CLASSIFICATION OF OBJECTS IN MULTIPLE LANGUAGES

A major project report submitted for the partial fulfillment of the requirements for the Degree

Of

B.Tech

By

ANSHUMAN SEKHAR DASH-1501105478
PRUTHIRAJ MARNDI-1501105496
SWAGAT DORA-1501105509
SANJAY TUDU-1501105501

In
Under the Guidance of

Prof. D.K. Swain
Dept of CSEA, IGIT Sarang

Department of Computer Science, Engineering & Applications

INDIRA GANDHI INSTITUTE OF TECHNOLOGY,SARANG
DHENKANAL,ODISHA-759146

Abstract

This project was an attempt at developing an object detection, classification and language translation system using modern computer vision technology and artificial intelligence. The project delivers an android application that is used to detects and classify objects in any speaking language.

In today's era of Globalization, people travel to many different countries for many different purposes whether it is business or simply as a tourist. For an individual learning the names of different objects, cuisines and dishes found across the world is a tedious task. This project is to help users overcome this problem with just the use of a Smartphone. The project will use Object detection Technique using Tensorflow to classify objects and then google translate api to get the classification of objects in many different languages.

It can be used in Classification of Different Foods in User's Local Languages and Object Detection and Classification into different languages.

The Application will be Developed using android studio and TensorFlow platform.The object will be detected using a smartphone camera and fed into the Tensorflow model which will process and give out the results.

Keyword: Detection,Tensorflow,Computer Vision,MobileNet,Deep Learning

References

1. https://link.springer.com/chapter/10.1007/978-3-642-36124-1_23 Research on Computer Vision-Based Object Detection and Classification
2. Research on Computer Vision-Based Object Detection and Classification by-Juan Wu , Bo Peng , Zhenxiang Huang , Jietao Xie

ACKNOWLEDGEMENTS

It is a great pleasure and privilege to express our profound sense of gratitude to Prof. D.K. Swain, Dept. of CSEA, IGIT, Sarang, our guide for his suggestions, motivation and support during the major project preparation and keen personal interest throughout the progress of my course work.

We would like to express our sincere thanks to Prof. (Dr.) S. N. Mishra, HOD, Dept. of CSEA, IGIT, Sarang, for his suggestion, motivation and support during the major project preparation.

We have been fortunate enough to have all support motivation from our parents. We sincerely express our thanks to the other faculties of the department of CSEA, IGIT, Sarang for their positive and constructive suggestion and kind cooperation during major project preparation and presentation.

.

ANSHUMAN SEKHAR DASH

PRUTHIRAJ MARNDI

SWAGAT DORA

SANJAY TUDU

DECLARATION BY THE SCHOLAR

We hereby declare that the synopsis entitled **"Classification of Objects in Multiple Langauges"** *under BPUT, ROURKELA, Odisha at* ***Indira Gandhi Institute Technology, Sarang, Dhenkanal, Odisha*** *is an authentic record of our work carried out under the supervision* ***D.K. SWAIN, Dept. of CSEA, IGIT, Sarang.*** *We have not submitted this major project report elsewhere for any other degree or diploma.*

ANSHUMAN SEKHAR DASH

PRUTHIRAJ MARNDI

SWAGAT DORA

SANJAY TUDU

Department of CSEA,

IGIT,Sarang

INDIRA GANDHI INSTITUTE OF TECHNOLOGY
SARANG

Certificate

*This is to certify that this project entitled "**Object Classification in Multiple Languages**" submitted by **Anshuman Sekhar Dash** (1501105478)**,Pruthiraj Marndi** (1501105496)**, Swagat Dora** (1501105509) **and Sanjay Tudu** (1501105501) of Computer Science Engineering & Applications Department, Indira Gandhi Institute of Technology, Sarang for the partial fulfillment of the requirements for the award of Bachelor of Technology (Computer Science & Engineering) Degree of BPUT, Odisha, is a record of students own study carried under our supervision & guidance.*

This report has not been submitted to any other university or institution for the award of any degree.

Date:6th April 2019

D.K. Swain,
Asst. Professor

(Dr. S. N. Mishra)
Professor & Head,
Dept. of CSEA, IGIT, Sarang
Dhenkanal, Odisha

S.K. Sahoo,
Asst. Professor

S.K. Patra,
Asst. Professor

Signature of
External Examiner

LIST OF TABLE

LIST OF FIGURES

LIST OF ABBREVIATIONS

App	Android Application
UI	User Interface
EULA	End User License Agreement
UX	User Experience
OpenCV	Open Computer Vision

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Short Description

This Project is based on Existing Object Detection and Classification Techniques present but Instead of Classifying the Objects into any particular Language (for e.g. English) the Android Application Developed during this project work can classify in more than one Speaking Language i.e. English,Odia,Hindi etc. The product specified belongs to the class of object detection applications but unlike other object detecting applications emphasis was put on classifying objects according to the user's language choice in order to get more specific results. The android application is developed using android studio. For the deep learning object classifier model the concept of transfer learning is used and the mobilenet model is further trained using tensorflow deep learning environment which is then used inside the android application. The android application processes the camera feed frame by frame which is then fed to the deep net model to get the label Text. The label text is matched with the language chosen and the translated text is then displayed in the text box. If in any case the object is wrongly classified a feedback option is provided to report errors.

Upon clicking the feedback option the frame image is saved and the image along with the correct labelled text is upload to firebase cloud storage using the firebase libraries. These images will be used for further testing.

1.2 Product Functions

Product Functions	**Description**
Scan for Objects	Look around for objects that can be classified when the user opens the application.
Accept the object	Get the input from the user of the object that requires to be classified
Process and Classify	Process the object image using the feature map inside the inbuilt Tensorflow model and return the result.
Translate	The Result is translated into the desired language and the result is shown to the user.
Error Reporting	The User reports any error made in the classification for further improvement.

Table-1

1.3 Operating Environment

The Android application will require a Smartphone with an inbuilt camera. The Trained deep learning model will be kept inside the application for offline use. Error Reporting will require internet connectivity to send the reports and corrections. For Images to be clear and precise the application will require ambient light surroundings to operate on.

1.4 Main Parts of the Application Development:-

1. Data Collection
2. Deep Net Model Training
3. Label Translations
4. UI Development
5. Connecting Interfaces

All these steps are discussed in subsequent chapters.

1.5 Tools Used:-

1. **Java**
   To make android application Java Programming Language is used.
   Downloaded from https://www.java.com/download/
2. **Python**
   For training and development of deep learning neural network python along with tensorflow is used.
   Downloaded from https://www.python.org/downloads/
3. **Tensorflow**
   Used for developing the Neural Network with the help of python
   Installed using the command –pip install tensorflow on the command line
4. **Android Studio**
   Android Studio is the official IDE to develop and debug android applications.
   Downloaded from https://developer.android.com/studio
5. **A Smartphone equipped with camera**
   Required to Test and Debug the Application
6. **Firebase**
   Used to store the feedback images for further training and processing.

