Post Training Quantization (PTQ) vs Quantization Aware Training (QAT)

DCA UFRN

Ivanovitch Silva
ivanovitch.silva@ufrn.br

Post Training Quantization (PTQ)

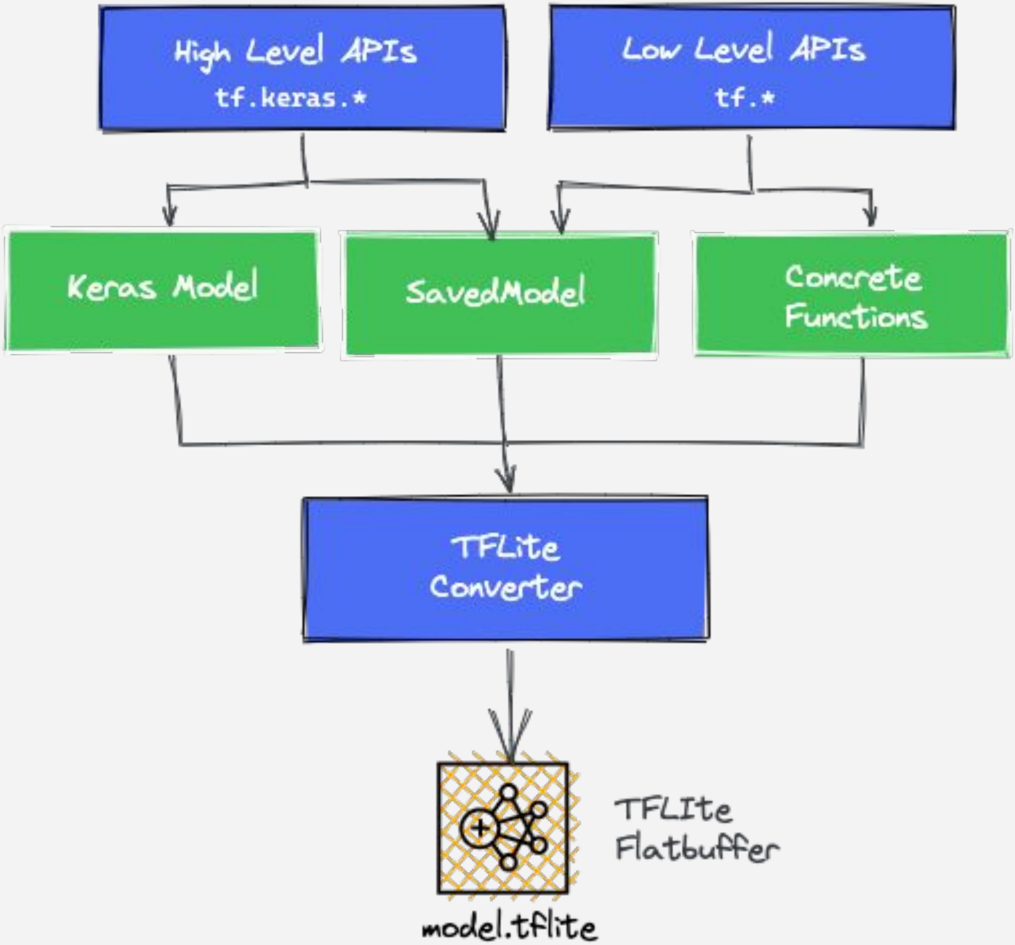
Was this helpful?

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 quantize an already-trained float TensorFlow model when you convert it to TensorFlow Lite format using the TensorFlow Lite Converter.

