160\_quant | paper, tflite&pb | 2.6 Mb | 62.4% | 83.7% | 24.3 ms |
| Mobilenet\_V1\_0.75\_192\_quant | paper, tflite&pb | 2.6 Mb | 66.1% | 86.2% | 33.8 ms |
| Mobilenet\_V1\_0.75\_224\_quant | paper, tflite&pb | 2.6 Mb | 66.9% | 86.9% | 45.4 ms |
| Mobilenet\_V1\_1.0\_128\_quant | paper, tflite&pb | 4.3 Mb | 63.3% | 84.1% | 24.9 ms |
| Mobilenet\_V1\_1.0\_160\_quant | paper, tflite&pb | 4.3 Mb | 66.9% | 86.7% | 37.4 ms |
| Mobilenet\_V1\_1.0\_192\_quant | paper, tflite&pb | 4.3 Mb | 69.1% | 88.1% | 51.9 ms |
| Mobilenet\_V1\_1.0\_224\_quant | paper, tflite&pb | 4.3 Mb | 70.0% | 89.0% | 70.2 ms |
| Mobilenet\_V2\_1.0\_224\_quant | paper, tflite&pb | 3.4 Mb | 70.8% | 89.9% | 80.3 ms |
| Inception\_V1\_quant | paper, tflite&pb | 6.4 Mb | 70.1% | 89.8% | 154.5 ms |
| Inception\_V2\_quant | paper, tflite&pb | 11 Mb | 73.5% | 91.4% | 235.0 ms |
| Inception\_V3\_quant | paper, tflite&pb | 23 Mb | 77.5% | 93.7% | 637 ms |
| Inception\_V4\_quant | paper, tflite&pb | 41 Mb | 79.5% | 93.9% | 1250.8 ms |

Depending on the task, we will need to make a tradeoff between model complexity and size. Since the task requires Moderate accuracy, we won’t need a large and complex model. For tasks that require less precision, it is better to use a smaller model because they not only use less disk space and memory, but they are also generally faster and more energy efficient. For example, graphs below show accuracy and latency tradeoffs for some common image classification models.





|  |  |  |
| --- | --- | --- |
| **Model Name** | **Device** | **Mean Inference Time** |
| Mobilenet\_1.0\_224(float) | Pixel 2 | 123.3 ms |
|  | Pixel XL | 113.3 ms |
| Mobilenet\_1.0\_224 (quant) | Pixel 2 | 65.4 ms |
|  | Pixel XL | 74.6 ms |
| NASNet mobile | Pixel 2 | 273.8 ms |
|  | Pixel XL | 210.8 ms |
| SqueezeNet | Pixel 2 | 234.0 ms |
|  | Pixel XL | 158.0 ms |
| Inception\_ResNet\_V2 | Pixel 2 | 2846.0 ms |
|  | Pixel XL | 1973.0 ms |
| Inception\_V4 | Pixel 2 | 3180.0 ms |
|  | Pixel XL | 2262.0 ms |

We are going to use a model trained on the Image Net Large Visual Recognition Challenge dataset. These models can differentiate between 1,000 different classes, like Dalmatian or dishwasher. We will have a choice of model architectures, so that we can determine the right trade-off between speed, size and accuracy for our problem. We will use this same model, but retrain it to tell apart a small number of classes based on our own examples.

There are 32 different Mobilenet models to choose from, with a variety of file

size and latency options. The first number can be '1.0', '0.75', '0.50', or

'0.25' to control the size, and the second controls the input image size, either

'224', '192', '160', or '128', with smaller sizes running faster

After careful consideration and using the above facts and analysis we select mobilenet\_0.50\_224 model for the purpose of transfer learning.

IMAGE\_SIZE=224

ARCHITECTURE="mobilenet\_0.50\_${IMAGE\_SIZE}"

**3.3 Training:-**

**3.3.1 Retrain.py**

MAX\_NUM\_IMAGES\_PER\_CLASS = 2 \*\* 27 – 1

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.“””

if not gfile.Exists(image\_dir):

tf.logging.error("Image directory '" + image\_dir + "' not found.")

return None

result = collections.OrderedDict()

sub\_dirs = [os.path.join(image\_dir,item)

for item in gfile.ListDirectory(image\_dir)]

sub\_dirs = sorted(item for item in sub\_dirs if gfile.IsDirectory(item))

for sub\_dir in sub\_dirs:

extensions = ['jpg', 'jpeg', 'JPG', 'JPEG']

file\_list = []

dir\_name = os.path.basename(sub\_dir)

if dir\_name == image\_dir:

continue

tf.logging.info("Looking for images in '" + dir\_name + "'")

for extension in extensions:

file\_glob = os.path.join(image\_dir, dir\_name, '\*.' + extension)

file\_list.extend(gfile.Glob(file\_glob))

if not file\_list:

tf.logging.warning('No files found')

continue

if len(file\_list) < 20:

tf.logging.warning('WARNING: Folder has less than 20 images, which may cause issues.')

elif len(file\_list) > MAX\_NUM\_IMAGES\_PER\_CLASS:

tf.logging.warning('WARNING: Folder {} has more than {} images. Some images will never be selected.'.format(dir\_name, MAX\_NUM\_IMAGES\_PER\_CLASS))

label\_name = re.sub(r'[^a-z0-9]+', ' ', dir\_name.lower())

training\_images = []

testing\_images = []

validation\_images = []

for file\_name in file\_list:

base\_name = os.path.basename(file\_name)

# We want to ignore anything after '\_nohash\_' in the file name when

# deciding which set to put an image in, the data set creator has a way of

# grouping photos that are close variations of each other. For example

# this is used in the plant disease data set to group multiple pictures of

# the same leaf.

hash\_name = re.sub(r'\_nohash\_.\*$', '', file\_name)