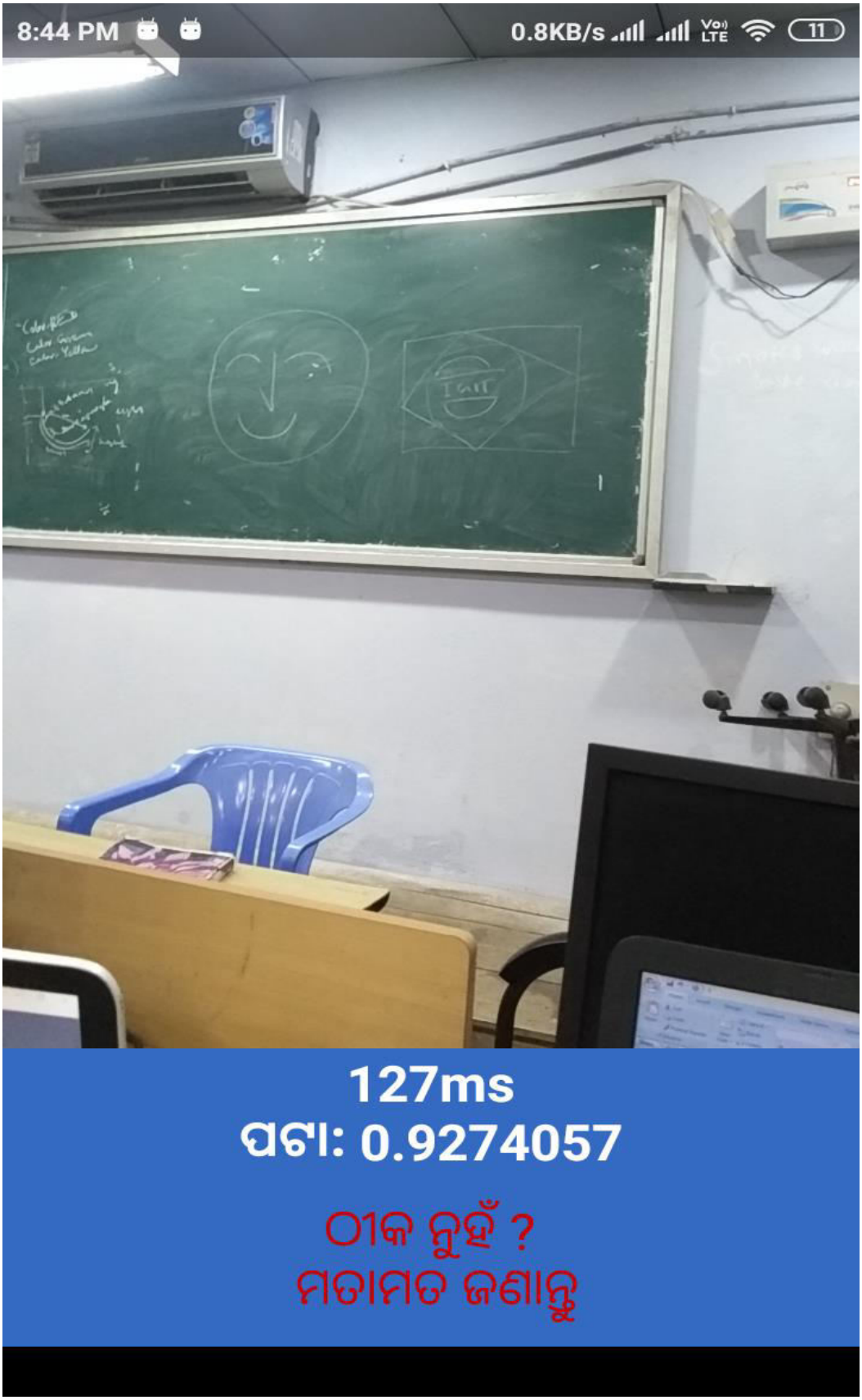

Fig-1

Screenshot of the End Product

CHAPTER 2

PROJECT THEORY

2.1 Transfer Learning:-

The definition of transfer learning is given in terms of domain and task. The domain D consists of: a feature space X and a marginal probability distribution $P(X)$, where $X=\{x1, \dots xn\} \in X$. Given a specific domain, $D=\{X, P(X)\}$, a task consists of two components: a label space Y and an objective predictive function $f(.)$(denoted by $T=\{Y , f(.)\}$ $T=\{Y , f(.)\}$), which is learned from the training data consisting of pairs , which consist of pairs $\{x_i , y_i\}$, where $x_i \in X$ and $y_i \in Y$. The function $f(.)$ can be used to predict the corresponding label, $f(x)$ of a new instance x.

Given a source domain D_S and learning task T_S target domain D_T and learning task T_T transfer learning aims to help improve the learning of the target predictive function $f_T(.)$ in D_T using the knowledge in D_S and T_S where $D_S=D_T$ or $T_S=T_T$.

The concept of Transfer Learning is used for Developing the Deep Learning Model. Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks. This area of research bears some relation to the long history of psychological literature on transfer of learning, although formal ties between the two fields are limited.

Modern image recognition models have millions of parameters. Training them from scratch requires a lot of labelled training data and a lot of computing power (hundreds of GPU-hours or more). Transfer learning is a technique that shortcuts much of this by taking a piece of a model that has already been trained on a related task and reusing it in a new model. In this Project we will reuse the feature extraction capabilities from powerful image classifiers trained on Image Net and simply train a new classification layer on top.

We will be using transfer learning, which means we are starting with a model that has been already trained on another problem. We will then retrain it on a similar problem. Deep learning from scratch can take days, but transfer learning can be done in short order.Though it's not as good as training the full model, this is surprisingly effective for many applications, works with moderate amounts of training data (thousands, not millions of labelled images, and can be run in as little as thirty minutes on a laptop without a GPU.

2.2 Convolution Neural Networks:-

Neural Networks are essentially mathematical models to solve an optimization problem. They are made of neurons, the basic computation unit of neural networks. A neuron takes an input (say x), do some computation on it(say: multiply it with a variable w and adds another variable b) to produce a value (say; z= wx+b). This value is passed to a non-linear function called activation function (f) to produce the final output (activation) of a neuron. There are many kinds of activation functions. One of the popular activation function is Sigmoid, which is:

$$y = \frac{1}{1 + e^{-z}}$$

The neuron which uses sigmoid function as an activation function will be called Sigmoid neuron. Depending on the activation functions, neurons are named and there are many kinds of them like RELU, TanH .One neuron can be connected to multiple neurons, like this:

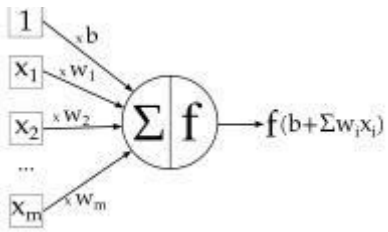

Fig-2

Connection of a Neuron

The original goal of the ANN approach was to solve problems in the same way that a human brain would. However, over time, attention moved to performing specific tasks, leading to deviations from biology. Artificial neural networks have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.