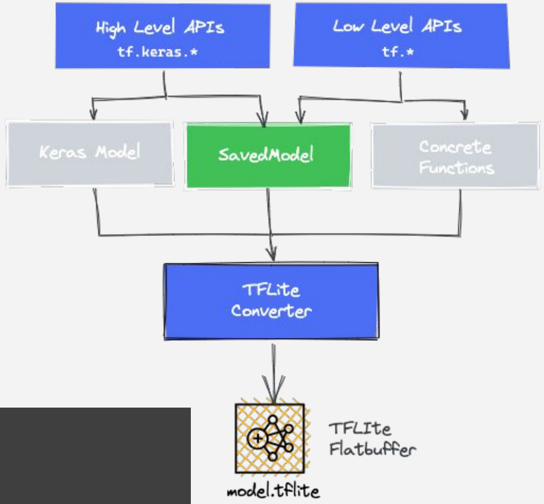
https://www.tensorflow.org/lite/performance/post_training_quantization

TensorFlow

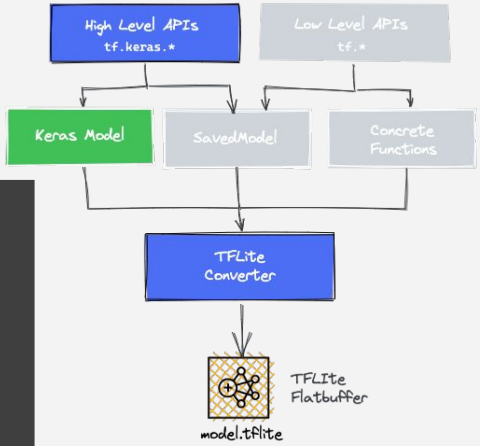


TensorFlow Lite

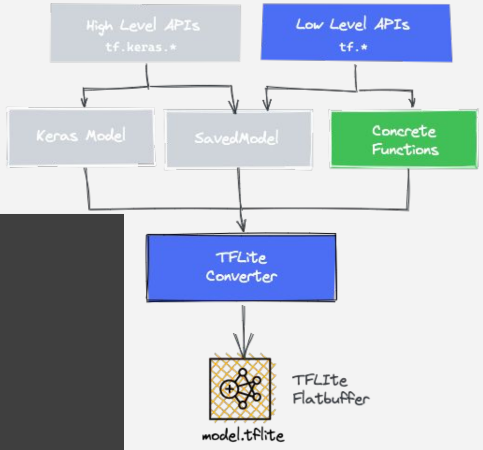
```python
import tensorflow as tf

# Convert the model
# path to the SavedModel directory
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```



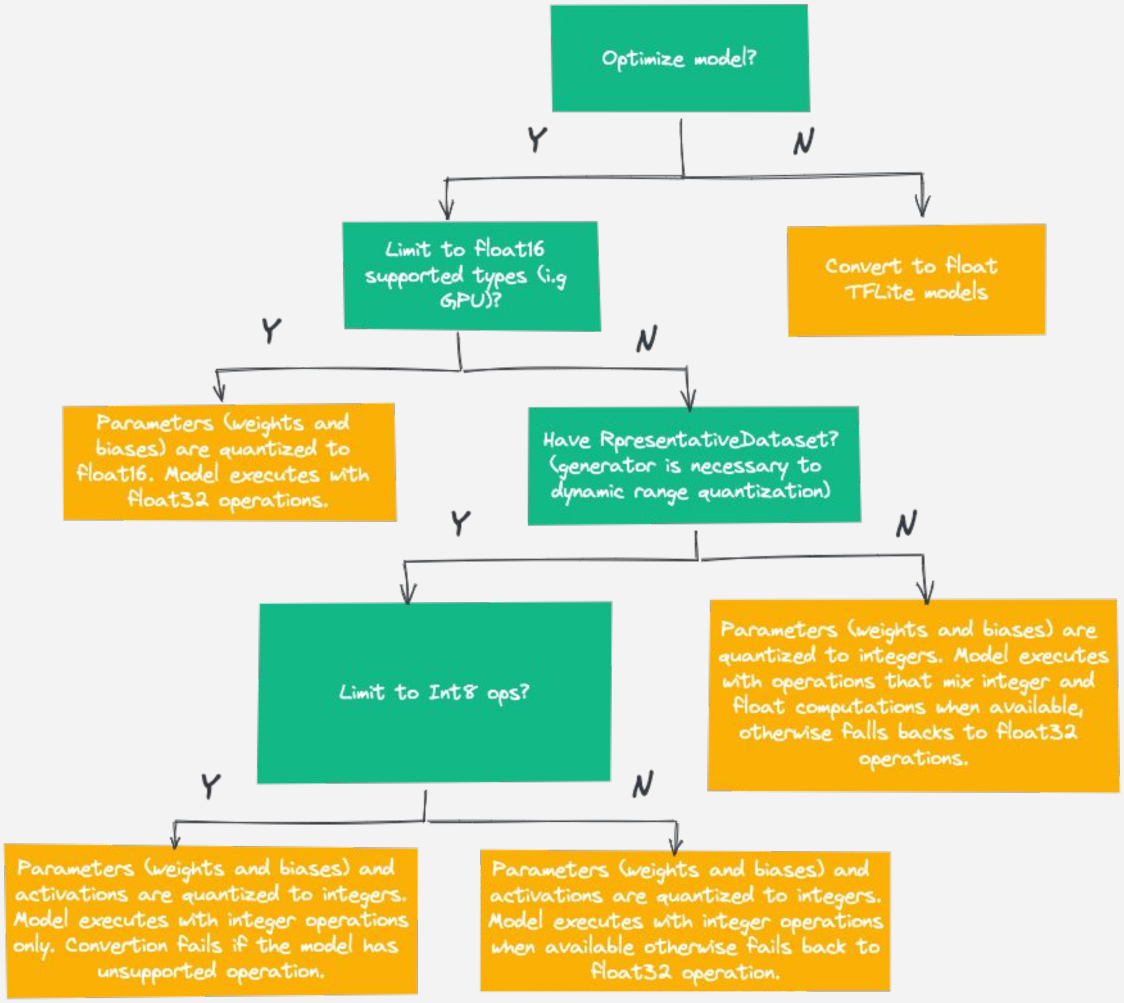
```
import tensorflow as tf

