Developing SSD-Object Detection Models for Android Using TensorFlow

Abstract

Mobile operating environments like smartphones can benefit from on-device inference for machine learning tasks. It is common for mobile devices to use machine learning models hosted on the cloud. This approach creates latency and service availability problems, in addition to cloud service costs. With Tensorflow Lite, it becomes possible to do such inference tasks on the mobile device itself. Model training is done on high performance computing systems and the model is then converted and imported to run on Tensorflow Lite installed on the mobile.

This work demonstrates a method to train convolutional neural network (CNN) based multiclass object detection classifiers and then import the model to an Android device. In this study, TensorFlow Lite is

used to process images of cars and identify its parts on an Android mobile phone. This technique can be applied to a camera video stream in real-time, providing a kind of augmented reality (AR) experience.

Contents

Introduction

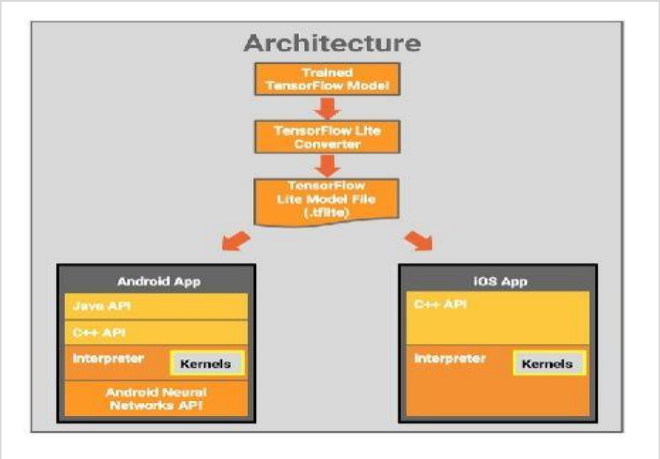
Tensorflow Lite, the next evolution of TensorFlow Mobile promises better performance to leverage hardware acceleration on supported devices. It also has few dependencies, resulting in smaller binaries than its predecessor. TensorFlow Lite is TensorFlow's lightweight solution for mobile and embedded devices. It enables on-device machine learning **inference** with low latency and a small binary size. TensorFlow Lite supports hardware acceleration with the Android Neural Networks API.

Architecture Overview of TensorFlow Lite

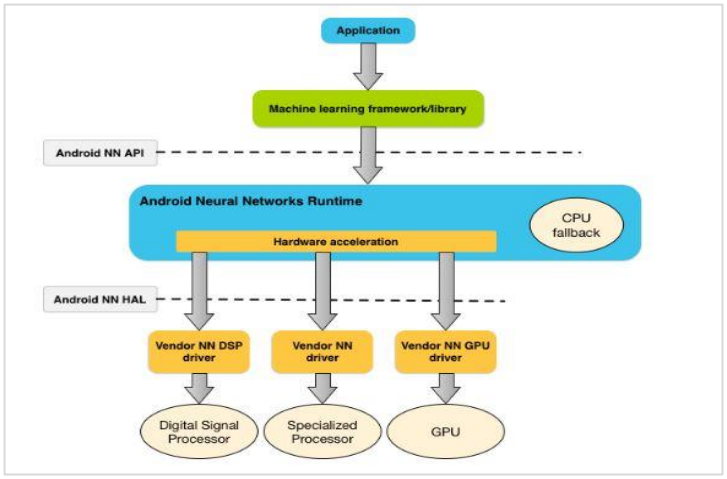
TensorFlow Lite supports both Android and iOS platforms. The initial step involves conversion of a trained TensorFlow model to TensorFlow Lite file format (**.tflite**) using the TensorFlow Lite Converter. This converted model file is used in the application.

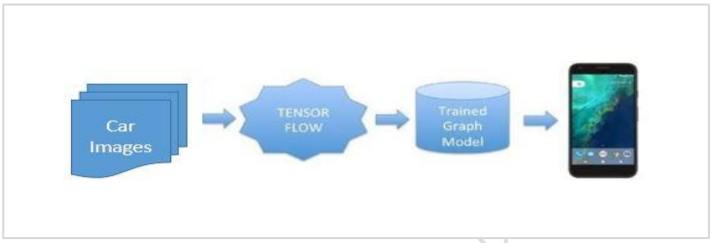
Following are the important components for deploying the model as shown in the architecture diagram:

1. **Java API:** A wrapper around the C++ API (for Android).
2. **C++ API:** The C++ API is responsible for loading the model file and invoking the interpreter for further processing and execution. The same library is used for Android and iOS.
3. **Interpreter:** Interpreter executes the model using the defined kernels. The interpreter supports selective kernel loading and developers can also define their own custom kernels that can be used to perform the execution.
4. On few Android devices, the Interpreter uses the Android Neural Networks API for hardware acceleration or default to using a CPU.