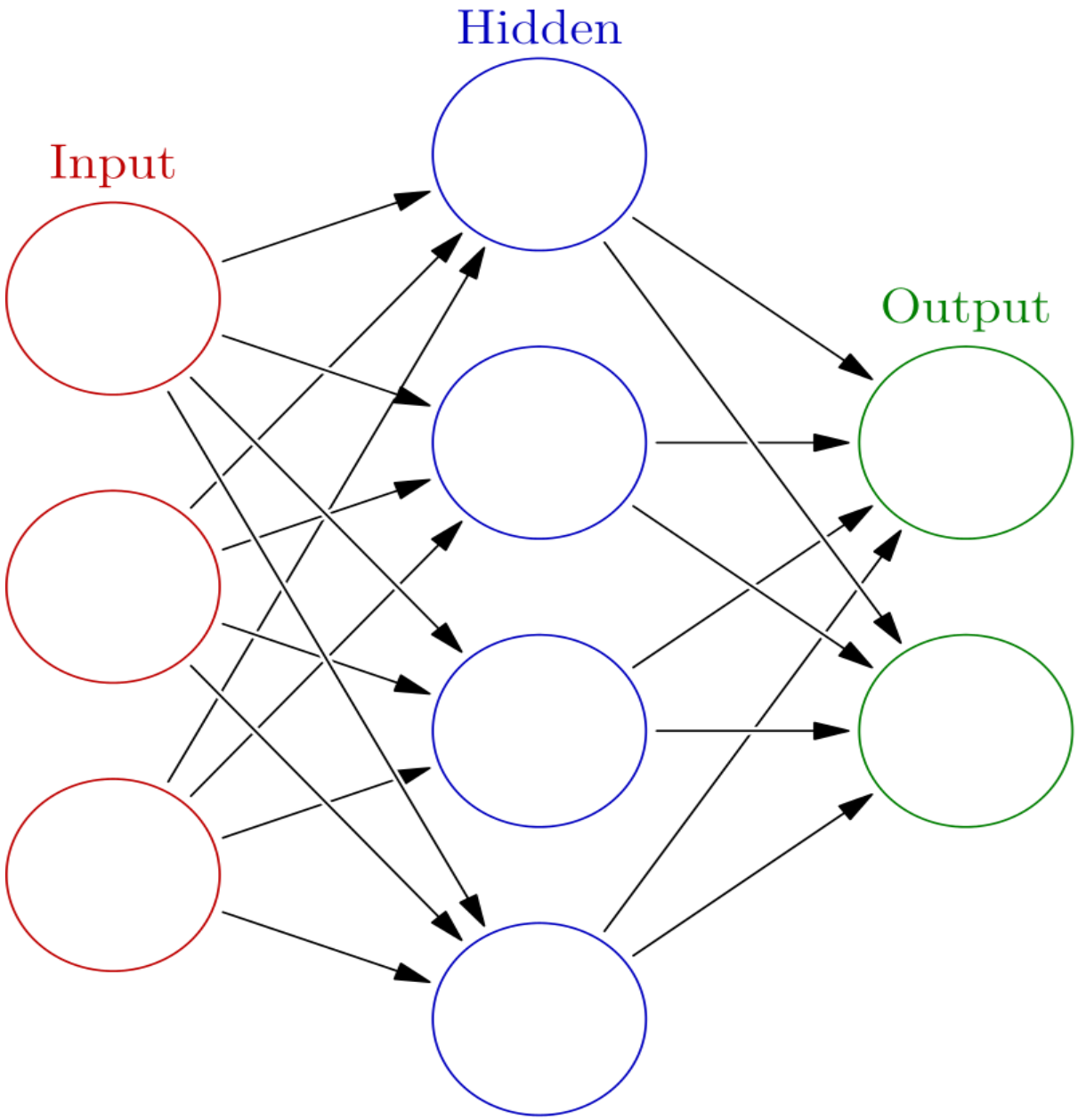

Fig-3

Schematic Diagram of a Neural Network

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

CNNs use a variation of multilayer perceptrons designed to require minimal pre-processing. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.

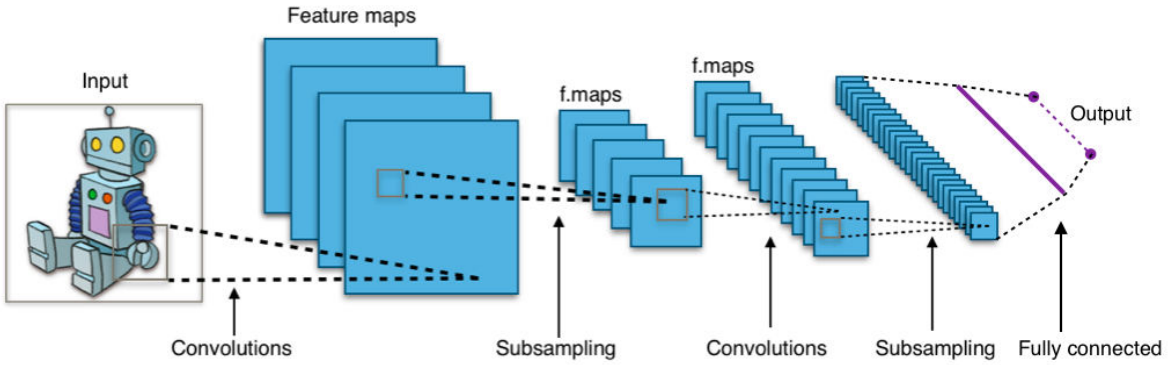

Fig-4

Convolutional Neural Network

Convolutional networks are at the core of most state-of-the-art computer vision solutions for a wide variety of tasks. Since 2014 very deep convolutional networks started to become mainstream, yielding substantial gains in various benchmarks. Although increased model size and computational cost tend to translate to immediate quality gains for most tasks (as long as enough labeled data is provided for training), computational efficiency and low parameter count are still enabling factors for various use cases such as mobile vision and big-data scenarios.

2.3 Mobile Nets:-

MobileNets are based on a streamlined architecture that uses depth-wise separable convolutions to build light weight deep neural networks. This architecture uses depthwise separable convolutions which significantly reduces the number of parameters when compared to the network with normal convolutions with the same depth in the networks. This results in light weight deep neural networks.

The normal convolution is replaced by depth wise convolution followed by point wise convolution which is called as depth wise separable convolution.

In the normal convolution, if the input feature map is of $\mathbf{H_i,W_i,C_i}$ dimension and we want $\mathbf{C_o}$ feature maps with convolution kernel size K then there are $\mathbf{C_o}$ convolution kernels each with dimension $\mathbf{K,K,C_i}$. This results in a feature map of $\mathbf{H_i,W_i,C_o}$ dimension after convolution operation.

In the depth wise separable convolution, if the input feature map is of $\mathbf{H_i,W_i,C_i}$ dimension and we want $\mathbf{C_o}$ feature maps in the resulting feature map and the convolution kernel size is **K** then there are $\mathbf{C_i}$ convolution kernels, one for each input channel, with dimension K,K,1. This results in a feature map of $\mathbf{H_o,W_o,C_i}$ after depth wise convolution. This is followed by point wise convolution [1x1 convolution]. This convolution kernel is of dimension $\mathbf{1,1,C_i}$ and there are $\mathbf{C_o}$ different kernels which results in the feature map of $\mathbf{H_o,W_o,C_o}$ dimension.

This results in the reduction of number of parameters significantly and thereby reduces the total number of floating point multiplication operations which is favourable in mobile and embedded vision applications with less compute power.

By using depth wise separable convolutions, there is some sacrifice of accuracy for low complexity deep neural network.

MobileNets, a family of mobile-first computer vision models for TensorFlow, designed to effectively maximize accuracy while being mindful of the restricted resources for an on-device or embedded application. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of a variety of use cases. They can be built upon for classification, detection, embeddings and segmentation similar to how other popular large scale models, such as Inception, are used.

The Mobile Net is configurable in two ways:

- Input image resolution: 128,160,192, or 224px. Unsurprisingly, feeding in a higher resolution image takes more processing time, but results in better classification accuracy.
- The relative size of the model as a fraction of the largest Mobile Net: 1.0, 0.75, 0.50, or 0.25.

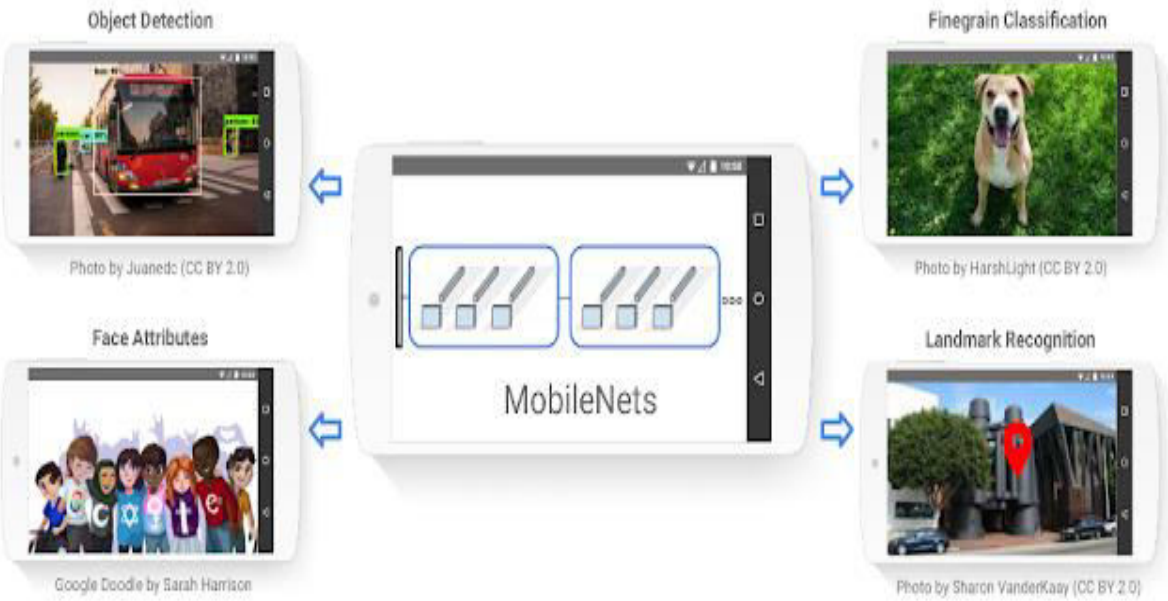

Fig-5

MobileNets Feautures

The ImageNet project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. ImageNet contains more than 20,000 categories with a typical category, such as "balloon" or "strawberry", consisting of several hundred images.

