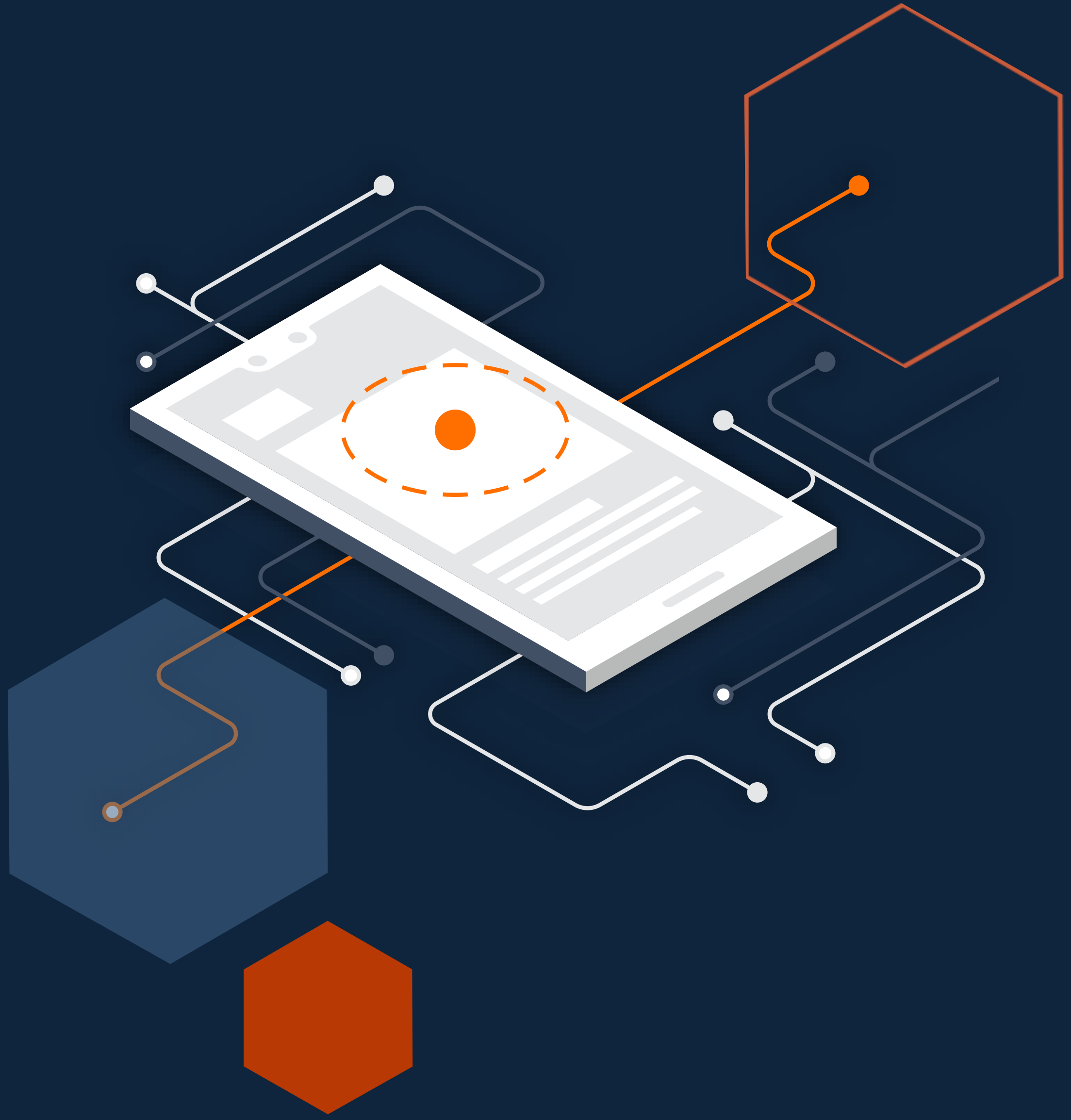
Post-Training Quantization with TensorFlow

Yfantidis
Dimitrios



Post-Training Quantization Techniques

Dynamic Range Quantization

- Statically quantizes only the weights (not biases) from floating point to integer at conversion time.
- Provides 8-bits of precision.
- Does not require a representative dataset for calibration.
- The outputs are still stored using floating point.