Generic Process Flow





Getting Started with TensorFlow & TensorFlow Lite

Components Required

The components required for getting started with TensorFlow are as follows:

- Android Studio IDE
- Bazel (Linux OS)
- Anaconda
- GPU
- LabelImg Labelling Tool
- Image Augmentation Tool

Train SDD MobileNet v1

Transfer learning is a machine learning method, where a model developed for a task is reused as the starting point for a model on a second task.

A pre-trained model is used for transfer learning to learn new objects. The benefit of transfer learning is that training can be much quicker and the required data is much less. In this example, the SSD MobileNet pre-trained model (on COCO) is used to train labeled car parts, like front and back doors, bumper, windshield, left and right headlights, grille, and so on. This training is done using vanilla TensorFlow on a machine with a GPU. The model is then exported to Android running TensorFlow Lite.

1. Installing TensorFlow-GPU 1.5

- Type the following command in anaconda prompt to install Tensorflow GPU.
  `C:\Users\Tensorflow> pip install tensorflow-gpu`

2. Setting Up TensorFlow Directory

The TensorFlow Object Detection API requires a specific directory structure as in its GitHub repository. It also requires several additional Python packages specific to the environmental variables (PATH and PYTHONPATH variables) and few commands to run or train an object detection model.

Following are the steps to set up TensorFlow Directory:

1. Create a folder in **C: drive** with name **tensorflow1**. This working directory contains the full TensorFlow object detection framework including training images, training data, trained classifier, and configuration files needed for the object detection classifier.

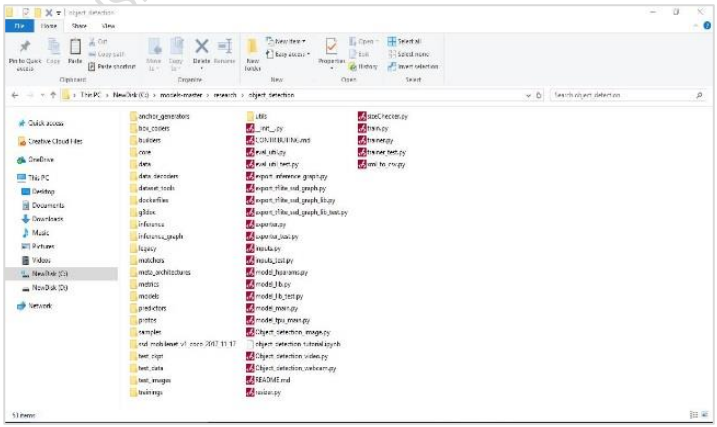
2. Click **Clone or Download** button for downloading the full TensorFlow object detection repository located at https://github.com/tensorflow/models

3. Extract the downloaded file **models-master** folder into the created local directory **C:\tensorflow1**. Rename models-master folder to **models**.

4. Click **Clone or Download** button to download the **SSD_MOBILENET_V1_COCO** model from TensorFlow's model **zoo** and extract it in the **\object_detection** folder, which is used for entire tutorial.

   https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

   \object_detection folder appears as shown in the image below:



   This repository contains the images, annotation data, .csv files, and TFRecords needed to train a car parts detector. This folder also contains Python scripts that are used to generate the training data. It has scripts to test out the object detection classifier on images, videos, or a webcam feed.

3. Setting Up the Anaconda Virtual Environment

Following are the steps required to set up Anaconda virtual environment:

1. Search for the **Anaconda** Prompt utility from the **Start** menu in Windows, right-click on it, and click **Run as Administrator**

2. Create a new virtual environment called **tensorflow1** by typing the following command in the command terminal
   ```
   C:\> conda create -n tensorflow1 pip python=3.5
   ```

3. Activate the environment by typing the following command
   ```
   C:\> activate tensorflow1
   ```

4. Install tensorflow-gpu in this environment by typing the following command:
   ```
   (tensorflow1) C:\> pip install --ignore-installed --upgrade tensorflow-gpu
   ```

5. Install the other necessary packages by typing the following commands:
   ```
   (tensorflow1) C:\> conda install -c anaconda protobuf
   (tensorflow1) C:\> pip install pillow
   (tensorflow1) C:\> pip install lxml
   (tensorflow1) C:\> pip install Cython
   (tensorflow1) C:\> pip install jupyter
   (tensorflow1) C:\> pip install matplotlib
   (tensorflow1) C:\> pip install pandas
   (tensorflow1) C:\> pip install opencv-python
   ```

   The python packages **pandas** and **opencv** are not required by TensorFlow, but they are used in the Python scripts to generate **TFRecords** for working with images, videos, and webcam feeds.