ImageNet models are networks with millions of parameters that can differentiate a large number of classes.

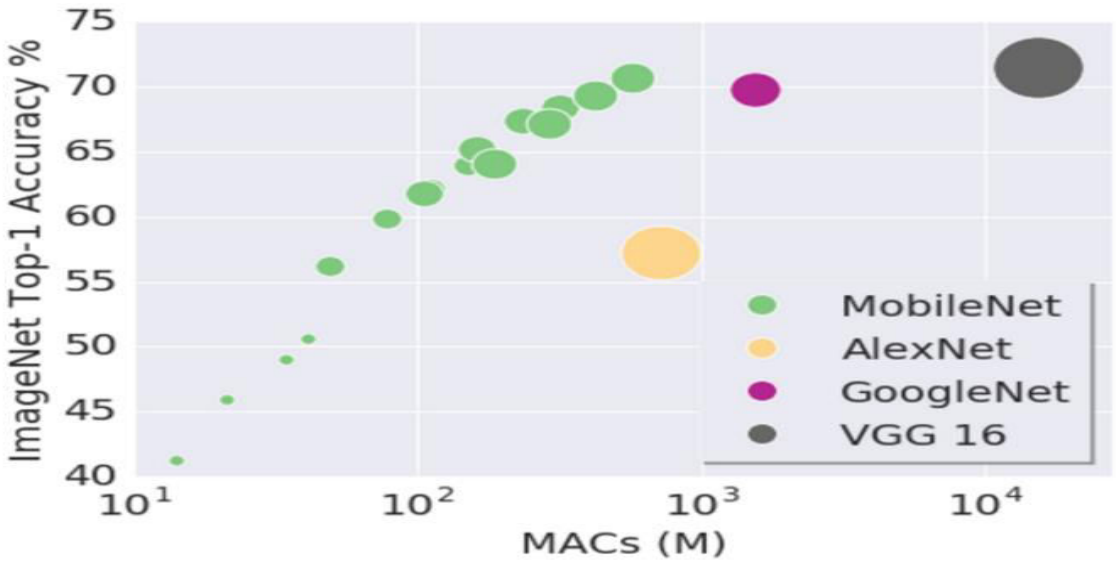

Fig-6

Accuracy % of different Models

The graph above shows the first-choice-accuracies of these configurations (y-axis), vs the number of calculations required (x-axis), and the size of the model (circle area).

16 points are shown for MobileNet. For each of the 4 model sizes (circle area in the figure) there is one point for each image resolution setting. The 128px image size models are represented by the lower-left point in each set, while the 224px models are in the upper right.

Model Checkpoint	Million MACs	Million Parameters	Top-1 Accuracy	Top-5 Accuracy
MobileNet_v1_1.0_224	569	4.24	70.7	89.5
MobileNet_v1_1.0_192	418	4.24	69.3	88.9
MobileNet_v1_1.0_160	291	4.24	67.2	87.5
MobileNet_v1_1.0_128	186	4.24	64.1	85.3
MobileNet_v1_0.75_224	317	2.59	68.4	88.2
MobileNet_v1_0.75_192	233	2.59	67.4	87.3
MobileNet_v1_0.75_160	162	2.59	65.2	86.1
MobileNet_v1_0.75_128	104	2.59	61.8	83.6
MobileNet_v1_0.50_224	150	1.34	64.0	85.4
MobileNet_v1_0.50_192	110	1.34	62.1	84.0
MobileNet_v1_0.50_160	77	1.34	59.9	82.5
MobileNet_v1_0.50_128	49	1.34	56.2	79.6
MobileNet_v1_0.25_224	41	0.47	50.6	75.0
MobileNet_v1_0.25_192	34	0.47	49.0	73.6
MobileNet_v1_0.25_160	21	0.47	46.0	70.7
MobileNet_v1_0.25_128	14	0.47	41.3	66.2

Table-2

MobileNets Version Analysis

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 labelled 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 colour, 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 labelled 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,	6.4 Mb	70.1%	89.8%	154.5 ms

	tflite&pb				
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

Table-3

Analysis of Different Models

Depending on the task, we will need to make a trade-off 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.

Fig-7

Model Size vs Accuracy

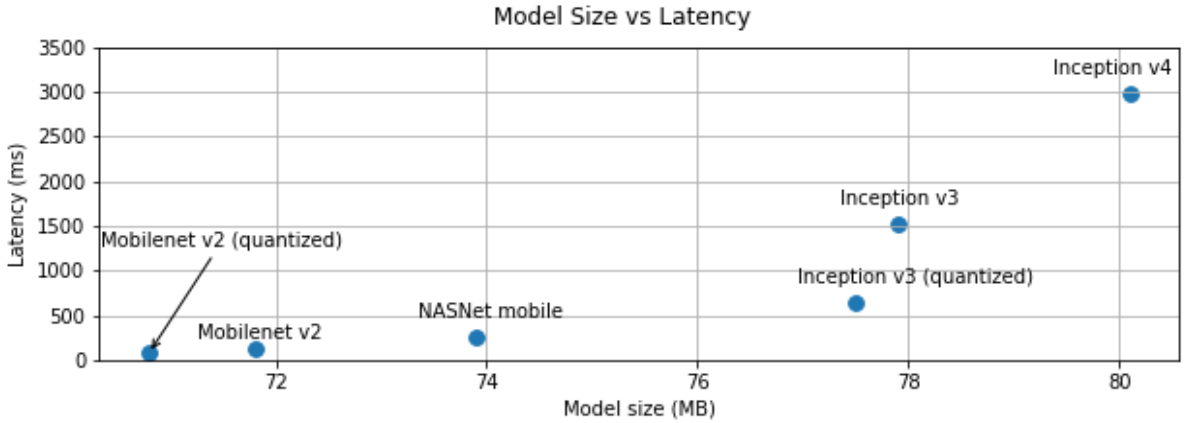

Fig-8

Model Size vs Latency

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

Table-4

Performance of MobileNets on different Device

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(sesss,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  thetraining
    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, butalso 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:**
  Where to 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 to randomly 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.

