Post-training quantization

Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. You can perform these techniques using an already-trained float TensorFlow model when you convert it to TensorFlow Lite format.

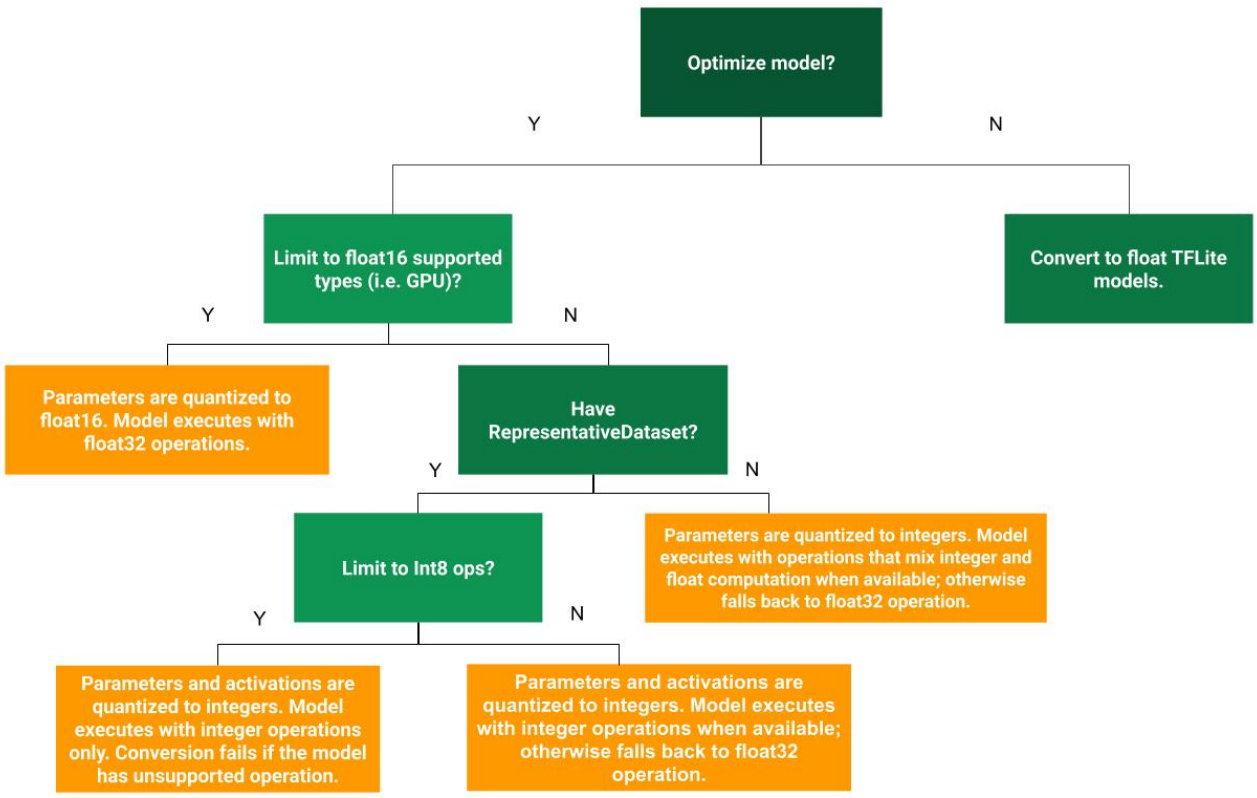
> **Note:** The procedures on this page require TensorFlow 1.15 or higher.

Optimization options

There are several post-training quantization options to choose from. Here is a summary table of the choices and the benefits they provide:

Technique	Benefits	Hardware
Weight quantization	4x smaller, 2-3x speedup, accuracy	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, etc.
Float16 quantization	2x smaller, potential GPU acceleration	CPU/GPU

This decision tree can help determine which post-training quantization method is best for your use case:



Alternatively, you might achieve higher accuracy if you perform quantization-aware training. However, doing so requires some model modifications to add fake quantization nodes, whereas the post-training quantization techniques on this page use an existing pre-trained model.