# This looks a bit magical, but we need to decide whether this file should

# go into the training, testing, or validation sets, and we want to keep

# existing files in the same set even if more files are subsequently

# added.

# To do that, we need a stable way of deciding based on just the file name

# itself, so we do a hash of that and then use that to generate a

# probability value that we use to assign it.

hash\_name\_hashed = hashlib.sha1(compat.as\_bytes(hash\_name)).hexdigest()

percentage\_hash = ((int(hash\_name\_hashed, 16) %

(MAX\_NUM\_IMAGES\_PER\_CLASS + 1)) \*

(100.0 / MAX\_NUM\_IMAGES\_PER\_CLASS))

if percentage\_hash < validation\_percentage:

validation\_images.append(base\_name)

elif percentage\_hash < (testing\_percentage + validation\_percentage):

testing\_images.append(base\_name)

else:

training\_images.append(base\_name)

result[label\_name] ={‘dir': dir\_name, 'training': training\_images,'testing':testing\_images,'validation': validation\_images,}

return result

def get\_image\_path(image\_lists, label\_name, index, image\_dir, category):

""""Returns a path to an image for a label at the given index.“””

if label\_name not in image\_lists:

tf.logging.fatal('Label does not exist %s.', label\_name)

label\_lists = image\_lists[label\_name]

if category not in label\_lists:

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

category\_list = label\_lists[category]

if not category\_list:

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

mod\_index = index % len(category\_list)

base\_name = category\_list[mod\_index]

sub\_dir = label\_lists['dir']

full\_path = os.path.join(image\_dir, sub\_dir, base\_name)

return full\_path

def get\_bottleneck\_path(image\_lists, label\_name, index, bottleneck\_dir,category, architecture):

""""Returns a path to a bottleneck file for a label at the given index.“””

return get\_image\_path(image\_lists, label\_name, index, bottleneck\_dir,

category) + '\_' + architecture + '.txt'

def create\_model\_graph(model\_info):

""""Creates a graph from saved GraphDef file and returns a Graph object.”””

with tf.Graph().as\_default() as graph:

model\_path = os.path.join(FLAGS.model\_dir, model\_info['model\_file\_name'])

with gfile.FastGFile(model\_path, 'rb') as f:

graph\_def = tf.GraphDef()

graph\_def.ParseFromString(f.read())

bottleneck\_tensor, resized\_input\_tensor = (tf.import\_graph\_def(graph\_def,

name='',return\_elements=[model\_info['bottleneck\_tensor\_name'],

model\_info['resized\_input\_tensor\_name'],]))

return graph, bottleneck\_tensor, resized\_input\_tensor

def run\_bottleneck\_on\_image(sess, image\_data, image\_data\_tensor,

decoded\_image\_tensor, resized\_input\_tensor,

bottleneck\_tensor):

"""Runs inference on an image to extract the 'bottleneck' summary layer.”””

# First decode the JPEG image, resize it, and rescale the pixel values.

resized\_input\_values = sess.run(decoded\_image\_tensor,{image\_data\_tensor: image\_data})

# Then run it through the recognition network.

bottleneck\_values = sess.run(bottleneck\_tensor,

{resized\_input\_tensor: resized\_input\_values})

bottleneck\_values = np.squeeze(bottleneck\_values)

return bottleneck\_values

def maybe\_download\_and\_extract(data\_url):

"""Download and extract model tar file.

If the pretrained model we're using doesn't already exist, this function

downloads it from the TensorFlow.org website and unpacks it into a directory.“”””

dest\_directory = FLAGS.model\_dir

if not os.path.exists(dest\_directory):

os.makedirs(dest\_directory)

filename = data\_url.split('/')[-1]

filepath = os.path.join(dest\_directory, filename)

if not os.path.exists(filepath):

def \_progress(count, block\_size, total\_size):

sys.stdout.write('\r>> Downloading %s %.1f%%' %(filename,

float(count \* block\_size) / float(total\_size) \* 100.0))

sys.stdout.flush()

filepath, \_ = urllib.request.urlretrieve(data\_url, filepath, \_progress)

print()

statinfo = os.stat(filepath)

tf.logging.info('Successfully downloaded', filename, statinfo.st\_size,'bytes.')

tarfile.open(filepath, 'r:gz').extractall(dest\_directory)

def create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,

image\_dir, category, sess, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor,bottleneck\_tensor):

"""Create a single bottleneck file."""

tf.logging.info('Creating bottleneck at ' + bottleneck\_path)

image\_path = get\_image\_path(image\_lists, label\_name, index,image\_dir, category)

if not gfile.Exists(image\_path):

tf.logging.fatal('File does not exist %s', image\_path)

image\_data = gfile.FastGFile(image\_path, 'rb').read()

try:

bottleneck\_values = run\_bottleneck\_on\_image(sess, image\_data, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor, bottleneck\_tensor)

except Exception as e:

raise RuntimeError('Error during processing file %s (%s)' % (image\_path,str(e)))

bottleneck\_string = ','.join(str(x) for x in bottleneck\_values)

with open(bottleneck\_path, 'w') as bottleneck\_file:

bottleneck\_file.write(bottleneck\_string)

def get\_or\_create\_bottleneck(sess, image\_lists, label\_name, index, image\_dir,category, bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor,bottleneck\_tensor, architecture):

"""Retrieves or calculates bottleneck values for an image.

If a cached version of the bottleneck data exists on-disk, return that,otherwise calculate the data and save it to disk for future use.

label\_lists = image\_lists[label\_name]

sub\_dir = label\_lists['dir']

sub\_dir\_path = os.path.join(bottleneck\_dir, sub\_dir)

ensure\_dir\_exists(sub\_dir\_path)

bottleneck\_path = get\_bottleneck\_path(image\_lists, label\_name, index,

bottleneck\_dir, category, architecture)

if not os.path.exists(bottleneck\_path):

create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,

image\_dir, category, sess, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor,bottleneck\_tensor)

with open(bottleneck\_path, 'r') as bottleneck\_file:

bottleneck\_string = bottleneck\_file.read()

did\_hit\_error = False

try:

bottleneck\_values = [float(x) for x in bottleneck\_string.split(',')]

except ValueError:

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

did\_hit\_error = True

if did\_hit\_error:

create\_bottleneck\_file(bottleneck\_path, image\_lists, label\_name, index,

image\_dir, category, sess, jpeg\_data\_tensor,

decoded\_image\_tensor, resized\_input\_tensor,

bottleneck\_tensor)

with open(bottleneck\_path, 'r') as bottleneck\_file:

bottleneck\_string = bottleneck\_file.read()

# Allow exceptions to propagate here, since they shouldn't happen after a

# fresh creation

bottleneck\_values = [float(x) for x in bottleneck\_string.split(',')]

return bottleneck\_values

def cache\_bottlenecks(sess, image\_lists, image\_dir, bottleneck\_dir,

jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor, bottleneck\_tensor, architecture):

"""Ensures all the training, testing, and validation bottlenecks are cached.

