Running TF2 Detection API Models on mobile

TensorFlow 2.4 Python 3.6

NOTE: This document talks about the *SSD* models in the detection zoo. For details on our (experimental) CenterNet support, see this notebook.

TensorFlow Lite(TFLite) 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 uses many techniques for this such as quantized kernels that allow smaller and faster (fixed-point math) models.

This document shows how eligible models from the TF2 Detection zoo can be converted for inference with TFLite. See this Colab tutorial for a runnable tutorial that walks you through the steps explained in this document:

Run in Google Colab

For an end-to-end Python guide on how to fine-tune an SSD model for mobile inference, look at this Colab.

NOTE: TFLite currently only supports **SSD Architectures** (excluding EfficientDet) for boxes-based detection. Support for EfficientDet is provided via the TFLite Model Maker library.

The output model has the following inputs & outputs:

```
One input:
  image: a float32 tensor of shape[1, height, width, 3] containing the
  *normalized* input image.
  NOTE: See the `preprocess` function defined in the feature extractor class
  in the object_detection/models directory.

Four Outputs:
  detection_boxes: a float32 tensor of shape [1, num_boxes, 4] with box
  locations
  detection_classes: a float32 tensor of shape [1, num_boxes]
  with class indices
  detection_scores: a float32 tensor of shape [1, num_boxes]
  with class scores
  num_boxes: a float32 tensor of size 1 containing the number of detected boxes
```

There are two steps to TFLite conversion:

Step 1: Export TFLite inference graph

This step generates an intermediate SavedModel that can be used with the TFLite Converter via commandline or Python API.

To use the script:

```
# From the tensorflow/models/research/ directory
python object_detection/export_tflite_graph_tf2.py \
    --pipeline_config_path path/to/ssd_model/pipeline.config \
    --trained_checkpoint_dir path/to/ssd_model/checkpoint \
    --output_directory path/to/exported_model_directory
```

Use `--help` with the above script to get the full list of supported parameters. These can fine-tune accuracy and speed for your model.

Step 2: Convert to TFLite

Use the TensorFlow Lite Converter to convert the `SavedModel` to TFLite. Note that you need to use `from_saved_model` for TFLite conversion with the Python API.

You can also leverage Post-training Quantization to optimize performance and obtain a smaller model. Note that this is only possible from the *Python API*. Be sure to use a representative dataset and set the following options on the converter:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8,
                                       tf.lite.OpsSet.TFLITE_BUILTINS]
converter.representative_dataset = <...>
```

Step 3: add Metadata to the model

To make it easier to use tflite models on mobile, you will need to add metadata to your model and also pack the associated labels file to it. If you need more information, This process is also explained in the Image classification sample

Running our model on Android

Integrate the model into your app

You can use the TFLite Task Library's ObjectDetector API to integrate the model into your Android app.

```
// Initialization
ObjectDetectorOptions options =
ObjectDetectorOptions.builder().setMaxResults(1).build();
ObjectDetector objectDetector = ObjectDetector.createFromFileAndOptions(context,
modelFile, options);

// Run inference
List<Detection> results = objectDetector.detect(image);
```

Test the model using the TFLite sample app

To test our TensorFlow Lite model on device, we will use Android Studio to build and run the TensorFlow Lite detection example with the new model. The example is found in the TensorFlow examples repository under `/lite/examples/object_detection`. The example can be built with Android Studio, and requires the Android SDK with build tools that support API >= 21. Additional details are available on the TensorFlow Lite example page.

Next we need to point the app to our new detect.tflite file . Specifically, we will copy our TensorFlow Lite flatbuffer to the app assets directory with the following command:

```
mkdir $TF_EXAMPLES/lite/examples/object_detection/android/app/src/main/assets
cp /tmp/tflite/detect.tflite \
  $TF_EXAMPLES/lite/examples/object_detection/android/app/src/main/assets
```

It's important to notice that the labels file should be packed in the model (as mentioned on Step 3)

We will now edit the gradle build file to use these assets. First, open the `build.gradle` file `$TF_EXAMPLES/lite/examples/object_detection/android/app/build.gradle`. Comment out the model download script to avoid your assets being overwritten: `// apply from:'download_model.gradle'` ```

If your model is named `detect.tflite`, the example will use it automatically as long as they've been properly copied into the base assets directory. If you need to use a custom path or filename, open up the $TF_EXAMPLES/lite/examples/object_detection/android/app/src/main/java/org/tensorflow/demo/DetectorActivity.java file in a text editor and find the definition of TF_OD_API_MODEL_FILE. Note that if your model is quantized, the flag TF_OD_API_IS_QUANTIZED is set to true, and if your model is floating point, the flag TF_OD_API_IS_QUANTIZED is set to false. This new section of DetectorActivity.java should now look as follows for a quantized model:

```
  private static final boolean TF_OD_API_IS_QUANTIZED = true;
  private static final String TF_OD_API_MODEL_FILE = "detect.tflite";
  private static final String TF_OD_API_LABELS_FILE = "labels_list.txt";
```

Once you've copied the TensorFlow Lite model and edited the gradle build script to not use the downloaded assets, you can build and deploy the app using the usual Android Studio build process.