6. Configure **PYTHONPATH** environment variable by typing the following commands
   ```
   (tensorflow1) C:\> set PYTHONPATH=C:\tensorflow1\models;
   C:\tensorflow1\models\research; C:\tensorflow1\models\research\slim
   ```

7. Compile **Protobufs** and run **setup.py**

8. Change directories In the Anaconda Command Prompt to the **\models\research** directory, copy and paste the following command into the command line and press **Enter**:
   ```
   protoc --python_out=. .\object_detection\protos\anchor_generator.proto
   .\object_detection\protos\argmax_matcher.proto
   .\object_detection\protos\bipartite_matcher.proto
   .\object_detection\protos\box_coder.proto
   .\object_detection\protos\box_predictor.proto
   .\object_detection\protos\eval.proto
   .\object_detection\protos\faster_rcnn.proto
   .\object_detection\protos\faster_rcnn_box_coder.proto
   .\object_detection\protos\grid_anchor_generator.proto
   .\object_detection\protos\hyperparams.proto
   .\object_detection\protos\image_resizer.proto
   .\object_detection\protos\input_reader.proto
   .\object_detection\protos\losses.proto
   ```

```
.\object_detection\protos\matcher.proto
.\object_detection\protos\mean_stddev_box_coder.proto
.\object_detection\protos\model.proto
.\object_detection\protos\optimizer.proto
.\object_detection\protos\pipeline.proto
.\object_detection\protos\post_processing.proto
.\object_detection\protos\preprocessor.proto
.\object_detection\protos\region_similarity_calculator.proto
.\object_detection\protos\square_box_coder.proto
.\object_detection\protos\ssd.proto
.\object_detection\protos\ssd_anchor_generator.proto
.\object_detection\protos\string_int_label_map.proto
.\object_detection\protos\train.proto
.\object_detection\protos\keypoint_box_coder.proto
.\object_detection\protos\multiscale_anchor_generator.proto
.\object_detection\protos\graph_rewriter.proto
```

This creates a **name_pb2.py** file from every **name.proto** file in the **\object_detection\protos** folder.

9. Run the following commands from the **C:\tensorflow1\models\research** directory:

```
(tensorflow1)C:\tensorflow1\models\research> python setup.py build
(tensorflow1) C:\tensorflow1\models\research> python setup.py install
```

4. Gathering and Labeling Images

TensorFlow requires hundreds of images of an object to train a good detection classifier. To train a robust classifier, the training images must have random objects in the image along with the desired objects, variety of backgrounds, and lighting conditions. The images may include the partially obscured images, overlapped images, or only halfway in the picture. Following are the steps to gather and label pictures:

1. Take pictures of the objects from mobile phone or download images of the objects from Google Image Search. Recommended number of images **per class** = 300 to 500

2. Ensure the images are not too large. The larger the images are, the longer it takes time to train the classifier. The **resizer.py** script in this repository is used to reduce the size of the images.

3. Move 20% of images to the **\object_detection\images\test** directory, and 80% of images to the **\object_detection\images\train** directory.

4. Ensure there are variety of pictures in both the **\test** and **\train** directories.

5. **LabelImg** is a great tool for labeling images, and its GitHub page has very clear instructions on how to install and use it.

6. Generate augmented images, use github code below or any other augmentation tools.
   https://github.com/codebox/image_augmentor

   https://github.com/tzutalin/labelImg

   https://www.dropbox.com/s/tq7zfrcwl44vxan/windows_v1.6.0.zip?dl=1

7. Download and install **LabelImg**, point it to your **\images\train** directory, and then draw a box around each object in each image. In this case we use car parts as labels for SSD. Repeat the process for all the images in the **\images\test** directory.