Because we're likely to read the same image multiple times (if there are no distortions applied during training) it can speed things up a lot if we calculate the bottleneck layer values once for each image during preprocessing, and then just read those cached values repeatedly during training. Here we go through all the images we've found, calculate those values, and save them off.

how\_many\_bottlenecks = 0

ensure\_dir\_exists(bottleneck\_dir)

for label\_name, label\_lists in image\_lists.items():

for category in ['training', 'testing', 'validation']:

category\_list = label\_lists[category]

for index, unused\_base\_name in enumerate(category\_list):

get\_or\_create\_bottleneck(sess, image\_lists, label\_name, index, image\_dir, category,

bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor, bottleneck\_tensor, architecture)

how\_many\_bottlenecks += 1

if how\_many\_bottlenecks % 100 == 0:

tf.logging.info(str(how\_many\_bottlenecks) + ' bottleneck files created.')

def get\_random\_cached\_bottlenecks(sess, image\_lists, how\_many, category, bottleneck\_dir, image\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor,

bottleneck\_tensor, architecture):

"""Retrieves bottleneck values for cached images.

If no distortions are being applied, this function can retrieve the cached bottleneck values directly from disk for images. It picks a random set of images from the specified category.””

class\_count = len(image\_lists.keys())

bottlenecks = []

ground\_truths = []

filenames = []

if how\_many >= 0:

# Retrieve a random sample of bottlenecks.

for unused\_i in range(how\_many):

label\_index = random.randrange(class\_count)

label\_name = list(image\_lists.keys())[label\_index]

image\_index = random.randrange(MAX\_NUM\_IMAGES\_PER\_CLASS + 1)

image\_name = get\_image\_path(image\_lists, label\_name, image\_index,

image\_dir, category)

bottleneck = get\_or\_create\_bottleneck( sess, image\_lists, label\_name, image\_index, image\_dir, category, bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor, bottleneck\_tensor, architecture)

ground\_truth = np.zeros(class\_count, dtype=np.float32)

ground\_truth[label\_index] = 1.0

bottlenecks.append(bottleneck)

ground\_truths.append(ground\_truth)

filenames.append(image\_name)

else:

# Retrieve all bottlenecks.

for label\_index, label\_name in enumerate(image\_lists.keys()):

for image\_index, image\_name in enumerate(

image\_lists[label\_name][category]):

image\_name = get\_image\_path(image\_lists, label\_name, image\_index,image\_dir, category)

bottleneck = get\_or\_create\_bottleneck(sess, image\_lists, label\_name, image\_index, image\_dir, category, bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_input\_tensor, bottleneck\_tensor, architecture)

ground\_truth = np.zeros(class\_count, dtype=np.float32)

ground\_truth[label\_index] = 1.0

bottlenecks.append(bottleneck)

ground\_truths.append(ground\_truth)

filenames.append(image\_name)

return bottlenecks, ground\_truths, filenames

def get\_random\_distorted\_bottlenecks(

sess, image\_lists, how\_many, category, image\_dir, input\_jpeg\_tensor,

distorted\_image, resized\_input\_tensor, bottleneck\_tensor):

"""Retrieves bottleneck values for training images, after distortions.

If we're training with distortions like crops, scales, or flips, we have to recalculate the full model for every image, and so we can't use cached bottleneck values. Instead we find random images for the requested category, run them through the distortion graph, and then the full graph to get the bottleneck results for each.

class\_count = len(image\_lists.keys())

bottlenecks = []

ground\_truths = []

for unused\_i in range(how\_many):

label\_index = random.randrange(class\_count)

label\_name = list(image\_lists.keys())[label\_index]

image\_index = random.randrange(MAX\_NUM\_IMAGES\_PER\_CLASS + 1)

image\_path = get\_image\_path(image\_lists, label\_name, image\_index, image\_dir,

category)

if not gfile.Exists(image\_path):

tf.logging.fatal('File does not exist %s', image\_path)

jpeg\_data = gfile.FastGFile(image\_path, 'rb').read()

# Note that we materialize the distorted\_image\_data as a numpy array before

# sending running inference on the image. This involves 2 memory copies and

# might be optimized in other implementations.

distorted\_image\_data = sess.run(distorted\_image,{input\_jpeg\_tensor: jpeg\_data})

bottleneck\_values = sess.run(bottleneck\_tensor,

{resized\_input\_tensor: distorted\_image\_data})

bottleneck\_values = np.squeeze(bottleneck\_values)

ground\_truth = np.zeros(class\_count, dtype=np.float32)

ground\_truth[label\_index] = 1.0

bottlenecks.append(bottleneck\_values)

ground\_truths.append(ground\_truth)

return bottlenecks, ground\_truths

def add\_final\_training\_ops(class\_count, final\_tensor\_name, bottleneck\_tensor,

bottleneck\_tensor\_size):

"""Adds a new softmax and fully-connected layer for training.

We need to retrain the top layer to identify our new classes, so this function adds the right operations to the graph, along with some variables to hold the weights, and then sets up all the gradients for the backward pass. The set up for the softmax and fully-connected layers is based on:<https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html>

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.

with tf.name\_scope('accuracy'):

with tf.name\_scope('correct\_prediction'):

prediction = tf.argmax(result\_tensor, 1)

correct\_prediction = tf.equal(

prediction, tf.argmax(ground\_truth\_tensor, 1))

with tf.name\_scope('accuracy'):

evaluation\_step = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

tf.summary.scalar('accuracy', evaluation\_step)

return evaluation\_step, prediction

def save\_graph\_to\_file(sess, graph, graph\_file\_name):

output\_graph\_def = graph\_util.convert\_variables\_to\_constants( sess, graph.as\_graph\_def(), [FLAGS.final\_tensor\_name])

with gfile.FastGFile(graph\_file\_name, 'wb') as f:

f.write(output\_graph\_def.SerializeToString())

return

def prepare\_file\_system():