# Create a model using high-level tf.keras.* APIs
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1]),
    tf.keras.layers.Dense(units=16, activation='relu'),
    tf.keras.layers.Dense(units=1)
])
model.compile(optimizer='sgd', loss='mean_squared_error') # compile the model
model.fit(x=[-1, 0, 1], y=[-3, -1, 1], epochs=5) # train the model
# (to generate a SavedModel) tf.saved_model.save(model, "saved_model_keras_dir")

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```



```
import tensorflow as tf

# Create a model using low-level tf.* APIs
class Squared(tf.Module):
  @tf.function(input_signature=[tf.TensorSpec(shape=[None], dtype=tf.float32)])
  def __call__(self, x):
    return tf.square(x)
model = Squared()
# (ro run your model) result = Squared(5.0) # This prints "25.0"
# (to generate a SavedModel) tf.saved_model.save(model, "saved_model_tf_dir")
concrete_func = model.__call__.get_concrete_function()

# Convert the model.

converter = tf.lite.TFLiteConverter.from_concrete_functions([concrete_func],model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

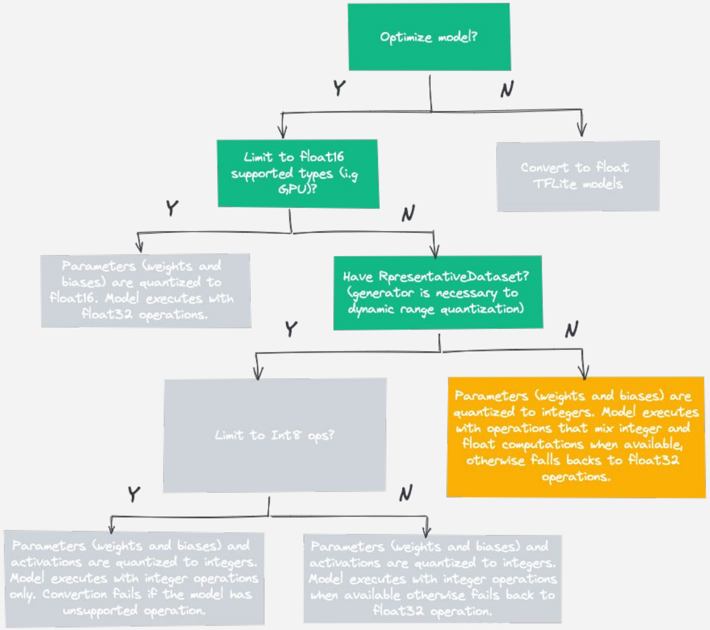


How to enable post-training integer quantization?


Optimize model?
Y
N
Limit to float16 supported types (i.g GPU)?
Convert to float TFLite models
Y
N
Parameters (weights and biases) are quantized to float16. Model executes with float32 operations.
Have RpresentativeDataset? (generator is necessary to dynamic range quantization)
Y
N
Limit to Int8 ops?
Parameters (weights and biases) are quantized to integers. Model executes with operations that mix integer and float computations when available, otherwise falls backs to float32 operations.
Y
N
Parameters (weights and biases) and activations are quantized to integers. Model executes with integer operations only. Convertion fails if the model has unsupported operation.
Parameters (weights and biases) and activations are quantized to integers. Model executes with integer operations when available otherwise fails back to float32 operation.

Dynamic range quantization

Dynamic range quantization is a recommended starting point because it provides reduced memory usage and faster computation without you having to provide a representative dataset for calibration. This type of quantization, statically **quantizes only the weights and biases** from floating point to integer at conversion time, which provides 8-bits of precision.




