Post-training integer quantization

> Indented block

View on TensorFlow.org Run in Google Colab View source on GitHub

Overview

TensorFlow Lite now supports converting all model values (weights and activations) to 8-bit intege
TensorFlow Lite's flat buffer format. This results in a 4x reduction in model size and a 3 to 4x perfo
In addition, this fully quantized model can be consumed by integer-only hardware accelerators.

In contrast to post-training "on-the-fly" quantization—which stores only the weights as 8-bit integer
weights *and* activations during model conversion.

In this tutorial, you'll train an MNIST model from scratch, check its accuracy in TensorFlow, and the
flatbuffer with full quantization. Finally, you'll check the accuracy of the converted model and com

Build an MNIST model

Setup

```
import logging
logging.getLogger("tensorflow").setLevel(logging.DEBUG)

try:
  # %tensorflow_version only exists in Colab.
  import tensorflow.compat.v2 as tf
except Exception:
  pass
tf.enable_v2_behavior()

from tensorflow import keras
import numpy as np
import pathlib
```

Train and export the model

```
# Load MNIST dataset
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the input image so that each pixel value is between 0 to 1.
train_images = train_images / 255.0
test_images = test_images / 255.0

# Define the model architecture
model = keras.Sequential([
  keras.layers.InputLayer(input_shape=(28, 28)),
  keras.layers.Reshape(target_shape=(28, 28, 1)),
  keras.layers.Conv2D(filters=12, kernel_size=(3, 3), activation=tf.nn.relu),
  keras.layers.MaxPooling2D(pool_size=(2, 2)),
  keras.layers.Flatten(),
  keras.layers.Dense(10, activation=tf.nn.softmax)
])

# Train the digit classification model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(
  train_images,
  train_labels,
  epochs=1,
  validation_data=(test_images, test_labels)
)
```

This training won't take long because you're training the model for just a single epoch, which trains

Convert to a TensorFlow Lite model

Using the Python TFLiteConverter, you can now convert the trained model into a TensorFlow Lite r

Now load the model using the `TFLiteConverter`:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
```

Write it out to a `.tflite` file:

```
tflite_models_dir = pathlib.Path("/tmp/mnist_tflite_models/")
tflite_models_dir.mkdir(exist_ok=True, parents=True)
```

```
tflite_model_file = tflite_models_dir/"mnist_model.tflite"
tflite_model_file.write_bytes(tflite_model)
```

Now you have a trained MNIST model that's converted to a `.tflite` file, but it's still using 32-bit flo

So let's convert the model again, this time using quantization...

Convert using quantization

First, first set the `optimizations` flag to optimize for size:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

Now, in order to create quantized values with an accurate dynamic range of activations, you need t

```
mnist_train, _ = tf.keras.datasets.mnist.load_data()
images = tf.cast(mnist_train[0], tf.float32) / 255.0
mnist_ds = tf.data.Dataset.from_tensor_slices((images)).batch(1)
def representative_data_gen():
  for input_value in mnist_ds.take(100):
    yield [input_value]

converter.representative_dataset = representative_data_gen
```

Finally, convert the model to TensorFlow Lite format:

```
tflite_model_quant = converter.convert()
tflite_model_quant_file = tflite_models_dir/"mnist_model_quant.tflite"
tflite_model_quant_file.write_bytes(tflite_model_quant)
```

Note how the resulting file is approximately `1/4` the size:

```
!ls -lh {tflite_models_dir}
```

Your model should now be fully quantized. However, if you convert a model that includes any oper
those ops are left in floating point. This allows for conversion to complete so you have a smaller a
be compatible with some ML accelerators that require full integer quantization. Also, by default, th
outputs, which also is not compatible with some accelerators.

So to ensure that the converted model is fully quantized (make the converter throw an error if it en
and to use integers for the model's input and output, you need to convert the model again using th

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

tflite_model_quant = converter.convert()
tflite_model_quant_file = tflite_models_dir/"mnist_model_quant_io.tflite"
tflite_model_quant_file.write_bytes(tflite_model_quant)
```

In this example, the resulting model size remains the same because all operations successfully qu
model now uses quantized input and output, making it compatible with more accelerators, such as

In the following sections, notice that we are now handling two TensorFlow Lite models: `tflite_mo` uses floating-point parameters, and `tflite_model_quant_file` is the same model converted with and output.

Run the TensorFlow Lite models

Run the TensorFlow Lite model using the Python TensorFlow Lite Interpreter.

Load the model into the interpreters

```
interpreter = tf.lite.Interpreter(model_path=str(tflite_model_file))
interpreter.allocate_tensors()
```

```
interpreter_quant = tf.lite.Interpreter(model_path=str(tflite_model_quant_file))
interpreter_quant.allocate_tensors()
input_index_quant = interpreter_quant.get_input_details()[0]["index"]
output_index_quant = interpreter_quant.get_output_details()[0]["index"]
```

Test the models on one image

First test it on the float model:

```
test_image = np.expand_dims(test_images[0], axis=0).astype(np.float32)

input_index = interpreter.get_input_details()[0]["index"]
output_index = interpreter.get_output_details()[0]["index"]
interpreter.set_tensor(input_index, test_image)
interpreter.invoke()
predictions = interpreter.get_tensor(output_index)
```

```
import matplotlib.pylab as plt

plt.imshow(test_images[0])
template = "True:{true}, predicted:{predict}"
_ = plt.title(template.format(true= str(test_labels[0]),
                              predict=str(np.argmax(predictions[0]))))
plt.grid(False)
```

Now test the quantized model (using the uint8 data):

```
input_index = interpreter_quant.get_input_details()[0]["index"]
output_index = interpreter_quant.get_output_details()[0]["index"]
interpreter_quant.set_tensor(input_index, test_image)
interpreter_quant.invoke()
predictions = interpreter_quant.get_tensor(output_index)
```

```
plt.imshow(test_images[0])
template = "True:{true}, predicted:{predict}"
_ = plt.title(template.format(true= str(test_labels[0]),
                              predict=str(np.argmax(predictions[0]))))
plt.grid(False)
```

Evaluate the models

```
# A helper function to evaluate the TF Lite model using "test" dataset.
def evaluate_model(interpreter):
  input_index = interpreter.get_input_details()[0]["index"]
  output_index = interpreter.get_output_details()[0]["index"]

  # Run predictions on every image in the "test" dataset.
  prediction_digits = []
  for test_image in test_images:
    # Pre-processing: add batch dimension and convert to float32 to match with
    # the model's input data format.
    test_image = np.expand_dims(test_image, axis=0).astype(np.float32)
    interpreter.set_tensor(input_index, test_image)

    # Run inference.
    interpreter.invoke()

    # Post-processing: remove batch dimension and find the digit with highest
    # probability.
    output = interpreter.tensor(output_index)
    digit = np.argmax(output()[0])
    prediction_digits.append(digit)

  # Compare prediction results with ground truth labels to calculate accuracy.
  accurate_count = 0
  for index in range(len(prediction_digits)):
    if prediction_digits[index] == test_labels[index]:
      accurate_count += 1
  accuracy = accurate_count * 1.0 / len(prediction_digits)

  return accuracy
```

```
print(evaluate_model(interpreter))
```

Repeat the evaluation on the fully quantized model using the uint8 data:

```
# NOTE: Colab runs on server CPUs, and TensorFlow Lite currently
# doesn't have super optimized server CPU kernels. So this part may be
# slower than the above float interpreter. But for mobile CPUs, considerable
# speedup can be observed.

print(evaluate_model(interpreter_quant))
```

In this example, you have fully quantized a model with almost no difference in the accuracy, compa