# Setup the directory we'll write summaries to for TensorBoard

if tf.gfile.Exists(FLAGS.summaries\_dir):

tf.gfile.DeleteRecursively(FLAGS.summaries\_dir)

tf.gfile.MakeDirs(FLAGS.summaries\_dir)

if FLAGS.intermediate\_store\_frequency > 0:

ensure\_dir\_exists(FLAGS.intermediate\_output\_graphs\_dir)

return

def create\_model\_info(architecture):

"""Given the name of a model architecture, returns information about it. There are different base image recognition pretrained models that can be retrained using transfer learning, and this function translates from the name of a model to the attributes that are needed to download and train with it.

architecture = architecture.lower()

if architecture == 'inception\_v3':

# pylint: disable=line-too-long

data\_url = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz'

# pylint: enable=line-too-long

bottleneck\_tensor\_name = 'pool\_3/\_reshape:0'

bottleneck\_tensor\_size = 2048

input\_width = 299

input\_height = 299

input\_depth = 3

resized\_input\_tensor\_name = 'Mul:0'

model\_file\_name = 'classify\_image\_graph\_def.pb'

input\_mean = 128

input\_std = 128

elif architecture.startswith('mobilenet\_'):

parts = architecture.split('\_')

if len(parts) != 3 and len(parts) != 4:

tf.logging.error("Couldn't understand architecture name '%s'",architecture)

return None

version\_string = parts[1]

if (version\_string != '1.0' and version\_string != '0.75' and

version\_string != '0.50' and version\_string != '0.25'):

tf.logging.error(

""""The Mobilenet version should be '1.0', '0.75', '0.50', or '0.25',

but found '%s' for architecture '%s'""",

version\_string, architecture)

return None

size\_string = parts[2]

if (size\_string != '224' and size\_string != '192' and

size\_string != '160' and size\_string != '128'):

tf.logging.error(""The Mobilenet input size should be '224', '192', '160', or '128' but found '%s' for architecture '%s'""",size\_string, architecture)

return None

if len(parts) == 3:

is\_quantized = False

else:

if parts[3] != 'quantized':

tf.logging.error("Couldn't understand architecture suffix '%s' for '%s'", parts[3],architecture)

return None

is\_quantized = True

data\_url = 'http://download.tensorflow.org/models/mobilenet\_v1\_'

data\_url += version\_string + '\_' + size\_string + '\_frozen.tgz'

bottleneck\_tensor\_name = 'MobilenetV1/Predictions/Reshape:0'

bottleneck\_tensor\_size = 1001

input\_width = int(size\_string)

input\_height = int(size\_string)

input\_depth = 3

resized\_input\_tensor\_name = 'input:0'

if is\_quantized:

model\_base\_name = 'quantized\_graph.pb'

else:

model\_base\_name = 'frozen\_graph.pb'

model\_dir\_name = 'mobilenet\_v1\_' + version\_string + '\_' + size\_string

model\_file\_name = os.path.join(model\_dir\_name, model\_base\_name)

input\_mean = 127.5

input\_std = 127.5

else:

tf.logging.error("Couldn't understand architecture name '%s'", architecture)

raise ValueError('Unknown architecture', architecture)

return {

'data\_url': data\_url,

'bottleneck\_tensor\_name': bottleneck\_tensor\_name,

'bottleneck\_tensor\_size': bottleneck\_tensor\_size,

'input\_width': input\_width,

'input\_height': input\_height,

'input\_depth': input\_depth,

'resized\_input\_tensor\_name': resized\_input\_tensor\_name,

'model\_file\_name': model\_file\_name,

'input\_mean': input\_mean,

'input\_std': input\_std,

}

def add\_jpeg\_decoding(input\_width, input\_height, input\_depth, input\_mean,

input\_std):

"""Adds operations that perform JPEG decoding and resizing to the graph.. jpeg\_data = tf.placeholder(tf.string, name='DecodeJPGInput')

decoded\_image = tf.image.decode\_jpeg(jpeg\_data, channels=input\_depth)

decoded\_image\_as\_float = tf.cast(decoded\_image, dtype=tf.float32)

decoded\_image\_4d = tf.expand\_dims(decoded\_image\_as\_float, 0)

resize\_shape = tf.stack([input\_height, input\_width])

resize\_shape\_as\_int = tf.cast(resize\_shape, dtype=tf.int32)

resized\_image = tf.image.resize\_bilinear(decoded\_image\_4d,resize\_shape\_as\_int)

offset\_image = tf.subtract(resized\_image, input\_mean)

mul\_image = tf.multiply(offset\_image, 1.0 / input\_std)

return jpeg\_data, mul\_image

def main(\_):

# Needed to make sure the logging output is visible.

# See https://github.com/tensorflow/tensorflow/issues/3047

tf.logging.set\_verbosity(tf.logging.INFO)

# Prepare necessary directories that can be used during training

prepare\_file\_system()

# Gather information about the model architecture we'll be using.

model\_info = create\_model\_info(FLAGS.architecture)

if not model\_info:

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

return -1

# Set up the pre-trained graph.

maybe\_download\_and\_extract(model\_info['data\_url'])

graph, bottleneck\_tensor, resized\_image\_tensor = (

create\_model\_graph(model\_info))

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

image\_lists = create\_image\_lists(FLAGS.image\_dir, FLAGS.testing\_percentage,

FLAGS.validation\_percentage)

class\_count = len(image\_lists.keys())

if class\_count == 0:

tf.logging.error('No valid folders of images found at ' + FLAGS.image\_dir)

return -1

if class\_count == 1:

tf.logging.error('Only one valid folder of images found at ' +

FLAGS.image\_dir +

' - multiple classes are needed for classification.')

return -1

# See if the command-line flags mean we're applying any distortions.

do\_distort\_images = should\_distort\_images(

FLAGS.flip\_left\_right, FLAGS.random\_crop, FLAGS.random\_scale,

FLAGS.random\_brightness)

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

# Set up the image decoding sub-graph.

jpeg\_data\_tensor, decoded\_image\_tensor = add\_jpeg\_decoding(

model\_info['input\_width'], model\_info['input\_height'],

model\_info['input\_depth'], model\_info['input\_mean'],

model\_info['input\_std'])

if do\_distort\_images:

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

(distorted\_jpeg\_data\_tensor,

distorted\_image\_tensor) = add\_input\_distortions(

FLAGS.flip\_left\_right, FLAGS.random\_crop, FLAGS.random\_scale,

FLAGS.random\_brightness, model\_info['input\_width'],

model\_info['input\_height'], model\_info['input\_depth'],

model\_info['input\_mean'], model\_info['input\_std'])

else:

# We'll make sure we've calculated the 'bottleneck' image summaries and

# cached them on disk.

cache\_bottlenecks(sess, image\_lists, FLAGS.image\_dir,FLAGS.bottleneck\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor, FLAGS.architecture)

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

(train\_step, cross\_entropy, bottleneck\_input, ground\_truth\_input,

final\_tensor) = add\_final\_training\_ops( len(image\_lists.keys()),FLAGS.final\_tensor\_name, bottleneck\_tensor, model\_info['bottleneck\_tensor\_size'])

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

evaluation\_step, prediction = add\_evaluation\_step(final\_tensor, ground\_truth\_input)

# Merge all the summaries and write them out to the summaries\_dir

merged = tf.summary.merge\_all()

train\_writer = tf.summary.FileWriter(FLAGS.summaries\_dir + '/train',sess.graph)

validation\_writer = tf.summary.FileWriter(FLAGS.summaries\_dir + '/validation')

# Set up all our weights to their initial default values.

init = tf.global\_variables\_initializer()

sess.run(init)

# Run the training for as many cycles as requested on the command line.

for i in range(FLAGS.how\_many\_training\_steps):

# Get a batch of input bottleneck values, either calculated fresh every

# time with distortions applied, or from the cache stored on disk.

if do\_distort\_images:

(train\_bottlenecks,train\_ground\_truth) = get\_random\_distorted\_bottlenecks(sess, image\_lists, FLAGS.train\_batch\_size, 'training',FLAGS.image\_dir,distorted\_jpeg\_data\_tensor, distorted\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor)

else:

(train\_bottlenecks,train\_ground\_truth, \_) = get\_random\_cached\_bottlenecks(sess, image\_lists, FLAGS.train\_batch\_size, 'training', FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor,FLAGS.architecture)

# Feed the bottlenecks and ground truth into the graph, and run a training

# step. Capture training summaries for TensorBoard with the `merged` op.

train\_summary, \_ = sess.run( [merged, train\_step],feed\_dict={bottleneck\_input: train\_bottlenecks, ground\_truth\_input: train\_ground\_truth})

train\_writer.add\_summary(train\_summary, i)

# Every so often, print out how well the graph is training.

is\_last\_step = (i + 1 == FLAGS.how\_many\_training\_steps)

if (i % FLAGS.eval\_step\_interval) == 0 or is\_last\_step: train\_accuracy, cross\_entropy\_value = sess.run( [evaluation\_step, cross\_entropy], feed\_dict={bottleneck\_input: train\_bottlenecks,

ground\_truth\_input: train\_ground\_truth})

tf.logging.info('%s: Step %d: Train accuracy = %.1f%%' %(datetime.now(), i, train\_accuracy \* 100))

tf.logging.info('%s: Step %d: Cross entropy = %f' %(datetime.now(), i, cross\_entropy\_value))

validation\_bottlenecks, validation\_ground\_truth, \_ = (get\_random\_cached\_bottlenecks(

sess, image\_lists, FLAGS.validation\_batch\_size, 'validation', FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor, FLAGS.architecture))

# Run a validation step and capture training summaries for TensorBoard

# with the `merged` op.

validation\_summary, validation\_accuracy = sess.run([merged, evaluation\_step],

feed\_dict={bottleneck\_input: validation\_bottlenecks, ground\_truth\_input: validation\_ground\_truth})

validation\_writer.add\_summary(validation\_summary, i)

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

# Store intermediate results

intermediate\_frequency = FLAGS.intermediate\_store\_frequency

if (intermediate\_frequency > 0 and (i % intermediate\_frequency == 0)and i > 0):

intermediate\_file\_name = (FLAGS.intermediate\_output\_graphs\_dir +

'intermediate\_' + str(i) + '.pb')

tf.logging.info('Save intermediate result to : ' +intermediate\_file\_name)

save\_graph\_to\_file(sess, graph, intermediate\_file\_name)

# We've completed all our training, so run a final test evaluation on

# some new images we haven't used before.

test\_bottlenecks, test\_ground\_truth, test\_filenames = (get\_random\_cached\_bottlenecks(sess, image\_lists, FLAGS.test\_batch\_size, 'testing', FLAGS.bottleneck\_dir, FLAGS.image\_dir, jpeg\_data\_tensor, decoded\_image\_tensor, resized\_image\_tensor, bottleneck\_tensor,FLAGS.architecture))

test\_accuracy, predictions = sess.run([evaluation\_step, prediction], feed\_dict={bottleneck\_input: test\_bottlenecks,ground\_truth\_input: test\_ground\_truth})

tf.logging.info('Final test accuracy = %.1f%% (N=%d)' %(test\_accuracy \* 100, len(test\_bottlenecks)))

if FLAGS.print\_misclassified\_test\_images:

tf.logging.info('=== MISCLASSIFIED TEST IMAGES ===')

for i, test\_filename in enumerate(test\_filenames):

if predictions[i] != test\_ground\_truth[i].argmax():

tf.logging.info('%70s %s' %(test\_filename, list(image\_lists.keys())[predictions[i]]))

# Write out the trained graph and labels with the weights stored as

# constants.

save\_graph\_to\_file(sess, graph, FLAGS.output\_graph)

with gfile.FastGFile(FLAGS.output\_labels, 'w') as f:

f.write('\n'.join(image\_lists.keys()) + '\n')

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument( '--image\_dir',type=str,default='', help='Path to folders of labelled images.’)

parser.add\_argument( '--output\_graph',type=str,default='/tmp/output\_graph.pb',

help='Where to save the trained graph.' )

parser.add\_argument( '--intermediate\_output\_graphs\_dir', type=str,

default='/tmp/intermediate\_graph/', help='Where to save the intermediate graphs.' )

parser.add\_argument('--intermediate\_store\_frequency',type=int,default=0, help="""\