```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model_quant = converter.convert()
```

Representation for quantized tensors

8-bit quantization approximates floating point values using the following formula.

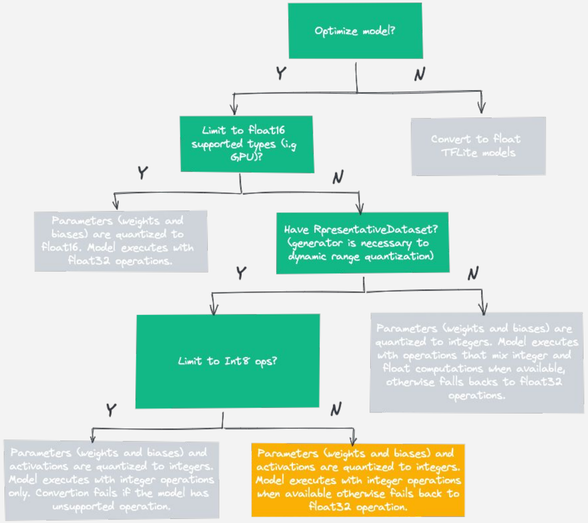
$$real\_value = (int8\_value - zero\_point) \times scale$$

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].

https://www.tensorflow.org/lite/performance/post_training_quantization

Full integer quantization
Integer with float fallback

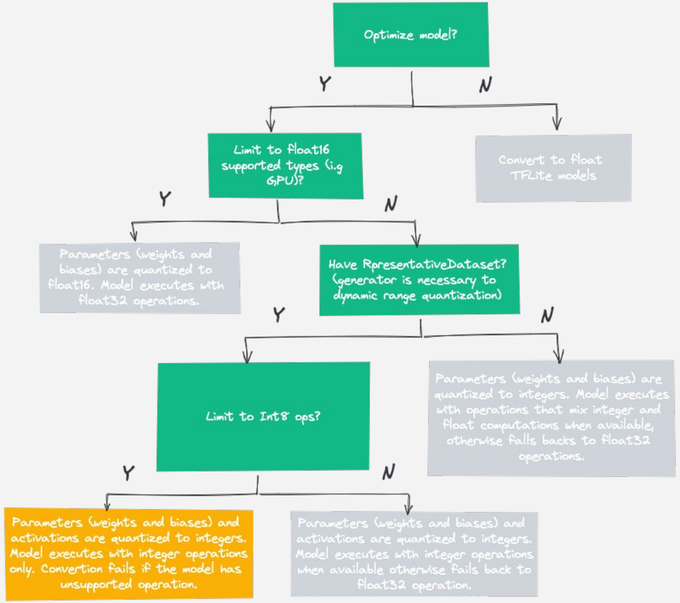


Unlike constant tensors such as weights and biases, variable tensors such as model input, activations (outputs of intermediate layers) and model output cannot be calibrated unless we run a few inference cycles.