Weight quantization

The simplest form of post-training quantization quantizes only the weights from floating point to 8-bits of precision (also called "hybrid" quantization). This technique is enabled as an option in the TensorFlow Lite converter:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_quant_model = converter.convert()
```

At inference, weights are converted from 8-bits of precision to floating point and computed using floating-point kernels. This conversion is done once and cached to reduce latency.

To further improve latency, hybrid operators dynamically quantize activations to 8-bits and perform computations with 8-bit weights and activations. This optimization provides latencies close to fully fixed-point inference. However, the outputs are still stored using floating point, so that the speedup with hybrid ops is less than a full fixed-point computation. Hybrid ops are available for the most compute-intensive operators in a network:

- tf.contrib.layers.fully_connected
- tf.nn.conv2d
- tf.nn.embedding_lookup
- BasicRNN
- tf.nn.bidirectional_dynamic_rnn for BasicRNNCell type
- tf.nn.dynamic_rnn for LSTM and BasicRNN Cell types

Full integer quantization of weights and activations

You can get further latency improvements, reductions in peak memory usage, and access to integer only hardware accelerators by making sure all model math is quantized.

To do this, you need to measure the dynamic range of activations and inputs by supplying a representative data set. You can simply create an input data generator and provide it to our converter. For example:

```
import tensorflow as tf

def representative_dataset_gen():
  for _ in range(num_calibration_steps):
    # Get sample input data as a numpy array in a method of your choosing.
    yield [input]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen
tflite_quant_model = converter.convert()
```

The resulting model should be fully quantized, but any ops that do not have quantized implementations are left in floating point. This allows conversion to occur smoothly, but the model won't be compatible with accelerators that require full integer quantization.

Additionally, the model still uses float input and output for convenience.

To ensure compatibility with some accelerators (such as the Coral Edge TPU), you can enforce full integer quantization for all ops and use integer input and output by adding the following lines before you convert:

```
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8
```

The first line makes the converter throw an error if it encounters an operation it cannot currently quantize.

> **Note:** `target_spec.supported_ops` was previously `target_ops` in the Python API.

Float16 quantization of weights

You can reduce the size of a floating point model by quantizing the weights to float16, the IEEE standard for 16-bit floating point numbers. The advantages of this quantization are as follows:

- reduce model size by up to half (since all weights are now half the original size)
- minimal loss in accuracy
- some delegates (e.g. the GPU delegate) can operate directly on float16 data, which results in faster execution than float32 computations.

This quantization may not be a good choice if you need maximum performance (a quantization to fixed point math would be better in that case). To enable float16 quantization of weights, specify "DEFAULT" optimization as above and then specify that float16 is in supported types for the target_spec:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.lite.constants.FLOAT16]
tflite_quant_model = converter.convert()
```

By default, a float16 quantized model will "dequantize" the weights values to float32 when run on the CPU. The GPU delegate will not perform this dequantization, since it can operate on float16 data.

Model accuracy

Since weights are quantized post training, there could be an accuracy loss, particularly for smaller networks. Pre-trained fully quantized models are provided for specific networks in the TensorFlow Lite model repository. It is important to check the accuracy of the quantized model to verify that any degradation in accuracy is within acceptable limits. There is a tool to evaluate TensorFlow Lite model accuracy.

If the accuracy drop is too high, consider using quantization aware training.

Representation for quantized tensors

8-bit quantization approximates floating point values using the following formula. `real_value = (int8_value - zero_point) * scale`.

The representation has two main parts:

- Per-axis (aka per-channel) or per-tensor weights represented by int8 two’s complement values in the range [-127, 127] with zero-point equal to 0.
- Per-tensor activations/inputs represented by int8 two’s complement values in the range [-128, 127], with a zero-point in range [-128, 127].

For a detailed view of our quantization scheme, please see our quantization spec. Hardware vendors who want to plug into TensorFlow Lite's delegate interface are encouraged to implement the quantization scheme described there.