8. **LabelImg** saves an **.xml** file containing the label data for each image. These **.xml** files are used to generate **TFRecords**, which are one of the inputs to the **TensorFlow** trainer. Each labeled and saved image consists of an .xml file in the **\test** and **\train** directories.

9. Check if the size of each bounding box is correct by running **sizeChecker.py**
   `(tensorflow1) C:\tensorflow1\models\research\object_detection> python sizeChecker.py --move`

   Sample Snapshot is displayed as shown below.

5. Generating Training Data

Following are the steps to generate training data:

1. With the images labeled, generate the TFRecords that serve as input data to the TensorFlow training model. The image .xml data is used to create **.csv** files containing all the data for the train and test images.

2. From the **\object_detection** folder, type the following command in the Anaconda command prompt:
   `(tensorflow1) C:\tensorflow1\models\research\object_detection> python xml_to_csv.py`

3. This creates a **train_labels.csv** and **test_labels.csv** file in the **\object_detection\images** folder.

4. Open the **generate_tfrecord.py** file in a text editor. Replace the label map starting at **line 31** with your own label map, where each object is assigned an ID number. This same number assignment is used, when configuring the **labelmap.pbtxt**


```
flags = tf.app.flags
flags.DEFINE_string('csv_input', '', 'Path to the CSV input')
flags.DEFINE_string('image_dir', '', 'Path to the image directory')
flags.DEFINE_string('output_path', '', 'Path to output TFRecord')
FLAGS = flags.FLAGS


# TO-DO replace this with label map
def class_text_to_int(row_label):
    if row_label == 'Bonnet':
        return 1
    elif row_label == 'Grille':
        return 2
    elif row_label == 'Front_Bumper':
        return 3
    elif row_label == 'Rear_Bumper':
        return 4
    elif row_label == 'Dickey':
        return 5
    elif row_label == 'Right_QP':
        return 6
    elif row_label == 'Left_Front_Door':
        return 7
    elif row_label == 'Left_Fender':
        return 8
    elif row_label == 'Left_QP':
        return 9
    elif row_label == 'Rear_Bumper':
        return 10
    elif row_label == 'Right_Rear_Door':
        return 11
    elif row_label == 'Right_Front_Door':
        return 12
    elif row_label == 'Right_Fender':
        return 13
    else:
        return 0
```

5. Generate the TFRecord files by entering these commands from the **\object_detection** folder:
   python generate_tfrecord.py --csv_input=images\train_labels.csv --image_dir=images\train --output_path=train.record

   python generate_tfrecord.py --csv_input=images\test_labels.csv --image_dir=images\test --output_path=test.record

6. Creating Label Map and Configuring Training

Following are the steps to create label map and configure training:

1. Create a label map and edit the training configuration file. The label map interprets the trainer about each object by defining a mapping of class names to class ID numbers.

2. Use a text editor to create a new file and save it as **labelmap.pbtxt** in the **C:\tensorflow1\models\research\object_detection\training** folder.

> **Note:** *Ensure the file type is **.pbtxt,** not **.txt***

3. In the text editor, copy/type in the label map in the format below:

```
item {
  id: 1
  name: 'Bonnet'
}
item {
  id: 2
  name: 'Grille'
}
item {
  id: 3
  name: 'Front_Bumper'
}
item {
```

```
  id: 4
  name: 'Rear_Bumper'
}
item {
  id: 5
  name: 'Right_QP'
}
item {
  id: 6
  name: 'Left_Front_Door'
}
Item{
  Id:7
  name:'Left_Fender'} ……..
```

The label map ID numbers remains the same as what is defined in the **generate_tfrecord.py** file.

Finally, the object detection training pipeline is configured. It defines which model and what parameters are used for training. This is the last step before running training.

4. Navigate to **C:\tensorflow1\models\research\object_detection\samples\configs** and copy the **ssd_mobilenet_v1_coco.config** file into the **\object_detection\training** directory. Then, open the **.config** file with a text editor. Change the number of classes and examples. Add the file paths to the training data.