```
def representative_data_gen():
 for input_value in tf.data.Dataset.from_tensor_slices(train_images).batch(1).take(100):
   # Model has only one input so each data point has one element.
   yield [input_value]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen

tflite_model_quant = converter.convert()
```

Full integer quantization
Integer only

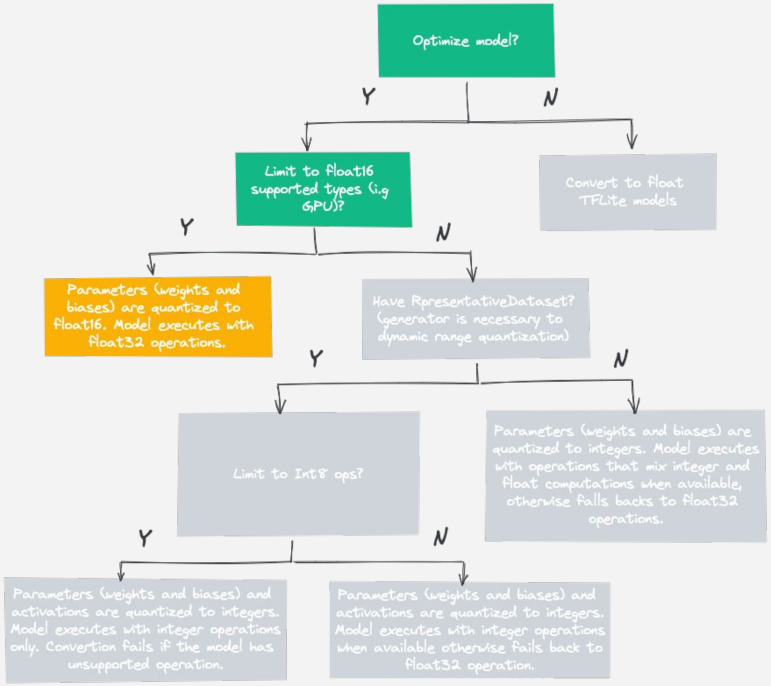
Additionally, to ensure compatibility with integer only devices (such as 8-bit microcontrollers) and accelerators (such as the Coral Edge TPU), you can enforce full integer quantization for all ops including the input and output.