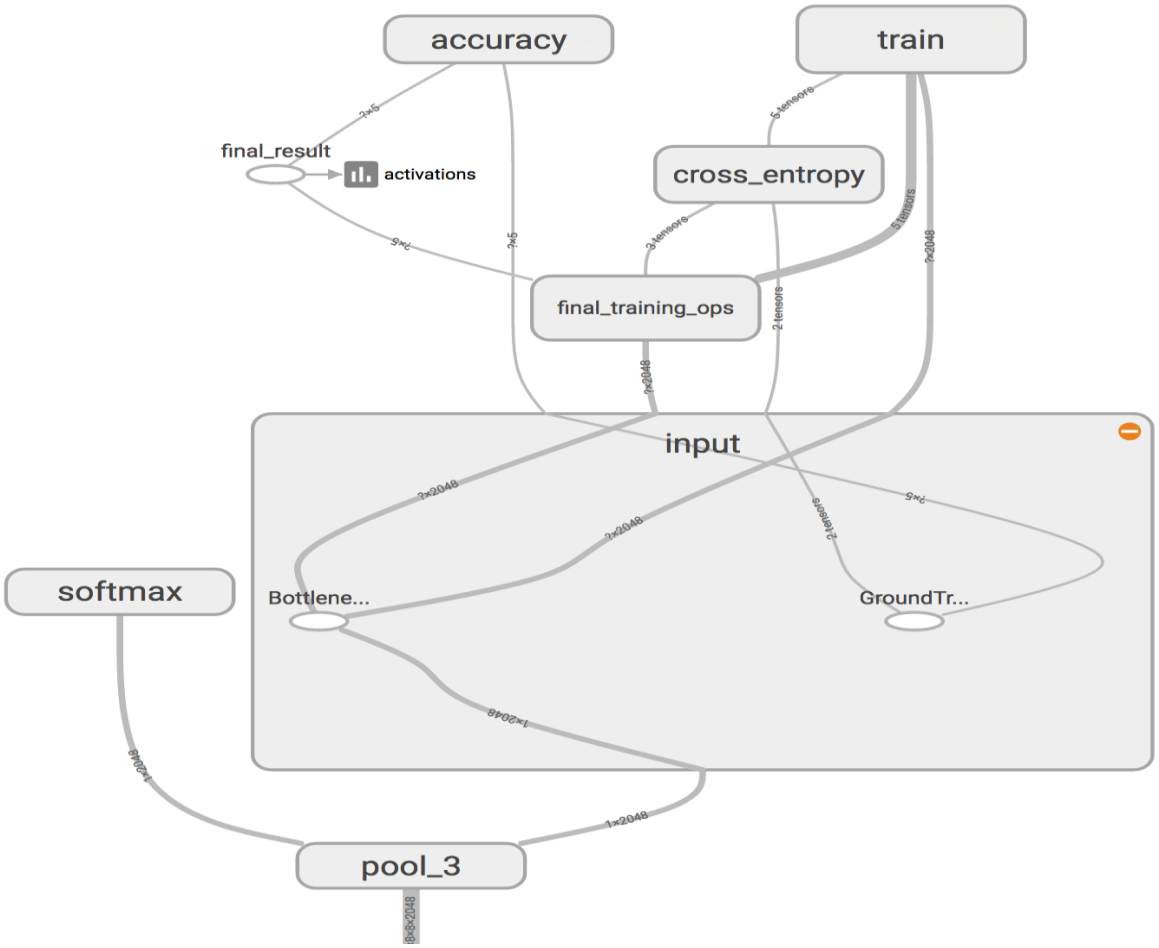

Fig-9

Working of Transfer Learning

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.

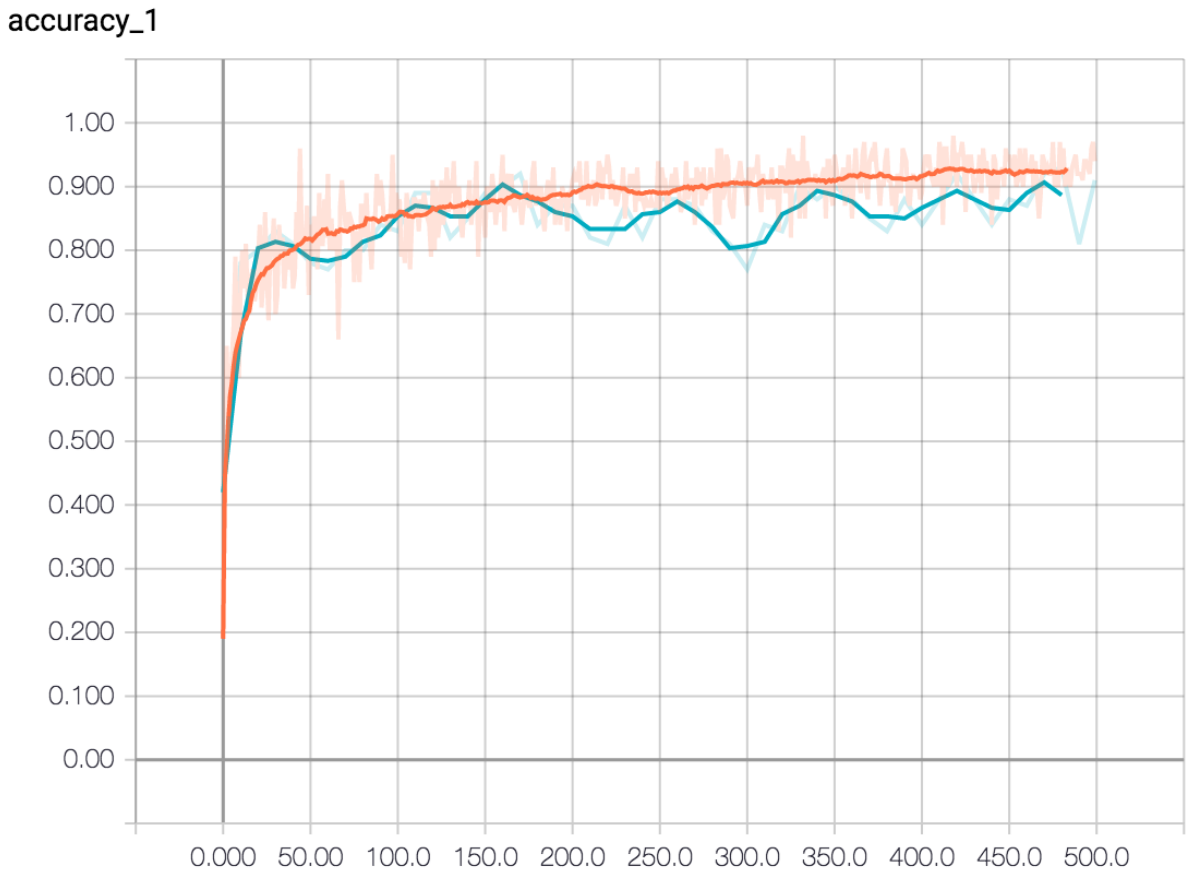

Fig-10

Accuracy on different Data Sets

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

Table-5

Represention of Quantized Values

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.

Chapter 4

Application Development

4.1 Need to For Model Optimization

Mobile devices have significant limitations, so any pre-processing that can be done to reduce an app's footprint is worth considering. Inference efficiency is a critical issue when deploying machine learning models to mobile devices because of the model size, latency, and power consumption.

Computational demand for training grows with the number of models trained on different architectures, whereas the computational demand for inference grows in proportion to the number of users.

Model optimization is useful for:

- Deploying models to edge devices with restrictions on processing, memory, or power-consumption. For example, mobile and Internet of Things (IoT) devices
- Reduce the payload size for over-the-air model updates.
- Execution on hardware constrained by fixed-point operations.
- Optimize models for special purpose hardware accelerators

- Innovation at the silicon layer is enabling new possibilities for hardware acceleration, and frameworks such as the Android Neural Networks API make it easy to leverage these.
- Recent advances in real-time computer-vision and spoken language understanding have led to mobile-optimized benchmark models being open sourced (e.g. MobileNets, SqueezeNet).
- Widely-available smart appliances create new possibilities for on-device intelligence.
- Interest in stronger user data privacy paradigms where user data does not need to leave the mobile device.
- Ability to serve ‘offline’ use cases, where the device does not need to be connected to a network

4.2 Tensorflow Lite

TensorFlow Lite is TensorFlow’s lightweight solution for mobile and embedded devices. It enables on-device machine learning inference with low latency and a small binary size. TensorFlow Lite also supports hardware acceleration with the Android Neural Networks API.

TensorFlow Lite uses many techniques for achieving low latency such as optimizing the kernels for mobile apps, pre-fused activations, and quantized kernels that allow smaller and faster (fixed-point math) models.

TensorFlow Lite supports a set of core operators, both quantized and float, which have been tuned for mobile platforms. They incorporate pre-fused activations and biases to further enhance performance and quantized accuracy. Additionally, TensorFlow Lite also supports using custom operations in models.