5. Make the following changes to the **ssd_mobilenet_v1_coco.config** file and save it.

> **Note:** *Enter the paths with single forward slashes (NOT BACKSLASHES) otherwise, TensorFlow prompts a file path error, when trying to train the model. Also, the paths must be provided in double quotation marks (“) and not single quotation marks (‘).*

- Line 9. Change **num_classes** to the number of different objects you want the classifier to detect. For the above Car parts Detector, it would be **num_classes: 30.**
- Line 110. Change **fine_tune_checkpoint** to:
  - fine_tune_checkpoint : **"C:/tensorflow1/models/research/object_detection/ssd_mobilenet_v1_coco_2017_11_17/model.ckpt"**
- Lines 126 and 128. In the **train_input_reader** section, change **input_path** and **label_map_path** to:
  - input_path : **"C:/tensorflow1/models/research/object_detection/train.record"**
  - label_map_path: **"C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"**
- Line 132. Change **num_examples** to the number of images in the **\images\test** directory.
- Lines 140 and 142. In the **eval_input_reader** section, change **input_path** and **label_map_path** to:
  - input_path: "**C:/tensorflow1/models/research/object_detection/test.record**"

- label_map_path:
  "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"
- Change the batchsize to 1.

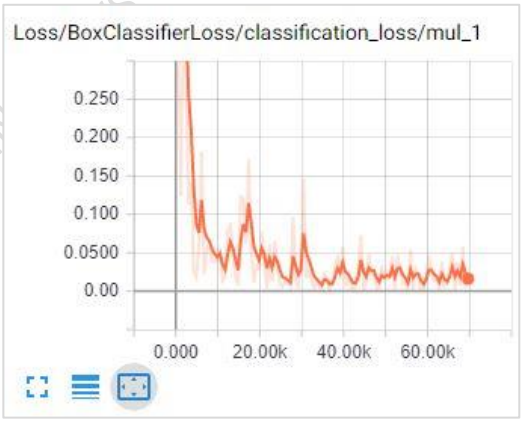
7. Running the Training

Following are the steps to run the training:

1. After setting up, initialize the training for TensorFlow. The initialization can take up to 30 seconds before the actual training begins.
   from Research/object_detection folder run
   python train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/ssd_mobilenet_v1_coco.config

2. Each step of training reports the loss. It starts high and gets low as training progresses. Allow your model to train until the loss consistently drops below 0.05, which takes around 40,000 steps, or about 2 hours (depends on power of CPU/GPU).
3. Progress of the training job is viewed by using TensorBoard. Open a new instance of Anaconda Prompt, activate the **tensorflow1** virtual environment, change to the **C:\tensorflow1\models\research\object_detection** directory, and type the following command:
   (tensorflow1)
   C:\tensorflow1\models\research\object_detection>tensorboard --logdir=training

   This creates a webpage on your local machine at **YourPCName:6006**, which is viewed through a web browser. The TensorBoard page provides information and graphs that show about the progress of the training. One important graph is the **Loss graph**, which shows the overall loss of the classifier over time.



8. Exporting The Inference Graph

Following are the steps to export the inference graph:

1. The last step is to generate the frozen inference graph (**.pb** file). From the **\object_detection** folder, type the following command, where "**XXXX**" in "**model.ckpt-XXXX**" must be replaced with the highest-numbered **.ckpt** file in the training folder:

   From research/object_detection folder run

```
python export_inference_graph.py --input_type image_tensor --pipeline_config_path training/faster_rcnn_inception_v2_pets.config --trained_checkpoint_prefix training/model.ckpt-XXXX --output_directory inference_graph
```

2. This creates a **frozen_inference_graph.pb** file in the **\object_detection\inference_graph** folder. The **.pb** file contains the object detection classifier.

Bazel Installation (LINUX)

Bazel is an open-source build and test tool similar to Make, Maven, and Gradle. It uses a human-readable, high-level build language. Bazel supports projects in multiple languages and builds outputs for multiple platforms. Bazel supports large codebases across multiple repositories, and large numbers of users. Following are the steps for installing Bazel:

1. Download the **Bazel** binary installer named **bazel-<version>-installer-linux-x86_64.sh** from the **Bazel** releases page on **GitHub**.
   ```
   sudo apt-get install pkg-config zip g++ zlib1g-dev unzip python
   ```
2. Run the Installer.
   ```
   chmod +x bazel-<version>-installer-linux-x86_64.sh
   ./bazel-<version>-installer-linux-x86_64.sh –user
   sudo apt-get update && sudo apt-get install bazel
   sudo add-apt-repository ppa:webupd8team/java
   sudo apt-get update && sudo apt-get install oracle-java8-installer
   ```
3. Setup your Environment
   ```
   export PATH="$PATH:$HOME/bin"
   ```
4. Install JDK
   ```
   sudo apt-get install openjdk-8-jdk
   ```
5. Edit **WORKSPACE** File. In the root of the TensorFlow repository, update the **WORKSPACE** file with the **api_level** and **location** of the SDK and NDK.
   If you installed it with Android Studio, the SDK path can be found in the SDK manager. The default NDK path is:**{SDK path}/ndk-bundle**.