```
converter = tf.lite.TFLiteConverter.from_keras_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8  # or tf.uint8
converter.inference_output_type = tf.int8  # or tf.uint8
tflite_quant_model = converter.convert()
```

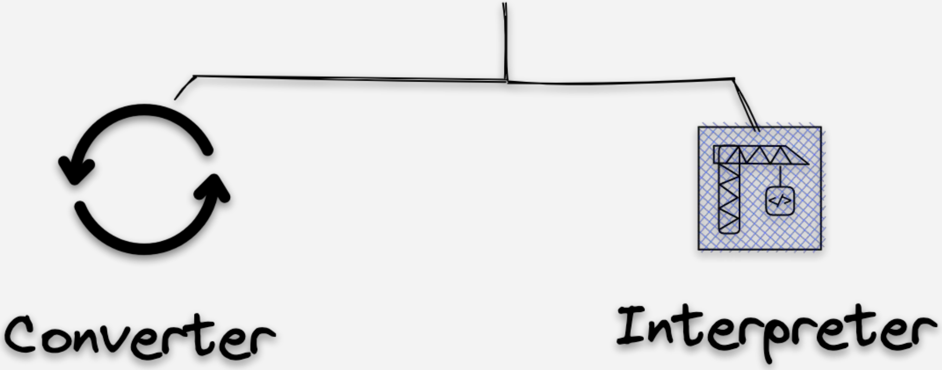
Float16 quantization

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.



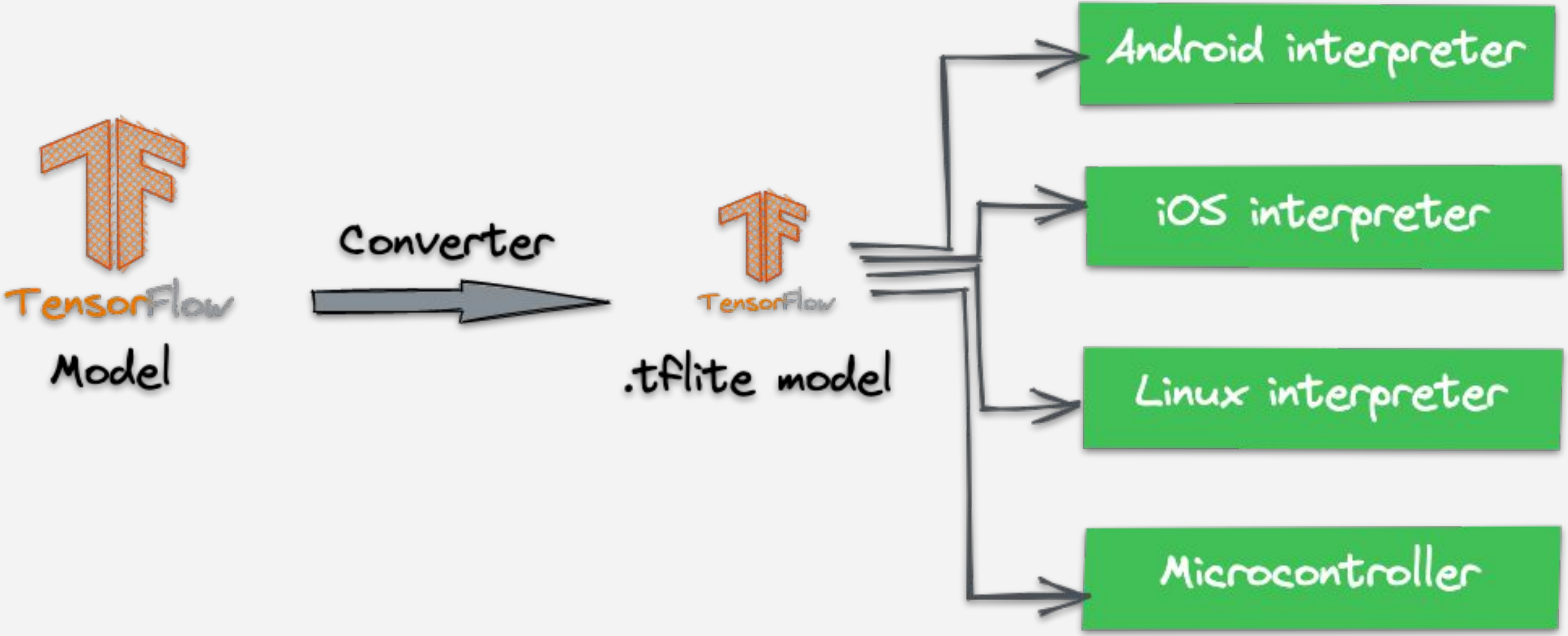

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
```

TensorFlow Lite



Convert TF models into a space-efficient format
For use on memory-constrained devices. It can
Apply optimizations that further reduce the model
Size and mate it run faster on small devices

It runs an appropriately converted TF Lite model using the most efficient operations for a given device


TensorFlow
Model
Converter
TensorFlow
.tflite model
Android interpreter
iOS interpreter
Linux interpreter
Microcontroller

Run the TensorFlow Lite models?

```
model.predict(text_y)
```

Run the TensorFlow Lite models

1. Instantiate an Interpreter object.
2. Call some methods that allocate memory for the model.
3. Write the input to the input tensor.
4. Invoke the model.
5. Read the output from the output tensor.