TensorFlow Lite defines a new model file format, based on FlatBuffers. FlatBuffers is an efficient open-source cross-platform serialization library. It is similar to protocol buffers, but the primary difference is that FlatBuffers does not need a parsing/unpacking step to a secondary representation before you can access data, often coupled with per-object memory allocation. Also, the code footprint of FlatBuffers is an order of magnitude smaller than protocol buffers.

TensorFlow Lite has a new mobile-optimized interpreter, which has the key goals of keeping apps lean and fast. The interpreter uses a static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

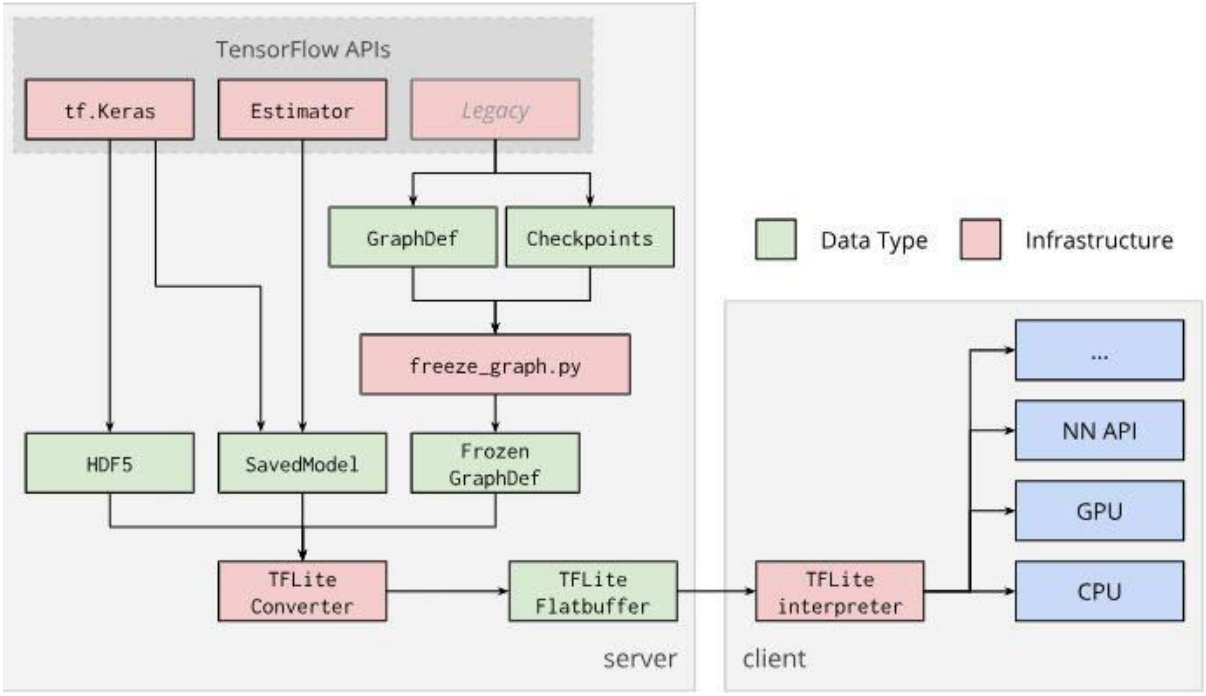

Fig-11

TesorflowLite Architecture

TensorFlow Lite supports multi-threaded kernels for many operators. We may increase the number of threads and speed up execution of operators. Increasing the number of threads will, however, make the model use more resources and power.

For some applications, latency may be more important than energy efficiency. We can increase the number of threads by setting the number of interpreter threads. Multi-threaded execution, however, comes at the cost of increased performance variability depending on what else is executed concurrently. This is particularly the case for mobile apps. For example, isolated tests may show 2x speed-up vs single-threaded, but, if another app is executing at the same time, it may result in worse performance than single-threaded.

4.3 Using Tensorflow Lite

With TFLite a new graph converter is now included with the TensorFlow installation. This program is called the "TensorFlow Lite Optimizing Converter" or tflite_convert. It is installed as a command line script, with TensorFlow.

While tflite_convert has advanced capabilities for dealing with quantized graphs, it also applies several optimizations that are still useful for our graph, (which does not use quantization). These include pruning unused graph-nodes, and performance improvements by joining operations into more efficient composite operations.The pruning is especially helpful given that TFLite does not support training operations yet, so these should not be included in the graph.

While tflite_convert can be used to optimize regular graph.pb files, TFLite uses a different serialization format from regular TensorFlow. TensorFlow uses Protocol Buffers, while TFLite uses FlatBuffers.

The primary benefit of FlatBuffers comes from the fact that they can be memory-mapped, and used directly from disk without being loaded and parsed. This gives much faster startup times, and gives the operating system the option of loading and unloading the required pages from the model file, instead of killing the app when it is low on memory.

The TensorFlow Lite converter generates a TensorFlow Lite FlatBuffer file (.tflite) from a TensorFlow model. TensorFlow Lite uses the optimized FlatBuffer format to represent graphs. Therefore, a TensorFlow model (protocol buffer) needs to be converted into a FlatBuffer file before deploying to clients.

The converter supports the following input formats:

- SavedModels
- Frozen GraphDef: Models generated by freeze_graph.py.
- tf.keras HDF5 models.
- Any model taken from a tf.Session (Python API only).

The TensorFlow Lite FlatBuffer file is then deployed to a client device (generally a mobile or embedded device), and the TensorFlow Lite interpreter uses the compressed model for on-device inference

We can create the TFLite FlatBuffer with the following command:

```
IMAGE_SIZE=224

tflite_convert \

  --graph_def_file=tf_files/retrained_graph.pb \

  --output_file=tf_files/optimized_graph.lite \
```

```
--input_format=TENSORFLOW_GRAPHDEF \
--output_format=TFLITE \
--input_shape=1,${IMAGE_SIZE},${IMAGE_SIZE},3 \
--input_array=input \
--output_array=final_result \
--inference_type=FLOAT \
--input_data_type=FLOAT
```

4.4 Application Developement

This app is made using the following external libraries

- a pre-compiled TFLite Android Archive (AAR). This AAR is hosted on jcenter.
- tensorflow-lite java library
- firebase android Library

The following lines in the module's build.gradle file include the newest version of the AAR, from the TensorFlow bintray maven repository, in the project.

```
repositories {
   maven {
      url 'https://google.bintray.com/tensorflow'
   }
}
dependencies {
   compile 'org.tensorflow:tensorflow-lite:+'
}
```

The following line instantiates a TFLite interpreter. The interpreter does the job of a tf.Session. We pass the interpreter a MappedByteBuffer containing the model. The local function loadModelFile creates a MappedByteBuffer containing the activity's graph.lite asset file

```
ImageClassifier(Activity activity) throws IOException {
        // The following lines load the label list and create the output buffer:
         tflite = new Interpreter(loadModelFile(activity));
        labelList = loadLabelList(activity);
        imgData =ByteBuffer.allocateDirect(  4 * DIM_BATCH_SIZE *
                        DIM_IMG_SIZE_X * DIM_IMG_SIZE_Y * DIM_PIXEL_SIZE);
         imgData.order(ByteOrder.nativeOrder());
         labelProbArray = new float[1][labelList.size()];
        Log.d(TAG, "Created a Tensorflow Lite Image Classifier.");
}
```

In the above code snippet byte buffer is sized to contain the image data once converted to float. The interpreter can accept float arrays directly as input, but the ByteBuffer is more efficient as it avoids extra copies in the interpreter.

The output buffer is a float array with one element for each label where the model will write the output probabilities.