   For example:

   ```
   android_sdk_repository (
       name = "androidsdk",
       api_level = 23,
       build_tools_version = "23.0.2",
       path = "/home/xxxx/android-sdk-linux/",
   )

   android_ndk_repository(
       name = "androidndk",
       path = "/home/xxxx/android-ndk-r10e/",
       api_level = 19,
   )
   ```
   Reference: https://docs.bazel.build/versions/master/install-ubuntu.html

Build in Android Studio with TensorFlow Lite AAR from JCenter

Use Android Studio to make changes in the project code and compile the demo app following the instructions below:

1. Install the latest version of Android Studio.
2. Ensure the Android SDK version is greater than 26 and NDK version is greater than 14 (in the Android Studio settings).
3. Import the **tensorflow/lite/java/demo** directory as a new Android Studio project.
4. Install all the Gradle extensions it requests.
5. Build and run the demo app.
6. Load Model to the assets directory: **tensorflow/lite/java/demo/app/src/main/assets/.** Some additional details are available on the TF Lite Android App page

Deploying the model in Android

Following are the steps to deploy the model in Android:

1. Tensorflow Lite has a limited support of operations. The script "**export_tflite_ssd_graph.py**" generates a frozen graph that works on **TFLite**.

2. Convert this graph to Tensorflow Lite format using **TOCO** (**TensorFlow Lite Optimizing Converter**). The **detect.tflitegraph** is the one, which is used in the mobile app.

Export graph for tensorflow lite

```
python
/tensorflow_models/research/object_detection/export_tflite_ssd_graph.py
\
--pipeline_config_path CONFIG_FILE \
--trained_checkpoint_prefix CHECKPOINT_PATH \
--output_directory OUTPUT_DIR \
```

Optimize graph for tensorflow lite using TOCO

```
cd  /tensorflow | \
bazel run -c opt tensorflow/lite/toco:toco -- \
--input_file=$OUTPUT_DIR/tflite_graph.pb \
--output_file=$OUTPUT_DIR/carparts.tflite \
--input_shapes=1,300,300,3 \
--input_arrays=normalized_input_image_tensor \
--
output_arrays='TFLite_Detection_PostProcess','TFLite_Detection_PostProc
ess:1','TFLite_Detection_PostProcess:2','TFLite_Detection_PostProcess:3
'  \
--inference_type=FLOAT \
--mean_values=128 \
--std_values=128 \
--change_concat_input_ranges=false \
```

```
--allow_custom_ops
```

Loading the model

The FlatBufferModel class encapsulates a model and is built in a couple of slightly different ways depending on where the model is stored:

```
class FlatBufferModel {
  // Build a model based on a file. Return a nullptr in case of
failure.
  static std::unique_ptr<FlatBufferModel> BuildFromFile(
      const char* filename,
      ErrorReporter* error_reporter);
```

Note: *If TensorFlow Lite detects the presence of Android's NNAPI, it automatically tries to use shared memory to store the FlatBufferModel*

```
private static final int TF_OD_API_INPUT_SIZE = 300;
private static final String TF_OD_API_MODEL_FILE = "carparts.tflite";
private static final String TF_OD_API_LABELS_FILE =
"file:///android_asset/car_parts_label.txt";

detector =
       TFLiteObjectDetectionAPIModel.create(
           getAssets(),
           TF_OD_API_MODEL_FILE,
           TF_OD_API_LABELS_FILE,
           TF_OD_API_INPUT_SIZE,
           TF_OD_API_IS_QUANTIZED);
cropSize = TF_OD_API_INPUT_SIZE;
```


Running the model

Following are the steps to run a model:

1. Build an Interpreter based on an existing **FlatBufferModel**
2. Optionally resize input tensors, if the predefined sizes are not desired.
3. Set input tensor values
4. Invoke inference
5. Read output tensor values