Run the TensorFlow Lite models

1. Instantiate an Interpreter object.
2. Call some methods that allocate memory for the model.
3. Write the input to the input tensor.
4. Invoke the model.
5. Read the output from the output tensor.

```
# Initialize the interpreter and allocate tensors
 interpreter = tf.lite.Interpreter(model_path=str(tflite_file))
 interpreter.allocate_tensors()
```

Run the TensorFlow Lite models (Get info about input)

```
interpreter.get_input_details()[0]


{'name': 'serving_default_dense_2_input:0',
 'index': 0,
 'shape': array([1, 1], dtype=int32),
 'shape_signature': array([-1,  1], dtype=int32),
 'dtype': numpy.int8,
 'quantization': (0.024556981399655342, -128),
 'quantization_parameters': {'scales': array([0.02455698], dtype=float32),
  'zero_points': array([-128], dtype=int32),
  'quantized_dimension': 0},
 'sparsity_parameters': {}}
```

Run the TensorFlow Lite models (Get info about output)

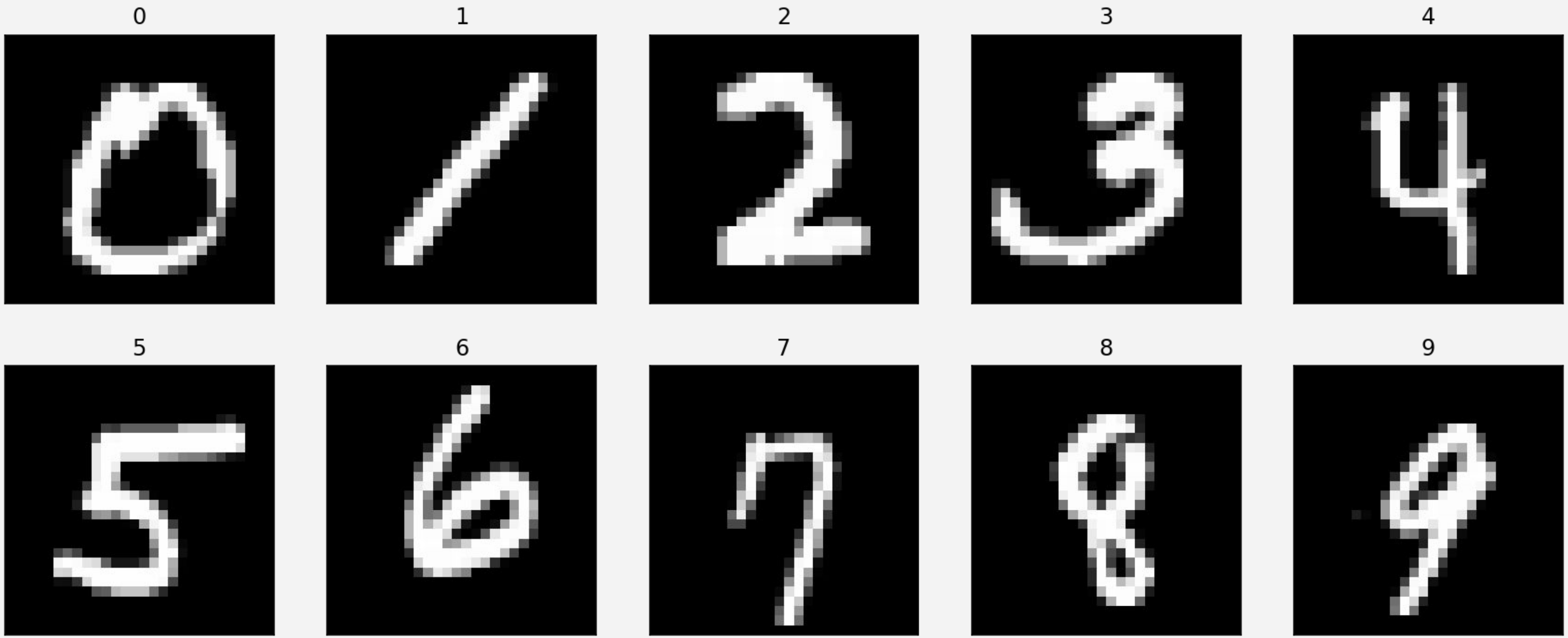
```
interpreter.get_output_details()[0]

{'name': 'StatefulPartitionedCall:0',
 'index': 9,
 'shape': array([1, 1], dtype=int32),
 'shape_signature': array([-1,  1], dtype=int32),
 'dtype': numpy.int8,
 'quantization': (0.008335912600159645, -3),
 'quantization_parameters': {'scales': array([0.00833591], dtype=float32),
  'zero_points': array([-3], dtype=int32),
  'quantized_dimension': 0},
 'sparsity_parameters': {}}
```