Full Integer Quantization

- Compatibility with integer only hardware devices or accelerators.
- Variable tensors such as model input, activations (outputs of intermediate layers) and model output are calibrated as well.
- A (small) representative dataset is required to calibrate them.

Other Quantization Techniques

- Integer only
- Float16 quantization
- 16-bit activations with 8-bit weights
- Depends on converter.target_spec.supported_ops, converter.inference_input_type and converter.inference_output_type

TFLite 8-bit quantization specification

```
FULLY_CONNECTED
  Input 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8
    range      : [-127, 127]
    granularity: per-axis (dim = 0)
    restriction: zero_point = 0
  Input 2 (Bias):
    data_type  : int32
    range      : [int32_min, int32_max]
    granularity: per-tensor
    restriction: (scale, zero_point) = (input0_scale * input1_scale[...], 0)
  Output 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
```

Biases are quantized as well, but as 32-bit integers instead of 8-bit ones. This is forced by TensorFlow as:

1. Biases occupy less space (linear space complexity) relative to weights (quadratic space complexity).
2. Each bias-vector entry is added to many output activations. Thus, any quantization error in the bias-vector tends to act as an overall bias. In other words, the model's accuracy would plunge, as the biases' quantization error would have greater impact to the computations.

TFLite 8-bit quantization specification

This is also true for Convolutional Tensors

```
CONV_2D
  Input 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8
    range      : [-127, 127]
    granularity: per-axis (dim = 0)
    restriction: zero_point = 0
  Input 2 (Bias):
    data_type  : int32
    range      : [int32_min, int32_max]
    granularity: per-axis
    restriction: (scale, zero_point) = (input0_scale * input1_scale[...], 0)
  Output 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
```

Biases are quantized as well, but as 32-bit integers instead of 8-bit ones. This is forced by TensorFlow as:

1. Biases occupy less space (linear space complexity) relative to weights (quadratic space complexity).
2. Each bias-vector entry is added to many output activations. Thus, any quantization error in the bias-vector tends to act as an overall bias. In other words, the model's accuracy would plunge, as the biases' quantization error would have greater impact to the computations.

TFLite 8-bit quantization specification

And for Convolutional Tensors with different kernels in each channel

```
DEPTHWISE_CONV_2D
  Input 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8
    range      : [-127, 127]
    granularity: per-axis (dim = 3)
    restriction: zero_point = 0
  Input 2 (Bias):
    data_type  : int32
    range      : [int32_min, int32_max]
    granularity: per-axis
    restriction: (scale, zero_point) = (input0_scale * input1_scale[...], 0)
  Output 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
```

Biases are quantized as well, but as 32-bit integers instead of 8-bit ones. This is forced by TensorFlow as:

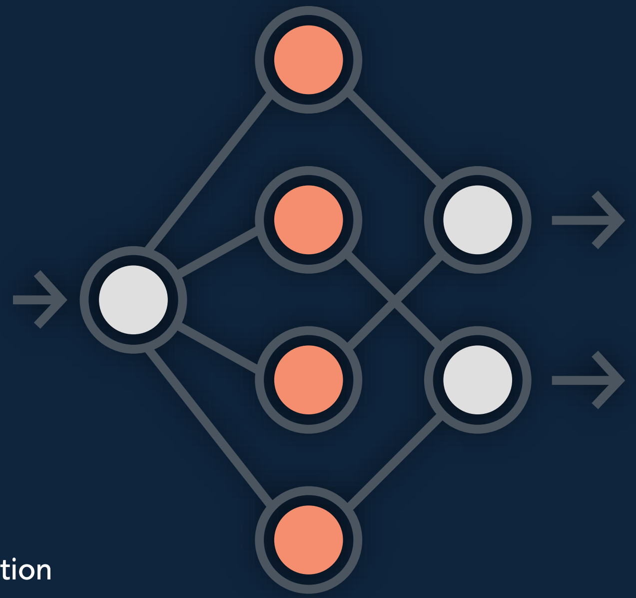
1. Biases occupy less space (linear space complexity) relative to weights (quadratic space complexity).
2. Each bias-vector entry is added to many output activations. Thus, any quantization error in the bias-vector tends to act as an overall bias. In other words, the model's accuracy would plunge, as the biases' quantization error would have greater impact to the computations.