How many steps to store intermediate graph. If "0" then will not store.\""")

parser.add\_argument('--output\_labels',type=str,default='/tmp/output\_labels.txt', help='Where to save the trained graph\'s labels.')

parser.add\_argument( '--summaries\_dir',type=str,default='/tmp/retrain\_logs',

help='Where to save summary logs for TensorBoard.' )

parser.add\_argument( '--how\_many\_training\_steps',type=int,default=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.\ """ )

parser.add\_argument( '--architecture',type=str,default='inception\_v3',

help="""\Which model architecture to use. 'inception\_v3' is the most accurate, but

also the slowest. For faster or smaller models, chose a MobileNet with the form parameter size>\_<input\_size>[\_quantized]'. For example, 'mobilenet\_1.0\_224' will pick a model that is 17 MB in size and takes 224 pixel input images, while 'mobilenet\_0.25\_128\_quantized' will choose a much less accurate, but smaller and faster network that's 920 KB on disk and takes 128x128 images. """)

FLAGS, unparsed = parser.parse\_known\_args()

tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

**3.3.2 Detailed analysis of Retrain.py**

TensorFlow is a multipurpose machine learning framework. TensorFlow can be used anywhere from training huge models across clusters in the cloud, to running models locally on an embedded system like your phone.We will be using Tensoflow Environment for training.

We're only training the final layer of the mobilenets network, so training will end in a reasonable amount of time.

python -m scripts.retrain \

--bottleneck\_dir=tf\_files/bottlenecks \

--how\_many\_training\_steps=500 \

--model\_dir=tf\_files/models/ \

--summaries\_dir=tf\_files/training\_summaries/"${ARCHITECTURE}" \

--output\_graph=tf\_files/retrained\_graph.pb \

--output\_labels=tf\_files/retrained\_labels.txt \

--architecture="${ARCHITECTURE}" \

--image\_dir=tf\_files/photos

This script downloads the pre-trained model, adds a new final layer, and trains that layer on the photos we've downloaded

The retraining script has several other command line options you can use:-

* -**-image\_dir**

Path to folders of labeled images.

* **--output\_graph**

Where to save the trained graph.

* **--intermediate\_output\_graphs\_dir**

Where to save the intermediate graphs.

* **--intermediate\_store\_frequency**

How many steps to store intermediate graph. If "0"then will not store

* **--output\_labels:**

Where to save the trained graph's labels.

* **--summaries\_dir:**

Whereto save summary logs for TensorBoard.

* **--how\_many\_training\_steps:**

How many training steps to run before ending

* **--learning\_rate:**

How large a learning rate to use when training

* **--testing\_percentage:**

What percentage of images to use as a test set

* **--validation\_percentage:**

What percentage of images to use as a validation set

* **--eval\_step\_interval:**

How often to evaluate the training results

* **--train\_batch\_size:**

How many images to train on at a time

* **--test\_batch\_size:**

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.

* **--validation\_batch\_size:**

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.

* **--print\_misclassified\_test\_images:**

Whether to print out a list of all misclassified test images.

* **--model\_dir:**

Path to classify\_image\_graph

* **--bottleneck\_dir:**

Path to cache bottleneck layer values as files.

* **--final\_tensor\_name:**

The name of the output classification layer in the Retrained graph.

* **--random\_crop:**

A percentage determining how much of a margin torandomly crop off the training images

* **--flip\_left\_right:**

Whether to randomly flip half of the training images horizontally

* **--random\_scale:**

A percentage determining how much to randomly scale up the size of the training images by.

* **--random\_brightness:**

A percentage determining how much to randomly multiple the training image input pixels up or down by

* **--architecture:**

Which model architecture to use

The --learning\_rate parameter controls the magnitude of the updates to the final layer during training. So far we have left it out, so the program has used the default learning\_rate value of 0.01. If you specify a small learning\_rate, like 0.005, the training will take longer, but the overall precision might increase. Higher values of learning\_rate, like 1.0, could train faster, but typically reduces precision, or even makes training unstable.

One of the things the script does under the hood when path to a folder of images is provided is divide them up into three different sets. The largest is usually the training set, which are all the images fed into the network during training, with the results used to update the model's weights. A big potential problem when we're doing machine learning is that our model may just be memorizing irrelevant details of the training images to come up with the right answers. For example, a network remembering a pattern in the background of each photo it was shown, and using that to match labels with objects. It could produce good results on all the images it's seen before during training, but then fail on new images because it's not learned general characteristics of the objects, just memorized unimportant details of the training images.

This problem is known as overfitting, and to avoid it we keep some of our data out of the training process, so that the model can't memorize them. We then use those images as a check to make sure that overfitting isn't occurring, since if we see good accuracy on them it's a good sign the network isn't overfitting. The usual split is to put 80% of the images into the main training set, keep 10% aside to run as validation frequently during training, and then have a final 10% that are used less often as a testing set to predict the real-world performance of the classifier.These ratios can be controlled using the  testing\_percentage  and  validation\_percentage  flags.