Note: *Tensors are represented by integers, in order to avoid string comparisons (and any fixed dependency on string libraries). An interpreter must not be accessed from concurrent threads. Memory allocation for input and output tensors must be triggered by calling **AllocateTensors()** right after resizing tensors.*

```
class Interpreter {
```

```
Interpreter(ErrorReporter* error_reporter);
// Read only access to list of inputs.
const std::vector<int>& inputs() const;
// Read only access to list of outputs.
const std::vector<int>& outputs() const;
// Change the dimensionality of a given tensor.
TfLiteStatus ResizeInputTensor(int tensor_index,
                               const std::vector<int>& dims);
// Returns status of success or failure.
TfLiteStatus AllocateTensors();

// Return a pointer into the data of a given input tensor.
template <class T>
T* typed_input_tensor(int index) {
  return typed_tensor<T>(inputs_[index]);
}

// Return a pointer into the data of a given output tensor.
template <class T>
T* typed_output_tensor(int index) {
  return typed_tensor<T>(outputs_[index]);
}
// Execute the model, populating output tensors.
TfLiteStatus Invoke();
};
protected void processImage() {
  ++timestamp;
  final long currTimestamp = timestamp;
  byte[] originalLuminance = getLuminance();
  tracker.onFrame(
      previewWidth,
      previewHeight,
      getLuminanceStride(),
      sensorOrientation,
      originalLuminance,
      timestamp);
  trackingOverlay.postInvalidate();
  if (computingDetection) {
    readyForNextImage();
    return;
  }
  computingDetection = true;
  LOGGER.i("Preparing image " + currTimestamp + " for detection in bg
thread.");

  rgbFrameBitmap.setPixels(getRgbBytes(), 0, previewWidth, 0, 0,
previewWidth, previewHeight);
  if (luminanceCopy == null) {
    luminanceCopy = new byte[originalLuminance.length];
  }
```

```
  System.arraycopy(originalLuminance, 0, luminanceCopy, 0,
originalLuminance.length);
  readyForNextImage();
  final Canvas canvas = new Canvas(croppedBitmap);
  canvas.drawBitmap(rgbFrameBitmap, frameToCropTransform, null);
  // For examining the actual TF input.
  if (SAVE_PREVIEW_BITMAP) {
    ImageUtils.saveBitmap(croppedBitmap);
  }
  runInBackground(
      new Runnable() {
        @Override
        public void run() {
          LOGGER.i("Running detection on image " + currTimestamp);
          final long startTime = SystemClock.uptimeMillis();
          final List<Classifier.Recognition> results =
  detector.recognizeImage(croppedBitmap);
          lastProcessingTimeMs = SystemClock.uptimeMillis() - startTime;
          cropCopyBitmap = Bitmap.createBitmap(croppedBitmap);
          final Canvas canvas = new Canvas(cropCopyBitmap);
          final Paint paint = new Paint();
          paint.setColor(Color.RED);
          paint.setStyle(Style.STROKE);
          paint.setStrokeWidth(2.0f);
          float minimumConfidence = MINIMUM_CONFIDENCE_TF_OD_API;
          switch (MODE) {
            case TF_OD_API:
              minimumConfidence = MINIMUM_CONFIDENCE_TF_OD_API;
              break;
          }
          final List<Classifier.Recognition> mappedRecognitions =
              new LinkedList<Classifier.Recognition>();
          for (final Classifier.Recognition result : results) {
            final RectF location = result.getLocation();
            if (location != null && result.getConfidence() >=
          minimumConfidence) {
              canvas.drawRect(location, paint);

              cropToFrameTransform.mapRect(location);
              result.setLocation(location);
              mappedRecognitions.add(result);
            }
          }
          tracker.trackResults(mappedRecognitions, luminanceCopy,
          currTimestamp);
          trackingOverlay.postInvalidate();
          requestRender();
          computingDetection = false;
        }
      });
```

```
}
```