Quantization Formula

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

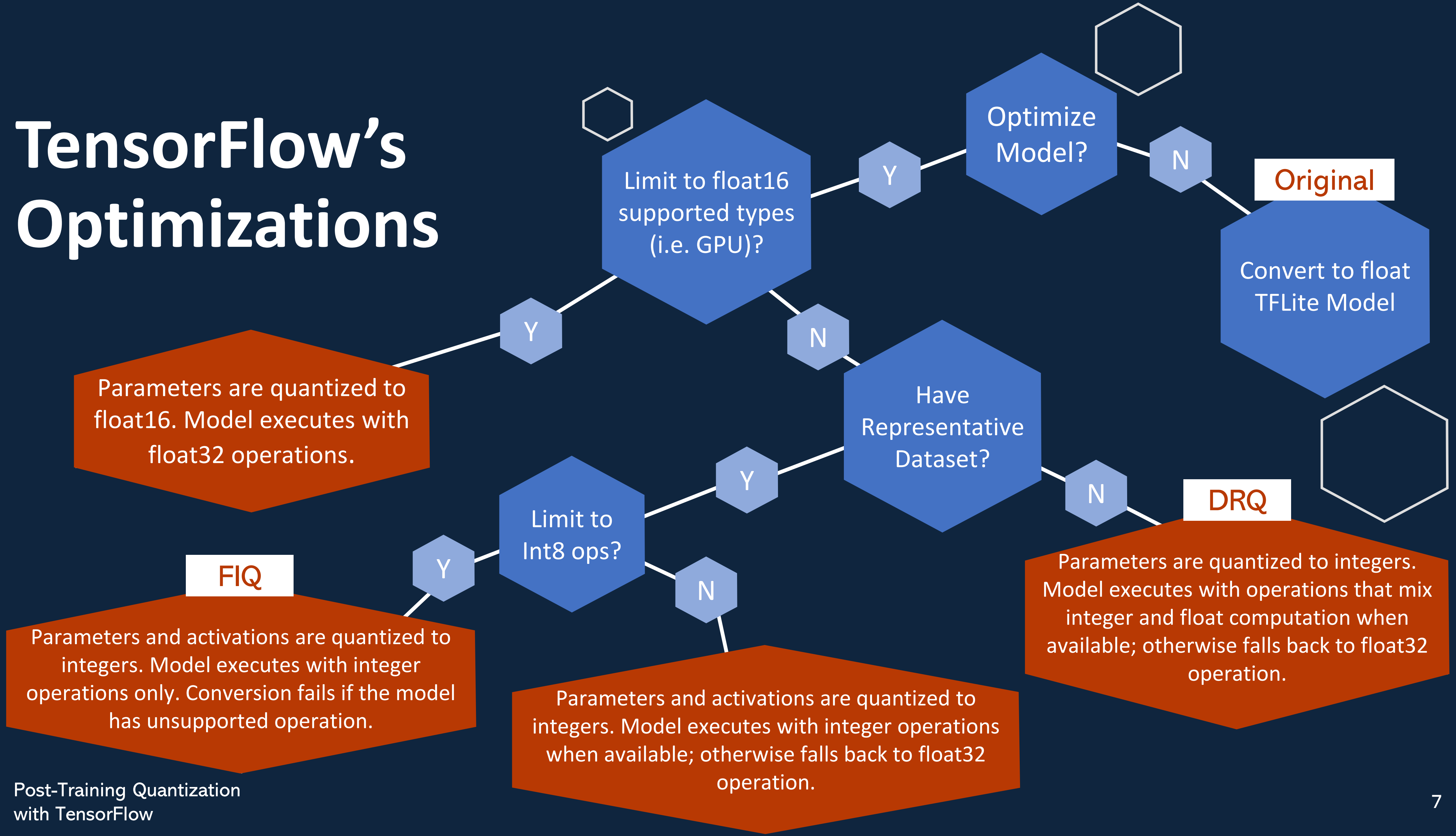
$$r = S(q - Z)$$

which is an affine mapping of integers q to real numbers r.