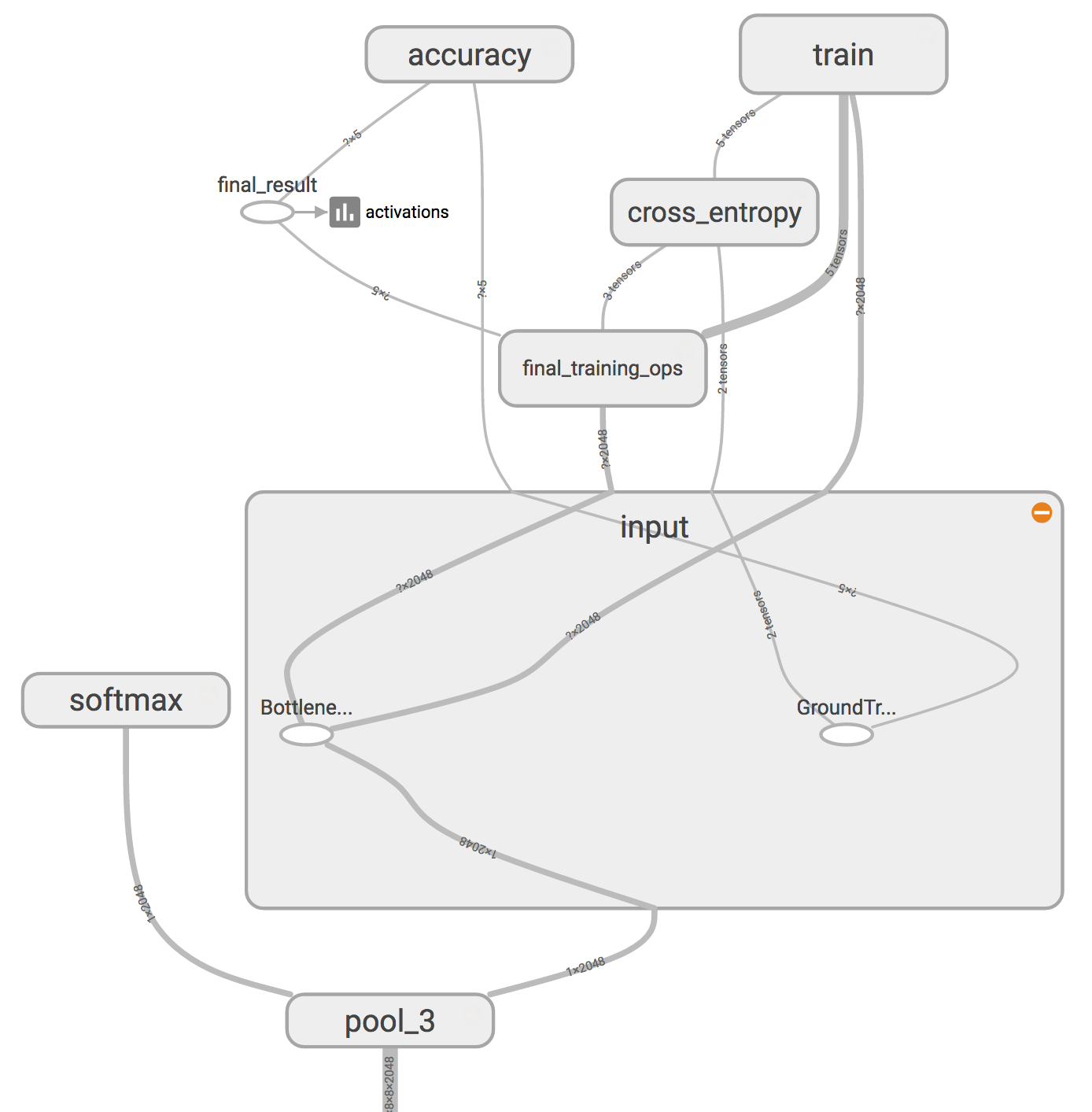
The script uses the image filenames (rather than a completely random function) to divide the images among the training, validation, and test sets. This is done to ensure that images don't get moved between training and testing sets on different runs, since that could be a problem if images that had been used for training a model were subsequently used in a validation set.

The validation accuracy fluctuates among iterations. Much of this fluctuation arises from the fact that a random subset of the validation set is chosen for each validation accuracy measurement. The fluctuations can be greatly reduced, at the cost of some increase in training time, by choosing --validation\_batch\_size=-1, which uses the entire validation set for each accuracy computation.

The Retrain script loads the pre-trained module and trains a new classifier on top for the flower photos you've downloaded.. The magic of transfer learning is that lower layers that have been trained to distinguish between some objects can be reused for many recognition tasks without any alteration.

The script can take thirty minutes or more to complete, depending on the speed of the machine. The first phase analyzes all the images on disk and calculates and caches the bottleneck values for each of them.

These ImageNet models are made up of many layers stacked on top of each other, These layers are pre-trained and are already very valuable at finding and summarizing information that will help classify most images. While all the previous layers retain their already-trained state.



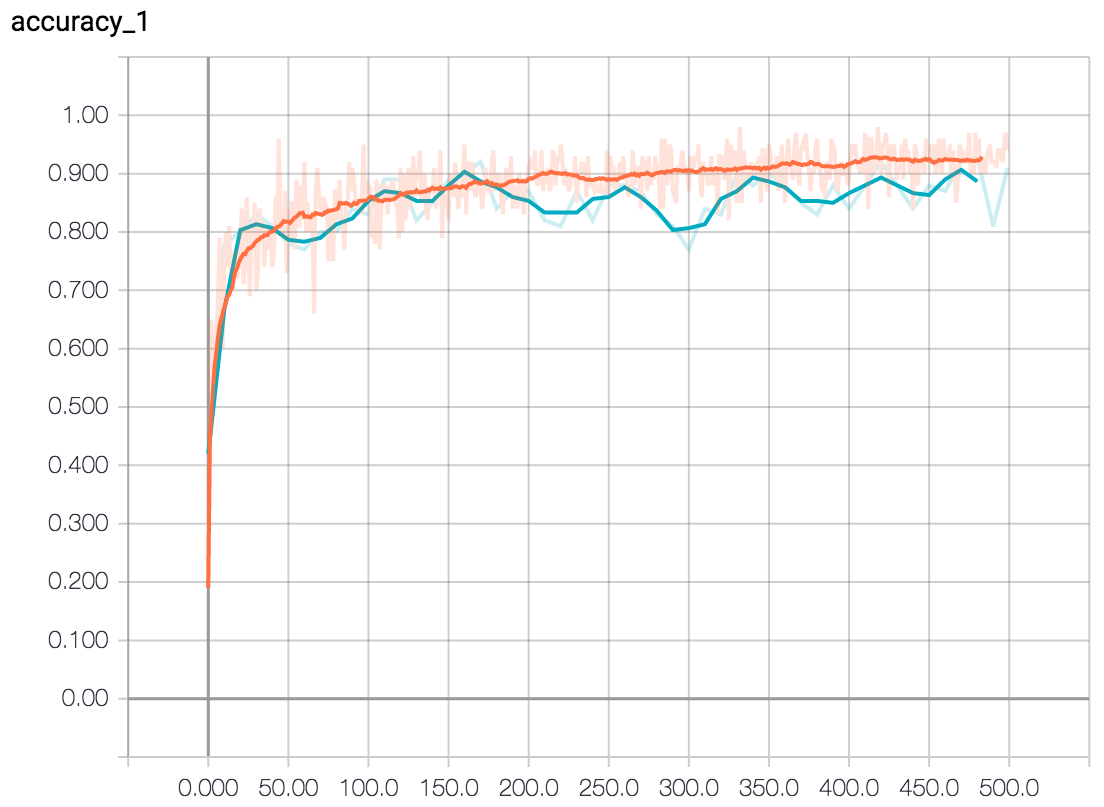
In the above figure, the node labeled "softmax", on the left side, is the output layer of the original model. While all the nodes to the right of the "softmax" were added by the retraining script.

