**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.

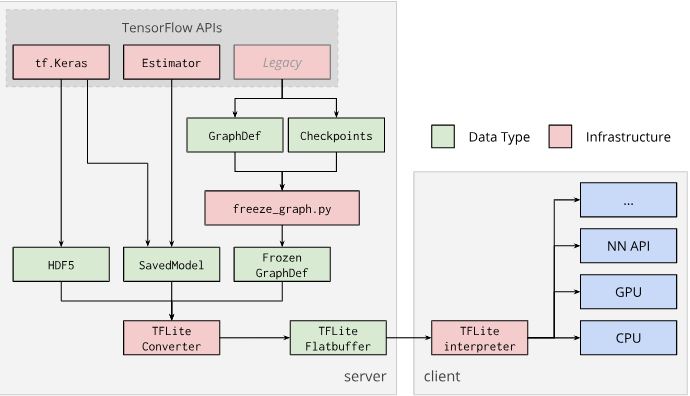


Fig-11

**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**

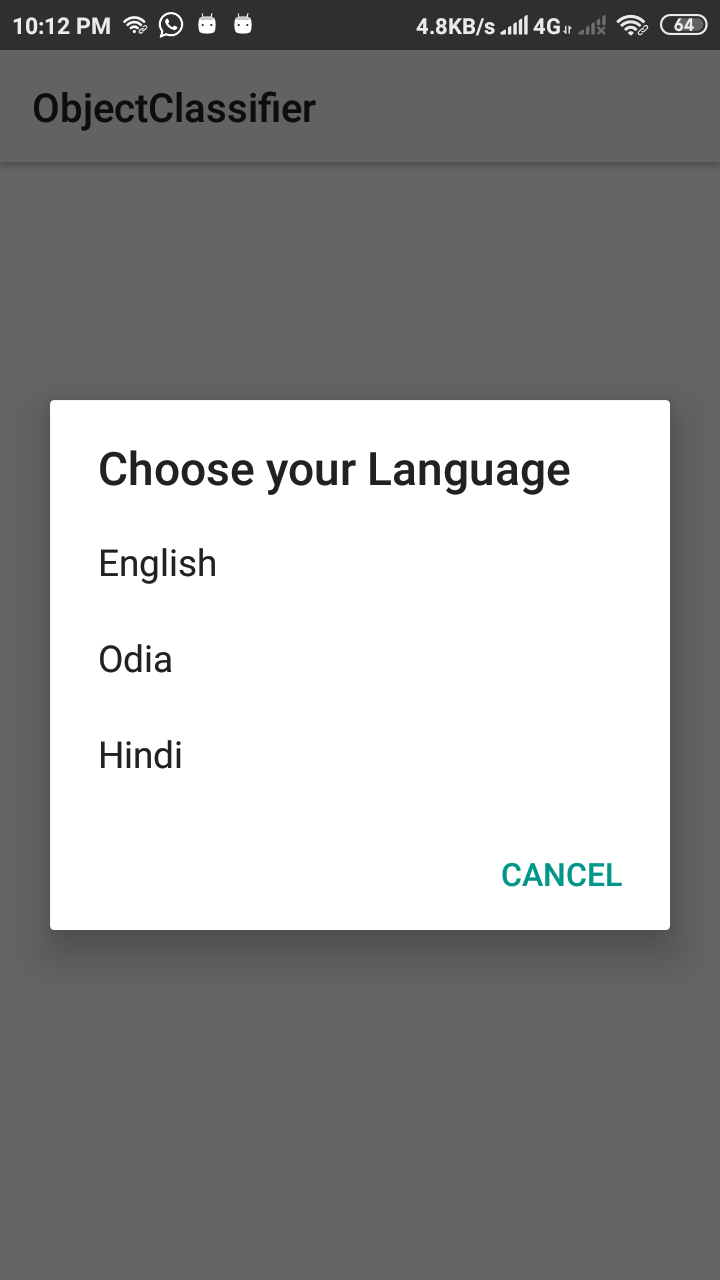


Fig-13

**Screenshot of ChooseLangugage Activity**



Fig-14

**Screenshot of Camera Activity**

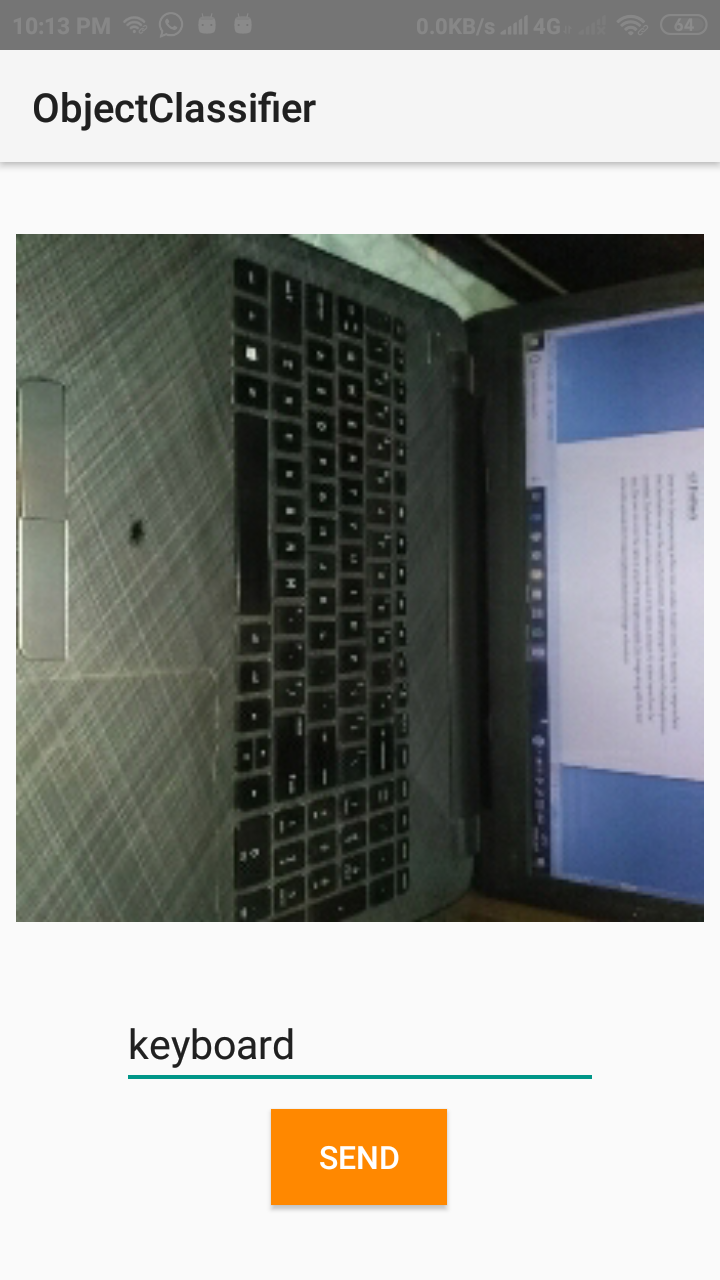


Fig-15

**Screenshot of Feedback Activity**