- S (for "scale") is an arbitrary positive real number. It is typically represented in software as a floating-point quantity, like the real values r.
- Z (for "zero-point") is of the same type as quantized values q, and is in fact the quantized value q corresponding to the real value 0. This allows us to automatically meet the requirement that the real value r = 0 be exactly representable by a quantized value. The motivation for this requirement is that efficient implementation of neural network operators often requires zero-padding of arrays around boundaries.

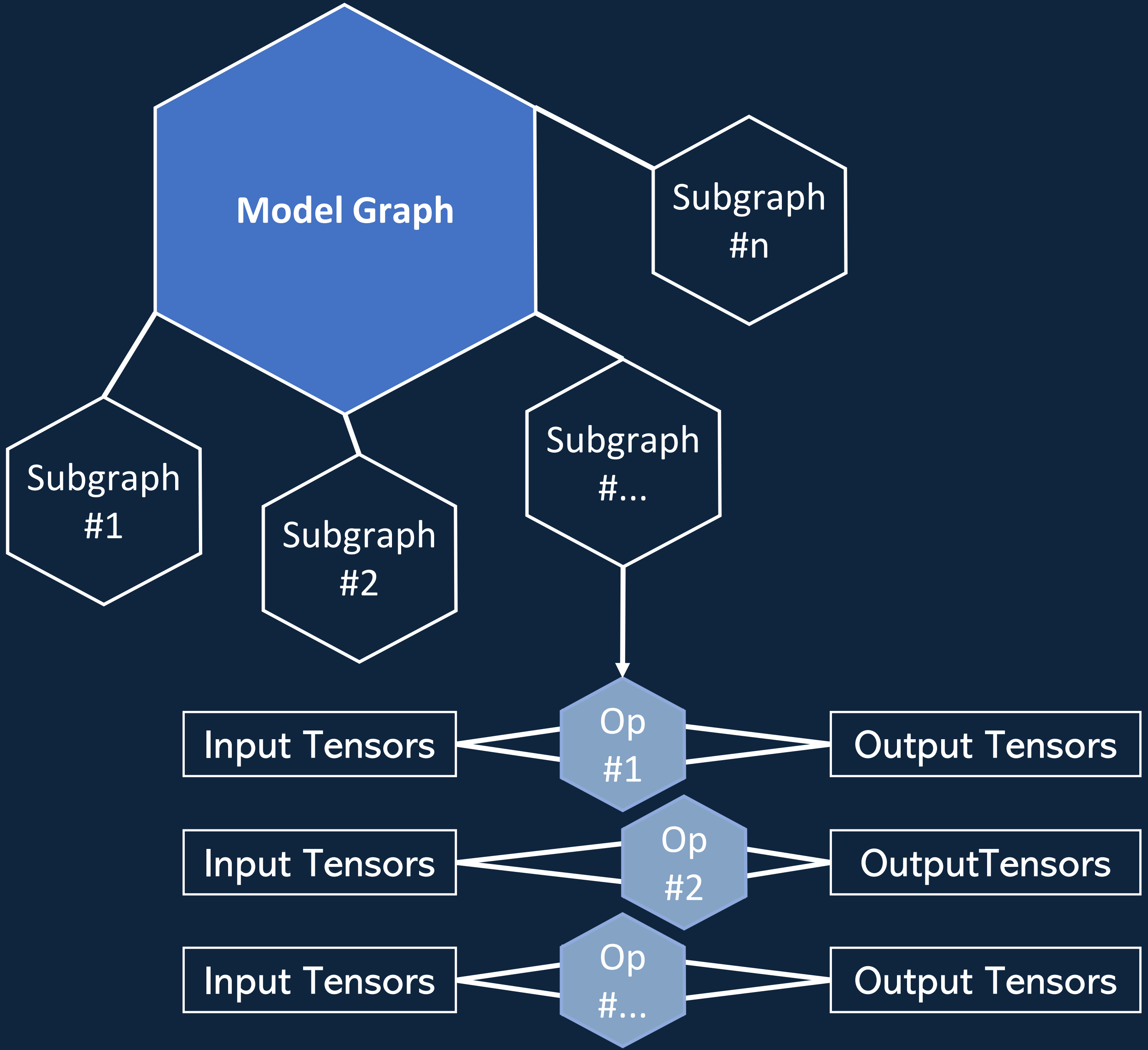
TensorFlow’s Optimizations



TensorFlow Lite Analyzer

Inspecting TFLite Model Structure

TFLite Model Structure



TFLite Model Structure

Each TFLite model is, essentially, a graph. This graph consists of subgraphs:

$$G = \{S_1, S_2, \dots, S_N\}$$

Each subgraph consists of a set of operators (nodes), O, and a set of tensors, T.

$$S_i = \left(\left\{O_1^{(i)}, O_2^{(i)}, \dots, O_{n_i}^{(i)}\right\}, T^{(i)}\right)$$

In most cases, a TFLite model consists of just one subgraph.