Object Detection API

TensorFlow Lite is designed for fast inference on small devices, which provides the APIs to avoid unnecessary copies at the expense of convenience.
Following are the sample code snippets for object detection API:

```
final TFLiteObjectDetectionAPIModel d = new
  TFLiteObjectDetectionAPIModel();

  InputStream labelsInput = null;
String actualFilename = labelFilename.split("file:///android_asset/")[1];
labelsInput = assetManager.open(actualFilename);
BufferedReader br = null;
br = new BufferedReader(new InputStreamReader(labelsInput));
String line;
while ((line = br.readLine()) != null) {
  LOGGER.w(line);
  d.labels.add(line);
}
br.close();

d.inputSize = inputSize;

try {
  d.tfLite = new Interpreter(loadModelFile(assetManager, modelFilename));
} catch (Exception e) {
  throw new RuntimeException(e);
}

d.isModelQuantized = isQuantized;
// Pre-allocate buffers.
int numBytesPerChannel;
if (isQuantized) {
  numBytesPerChannel = 1; // Quantized
} else {
  numBytesPerChannel = 4; // Floating point
}
d.imgData = ByteBuffer.allocateDirect(1 * d.inputSize * d.inputSize * 3 *
  numBytesPerChannel);
d.imgData.order(ByteOrder.nativeOrder());
d.intValues = new int[d.inputSize * d.inputSize];

d.tfLite.setNumThreads(NUM_THREADS);
d.outputLocations = new float[1][NUM_DETECTIONS][4];
d.outputClasses = new float[1][NUM_DETECTIONS];
d.outputScores = new float[1][NUM_DETECTIONS];
d.numDetections = new float[1];
return d;
```

```
  }

  Trace.beginSection("run");
tfLite.runForMultipleInputsOutputs(inputArray, outputMap);
Trace.endSection();

final ArrayList<Recognition> recognitions = new
  ArrayList<>(NUM_DETECTIONS);
for (int i = 0; i < NUM_DETECTIONS; ++i) {
  final RectF detection =
      new RectF(
          outputLocations[0][i][1] * inputSize,
          outputLocations[0][i][0] * inputSize,
          outputLocations[0][i][3] * inputSize,
          outputLocations[0][i][2] * inputSize);

  int labelOffset = 1;
  recognitions.add(
      new Recognition(
          "" + i,
          labels.get((int) outputClasses[0][i] + labelOffset),
          outputScores[0][i],
          detection));
}
```


Compile Android app

1. Compile the Android app using Bazel, it uses optimized native scripts in C++ to track the objects.

```
android_sdk_repository (
     name = "androidsdk",
     api_level = 23,
     build_tools_version = "23.0.2",
     path = "/home/xxxx/android-sdk-linux/",
   )

android_ndk_repository(
     name = "androidndk",
     path = "/home/xxxx/android-ndk-r10e/",
     api_level = 19,
   )
```

2. Set path in the workspace for the PATH_TO_SDK and PATH_TO_NDK.

```
bazel build -c opt --cxxopt='--std=c++11'
//tensorflow/tflite/android:tflite_demo
```

Install the app

The following code installs the app

adb install -r -f bazel-bin/tensorflow/tflite/android/tflite_demo.apk

Sample Output

The following are the output images showing car image with bounding boxes over individual car parts with their labels.

Advantages and Applications

TensorFlow Lite remains better in its usage and applications due to the following characteristics:

- TensorFlow Lite enables on-device machine learning inference with low latency. These characteristics led TensorFlow as fast in response with reliable operations.
- TensorFlow Lite occupies small binary size and remains suitable for mobile devices.
- TensorFlow Lite supports hardware acceleration with the Android Neural Networks API.
- TensorFlow Lite operates extensively without relying upon the internet connectivity.
- TensorFlow Lite also enriches developers toward the exploration of pioneering real time applications.

TensorFlow Lite uses several techniques for achieving low latency such as:

- Optimizing the kernels for mobile apps.
- Pre-fused activations.
- Quantized kernels that allow smaller and faster models.

Conclusion

In this study, TensorFlow model is deployed on mobile devices, which addresses many challenges during the deployment. This document fulfils the concepts detailing about the overview of TensorFlow, its architecture, its process flow, step-by-step procedures to start, train the model, and deployment. This document serves as a technical reference document for developers and provides an approach for deployment of TensorFlow Lite on Android.

References

https://github.com/tensorflow/models

https://www.tensorflow.org/lite/demo_android

https://medium.com/mindorks/android-tensorflow-lite-machine-learning-example-b06ca29226b6

https://riggaroo.co.za/building-a-custom-machine-learning-model-on-android-with-tensorflow-lite/

https://blog.mindorks.com/android-tensorflow-machine-learning-example-ff0e9b2654cc

https://github.com/EdjeElectronics/TensorFlow-Object-Detection-API-Tutorial-Train-Multiple-Objects-Windows-10

https://medium.com/@rdeep/tensorflow-lite-tutorial-easy-implementation-in-android-145443ec3775

https://heartbeat.fritz.ai/intro-to-machine-learning-on-android-how-to-convert-a-custom-model-to-tensorflow-lite-e07d2d9d50e3

https://medium.com/tensorflow/training-and-serving-a-realtime-mobile-object-detector-in-30-minutes-with-cloud-tpus-b78971cf1193