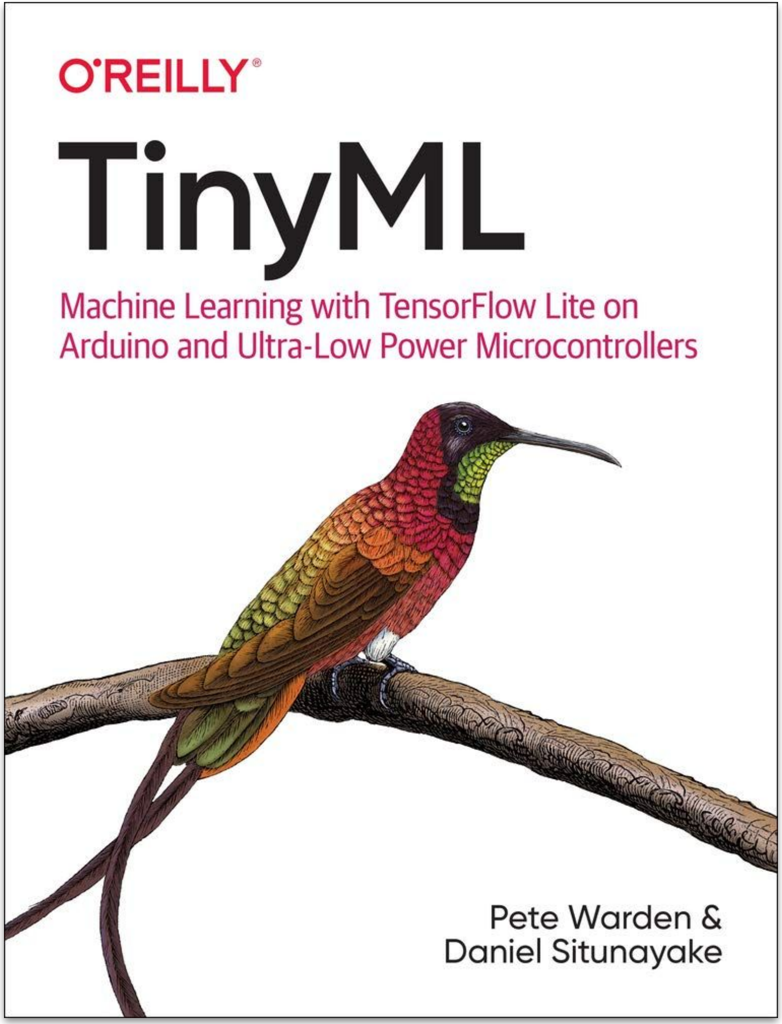
Run the TensorFlow Lite models

1. Instantiate an Interpreter object.
2. Call some methods that allocate memory for the model.
3. Write the input to the input tensor.
4. Invoke the model.
5. Read the output from the output tensor.

```
interpreter.set_tensor(input_details["index"], test_image)
interpreter.invoke()
output = interpreter.get_tensor(output_details["index"])[0]
```

Run the TensorFlow Lite models





Chapter 4

TinyML Book Screencast #4 - Quantization

https://www.youtube.com/watch?v=-jBmqY_aFwE

HarvardX Profession Certificate in Tiny Machine Learning (TinyML)

https://github.com/tinyMLx/courseware/tree/master/edX
Chapter 3.3