Each operator is a function that does maps input tensors to output tensors:

$$O_j^{(i)}: T_{in}^{(i,j)} \rightarrow T_{out}^{(i,j)}$$

$$\left(T_{in}^{(i,j)}, T_{out}^{(i,j)} \subset T^{(i,j)} \right) \wedge \left(T_{in}^{(i,j)} \cap T_{out}^{(i,j)} = \emptyset \right)$$

An operator may correspond to a layer of the equivalent Keras model, but not necessarily.

Model Analysis Output

Log File:

```
=== ./local/path/to/model.tflite ===

Your TFLite model has '1' subgraph(s). In the subgraph description below,
T# represents the Tensor numbers [...]

Subgraph#0 main(T#0) -> [T#64]
  Op#0 QUANTIZE(T#0) -> [T#39]
  Op#1 CONV_2D(T#39, T#5, T#6[3, -1366, -1099, -677, -146, ...]) -> [T#40]
  Op#2 CONV_2D(T#40, T#7, T#8[-107, -828, -1007, -693, -261, ...]) -> [T#41]
  [...]

Tensors of Subgraph#0
  T#0(serving_default_x:0) shape:[1, 128, 6, 1], type:UINT8
  T#1(arith.constant24) shape:[3], type:INT32 RO 12 bytes, buffer: 2, data:[0, -1, 0]
  T#2(arith.constant25) shape:[3], type:INT32 RO 12 bytes, buffer: 3, data:[0, 0, 0]
  T#3(arith.constant26) shape:[3], type:INT32 RO 12 bytes, buffer: 4, data:[1, 1, 1]
  T#4(Ordonez2016DeepOriginal/reshape/Reshape/shape) shape:[3], type:INT32 RO 12 bytes, buffer: 5, data:[1, 112, 384]
  T#5(Ordonez2016DeepOriginal/conv2d/Conv2D) shape:[64, 5, 1, 1], type:INT8 RO 320 bytes, buffer: 6, data:[T, ., R, q, p, ...]
  [...]

---------------------------------------------------------------
Your TFLite model has 'k' signature_def(s).

Signature#0 key: 'serving_default'
- Subgraph: Subgraph#0
- Inputs:
    'x' : T#0
- Outputs:
    'output_0' : T#64

---------------------------------------------------------------
              Model size:      477680 bytes
    Non-data buffer size:       15872 bytes (03.32 %)
  Total data buffer size:      461808 bytes (96.68 %)
    (Zero value buffers):          12 bytes (00.00 %)

[...]
```

The output for any TFLite Model should have a similar structure

Model Structure Report

Number of Tensors	Original TFLite Model	DRQ Compressed TFLite Model	FIQ Compressed TFLite Model
Int8	0	19 ↑	38 ↑↑
Int16	0	0 ~	2 ↑
Int32	4	4 ~	17 ↑↑
Float32	49	30 ↓	8 ↓↓
Total	53	53 ~	65 ↑

Quantization Metrics

Metric Name	Original TFLite Model	DRQ Compressed TFLite Model	FIQ Compressed TFLite Model
Accuracy	0.743	0.745 ↑	0.554 ↓
Precision	0.654	0.656 ↑	0.490 ↓
Recall	0.749	0.751 ↑	0.567 ↓
F1	0.684	0.685 ↑	0.503 ↓
Balanced Accuracy	0.749	0.751 ↑	0.567 ↓
Cohen's kappa	0.692	0.694 ↑	0.465 ↓

Quantization Debugger

Inspecting Quantization Errors (FIQ only)

Quantization Debugger

1 Debugger preparation

Provide the converter that you have been using to quantize the model.