The second block of interest is the classifyFrame method. It takes a Bitmap as input, runs the model and returns the text to print in the app

```
String classifyFrame(Bitmap bitmap) {
        convertBitmapToByteBuffer(bitmap);
        tflite.run(imgData, labelProbArray);
        //K is initialized as 1 since we only require the top result.
        String textToShow = printTopKLabels();
}
```

This method does three things. First converts and copies the input Bitmap to the imgData ByteBuffer for input to the model. Then it calls the interpreter's run method, passing the input buffer and the output array as arguments. The interpreter sets the values in the output array to the probability calculated for each class. The input and output nodes are defined by the arguments to the toco conversion step that created the .lite model file earlier.

The app is resizing each camera image frame (224 width * 224 height) to match the quantized MobileNets model . The resized image is converted—row by row—into a ByteBuffer. Its size is 1 * 224 * 224 * 3 bytes, where 1 is the number of images in a batch. 224 * 224 (299 * 299) is the width and height of the image. 3 bytes represents the 3 colors of a pixel.

The app uses the TensorFlow Lite Java inference API for models which take a single input and provide a single output. This outputs a two-dimensional array, with the first dimension being the category index and the second dimension being the confidence of classification.

The optimized graph file(.tflite) generated using the tflite-convert is kept inside the assets folder of the android project.

TensorFlow Lite inference is the process of executing a TensorFlow Lite model on-device and extracting meaningful results from it. Inference is the final step in using the model on-device in the architecture.Inference for TensorFlow Lite models is run through an interpreter.

TensorFlow Lite inference on device typically follows the following steps.

- Loading a Model
  The process involves loading the .tflite model into memory which contains the model's execution graph.Transforming Data Input data generally may not match the input data format expected by the model.
- Running Inference
  This step involves using the API to execute the model. It involves a few steps such as building the interpreter, and allocating tensors as explained in detail above.
- Interpreting Output
  This process involves retrieval of results from model inference and interpreting the tensors in a meaningful way to be used in the application.
  For example, a model may only return a list of probabilities. It is up to the application developer to meaningully map them to relevant categories and present it to their user.

4.5 Language Specific Changes

The lables.txt file generated after retraining is as shown below

pomegranate	backpack	bell pepper
board	broccoli	corn
daisy	dandelion	desk
flowervase	guava	keys
mango	marigold	mobilephones
onion	papaya	pen
pineapple	podium	potato
pumpkin	purse	roses
spectacles	sunflowers	tulips
wallet	Cap	Bottle

Wristwatch	Wind mill	Ferris wheel

Table-6

Label Names

The above labels are stored as a string array in strings.xml file.This array is the extracted as Arraylist.The classifyframe function then prints Out labels according to postion of the array.The original labels are used as the key and the specific language word is inside the value.

4.6 Feedback

Since the for faster processing and less size a smaller model is used, The accuracy is hampered and the Classification may not be correct.For Correction and Retraining of the model a feedback option is provided.This feedback option takes a snapshot of the object and asks for a label name from the user.The user can enter the name in any of the languages available.This image along with the text entered is uploaded to firebase realtime database for storage and analysis.