'Bottleneck' is an informal term we often use for the layer just before the final output layer that actually does the classification. "Bottleneck" is not used to imply that the layer is slowing down the network. We use the term bottleneck because near the output, the representation is much more compact than in the main body of the network. (TensorFlow Hub calls this an "image feature vector".) This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize. That means it has to be a meaningful and compact summary of the images, since it has to contain enough information for the classifier to make a good choice in a very small set of values. The reason our final layer retraining can work on new classes is that it turns out the kind of information needed to distinguish between all the 1,000 classes in ImageNet is often also useful to distinguish between new kinds of objects.

Because every image is reused multiple times during training and calculating each bottleneck takes a significant amount of time, it speeds things up to cache these bottleneck values on disk so they don't have to be repeatedly recalculated. By default they're stored in the /tmp/bottleneck directory, and if the script is rerun they’ll be reused.

Once the bottlenecks are complete, the actual training of the top layer of the network begins.. The training accuracy shows what percent of the images used in the current training batch were labeled with the correct class. The validation accuracy is the precision on a randomly-selected group of images from a different set. The key difference is that the training accuracy is based on images that the network has been able to learn from so the network can overfit to the noise in the training data. A true measure of the performance of the network is to measure its performance on a data set not contained in the training data -- this is measured by the validation accuracy. If the train accuracy is high but the validation accuracy remains low, that means the network is overfitting and memorizing particular features in the training images that aren't helpful more generally. Cross entropy is a loss function which gives a glimpse into how well the learning process is progressing (Lower numbers are better.) .The training's objective is to make the loss as small as possible, the learning is working if the loss keeps trending downwards, ignoring the short-term noise.

By default this script will run 4,000 training steps. Each step chooses ten images at random from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions. Those predictions are then compared against the actual labels to update the final layer's weights through the back-propagation process.



Two lines are shown. The orange line shows the accuracy of the model on the training data. While the blue line shows the accuracy on the test set (which was not used for training). This is a much better measure of the true performance of the network. If the training accuracy continues to rise while the validation accuracy decreases then the model is said to be "overfitting". Overfitting is when the model begins to memorize the training set instead of understanding general patterns in the data.

As the process continues the reported accuracy improve, and after all the steps are done, a final test accuracy evaluation is run on a set of images kept separate from the training and validation pictures. This test evaluation is the best estimate of how the trained model will perform on the classification task. The accuracy value of between 90% and 95% will be seen, though the exact value will vary from run to run since there's randomness in the training process. This number is based on the percent of the images in the test set that are given the correct label after the model is fully trained.

Once training is complete,Misclassified images in the test set can be examined by adding the flag --print\_misclassified\_test\_images. This is helpful in knowing which types of images were most confusing for the model, and which categories were most difficult to distinguish. For instance, some subtype of a particular category, or some unusual photo angle, is particularly difficult to identify, which when more training images of that subtype is added will be rectified. Oftentimes, examining misclassified images can also point to errors in the input data set, such as mislabelled, low-quality, or ambiguous images. However, one should generally avoid point-fixing individual errors in the test set, since they are likely to merely reflect more general problems in the (much larger) training set.

The --learning\_rate controls the magnitude of the updates to the final layer during training. Intuitively if this is smaller than the learning will take longer, but it can end up helping the overall precision. The --train\_batch\_size controls how many images are examined during each training step to estimate the updates to the final layer.

Results can be improved by altering the details of the learning process. The simplest one to try is --how\_many\_training\_steps. This defaults to 4,000, but if it is increased to 8,000 it will train for twice as long. The rate of improvement in the accuracy slows the longer you train for, and at some point will stop altogether (or even go down due to overfitting).

**3.4 After Training:-**

python label\_image.py \  
--graph=/tmp/output\_graph.pb --labels=/tmp/output\_labels.txt \  
--input\_layer=Placeholder \  
--output\_layer=final\_result \  
--image=$HOME/images/daisy/21652746\_cc379e0eea\_m.jpg

Since the model uses floating-point weights or activations, it may be possible to reduce the size of model up to ~4x by using quantization, which effectively turns the float weights to 8-bit. There are two flavours of quantization: post-training quantization and quantized training. The former does not require model re-training, but, in rare cases, may have accuracy loss. When accuracy loss is beyond acceptable thresholds, quantized training should be used instead.

TensorFlow approaches the conversion of floating-point arrays of numbers into 8-bit representations as a compression problem. Since the weights and activation tensors in trained neural network models tend to have values that are distributed across comparatively small ranges (e.g. -15 to +15 for weights or -500 to 1000 for image model activations).

Since neural networks tend to be robust at handling noise, the error introduced by quantizing to a small set of values maintains the precision of the overall results within an acceptable threshold. A chosen representation must perform fast calculations, especially with large matrix multiplications that comprise the bulk of the computations while running a model.

This is represented with two floats that store the overall minimum and maximum values corresponding to the lowest and highest quantized value. Each entry in the quantized array represents a float value in that range, distributed linearly between the minimum and maximum.

With our post-training quantization tooling, we use symmetric quantization for our weights, meaning we expand the represented range and force the min and max to be the negative of each other.

For example, with an overall minimum of -10.0 and a maximum of 30.0f, we instead represent a minimum of -30.0 and maximum of 30.0f. In an 8-bit array, the quantized values would be represented as follows:

|  |  |
| --- | --- |
| Quantized | Float |
| -42 | -10.0 |
| 0 | 0 |
| 127 | 30.0 |
| -127 | 30.0 |

**The advantages of this representation format are:**

* It efficiently represents an arbitrary magnitude of ranges.
* The linear spread makes multiplications straightforward.
* A symmetric range for weights enables downstream hardware optimizations.