Fig-12

Database Tree Structure

4.7 Application Screenshots

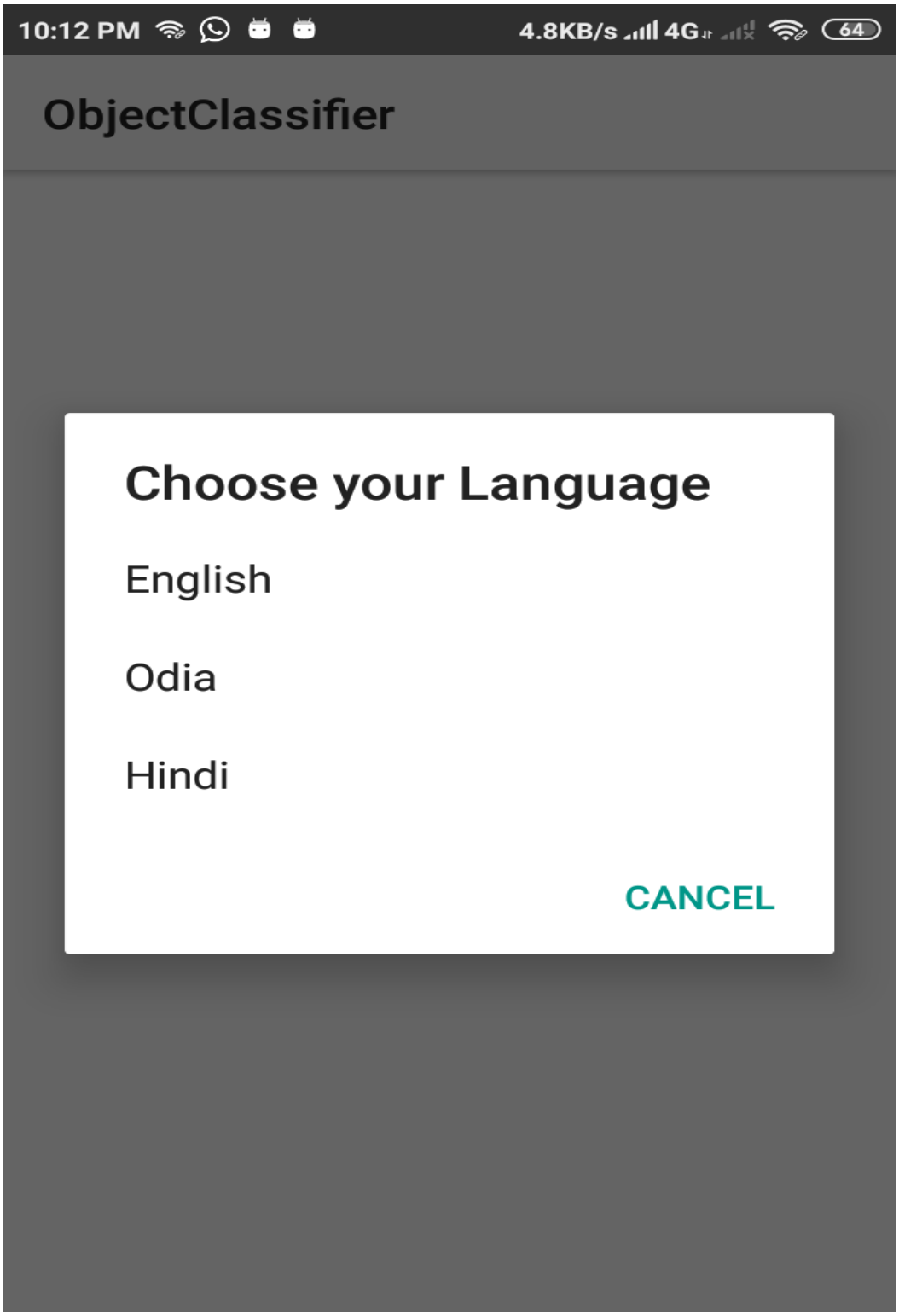

Fig-13

Screenshot of ChooseLangugage Activity

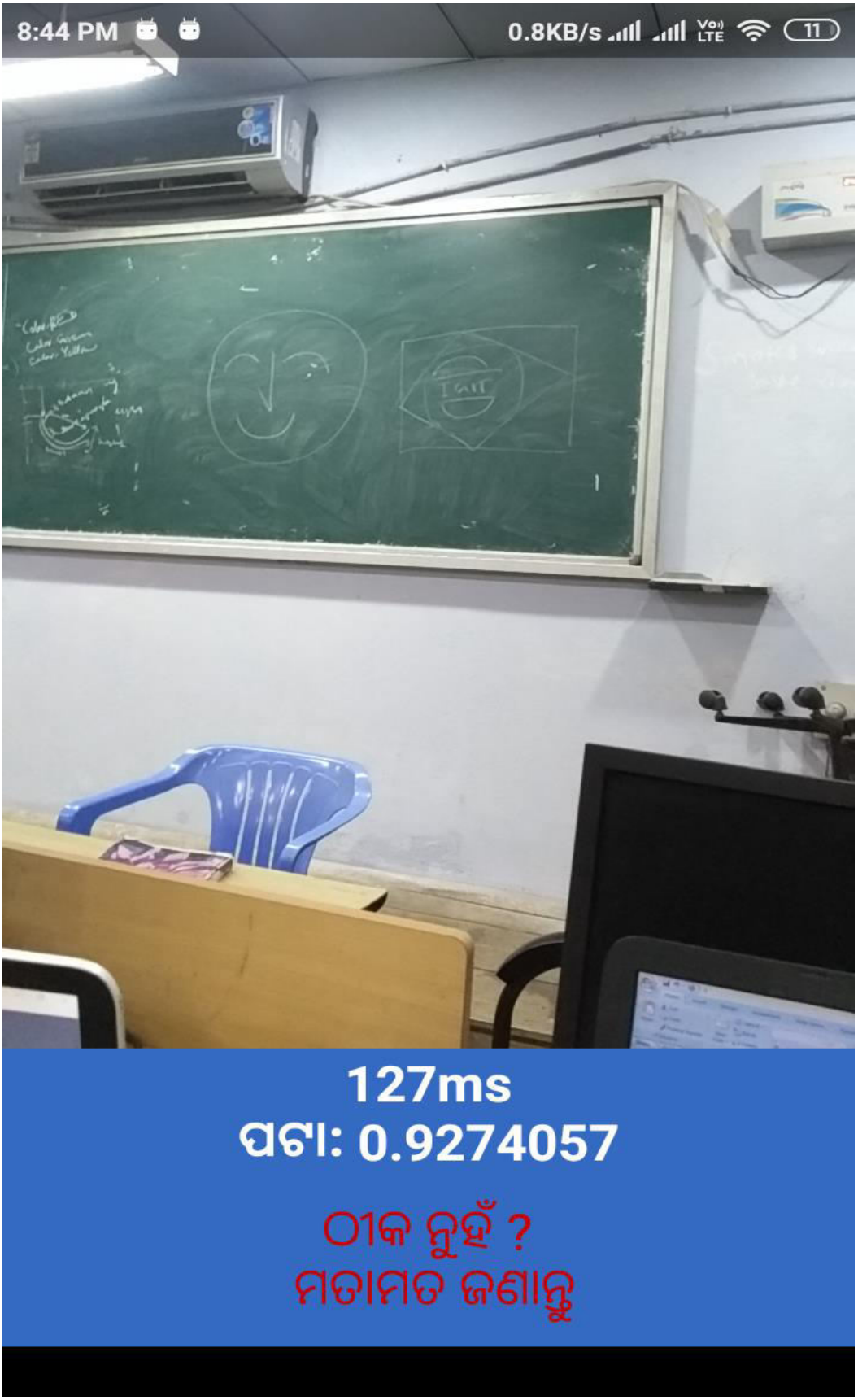

Fig-14

Screenshot of Camera Activity

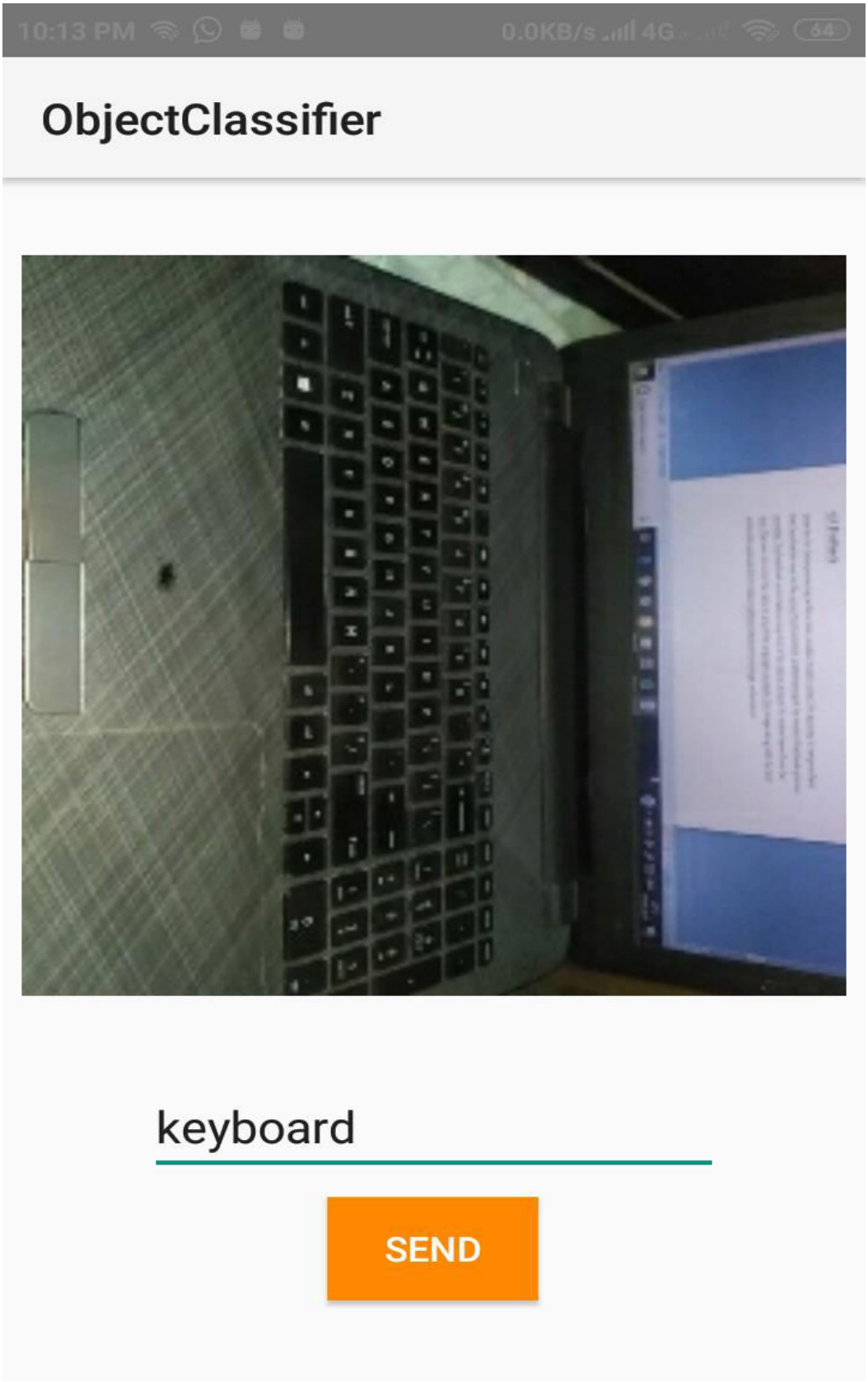

Fig-15

Screenshot of Feedback Activity

Chapter 5

Conclusion

Through this project we explored the field of computer vision and machine learning. This Application is an attempt at developing an object detection, classification and language translation system using modern computer vision technology and artificial intelligence. This application detects and classifies objects in any speaking language using Tensorflow Inference Engine and the Concept of Transfer Learning.

We used mobilenets image classification model for retraining with more more than 30 categories of objects. We have at present provided translation for only 3 languages namely English,odia and Hindi.

The further steps towards the development of the application will include:-

- Adding more languages
- Adding more classes of objects
- Word pronunciation feature for each language.

The application is uploaded on the Google Play store for use by the General People. The link to the application downloaded is given below.

https://play.google.com/store/apps/details?id=com.anshumansekhar.objectclassification

REFERENCES

1. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications by Andrew G. Howard, Menglong Zhu
2. Research on Computer Vision-Based Object Detection and Classification by-Juan Wu, Bo Peng , Zhenxiang Huang , Jietao Xie
3. Computer Vision Technology for Food Quality Evaluation by-Chengjin Du. Du, Hongju HeDa-Wen Sun
4. Transfer Learning by Lisa Torrey and Jude Shavlik
5. S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. Machine Learning
6. ImageNet Classification with Deep Convultional Neural Networks by Alex Krizhevsky,Ilya Sutskever,Geofferey E Hinto