2 Running the debugger

Logs differences between float tensors and quantized tensors for the same op location, and process them with given metrics.

3 Data analysis

Add some useful metrics derived from the debugger's outputs to find problematic layers.

4 Selective Quantization

When correct layers are skipped, we can expect some model quality recovery at the cost of increased latency and model size.

5 Advanced usages

Custom metrics, internal MLIR API access & selective quantization from an already calibrated model.

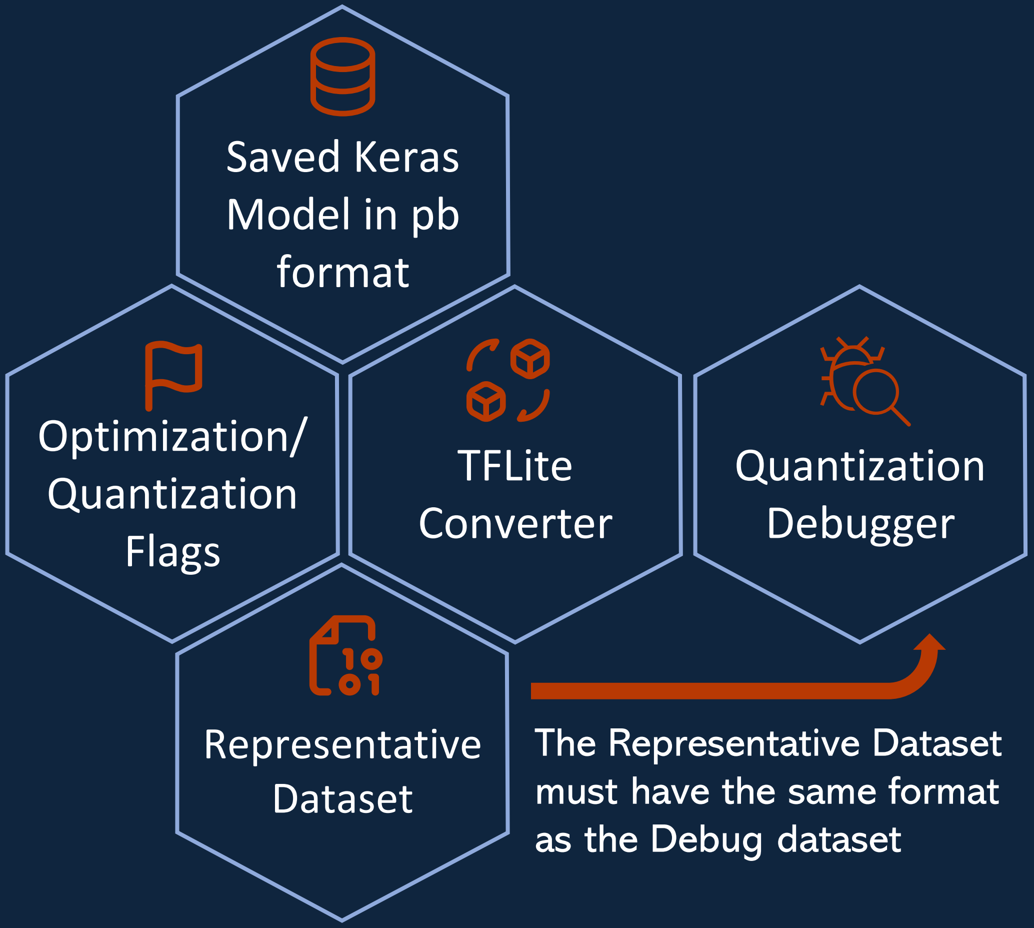
```python
converter = tf.lite.TFLiteConverter.from_saved_model(model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
debugger = tf.lite.experimental.QuantizationDebugger(
    converter=converter, debug_dataset=representative_dataset
)
```

Debugger Preparation

In order to quantize our model, we used a tf.lite.TFLiteConverter instance which was specified with appropriate flags.

The most important flag is tf.lite.Optimize.DEFAULT as it specifies that the converter will not only produce the TFLite model, but it will apply quantization to it as well.

Should tf.lite.Optimize.DEFAULT or the Representative Dataset be absent from the converter.optimizations property, the Quantization Debugger would raise an error.



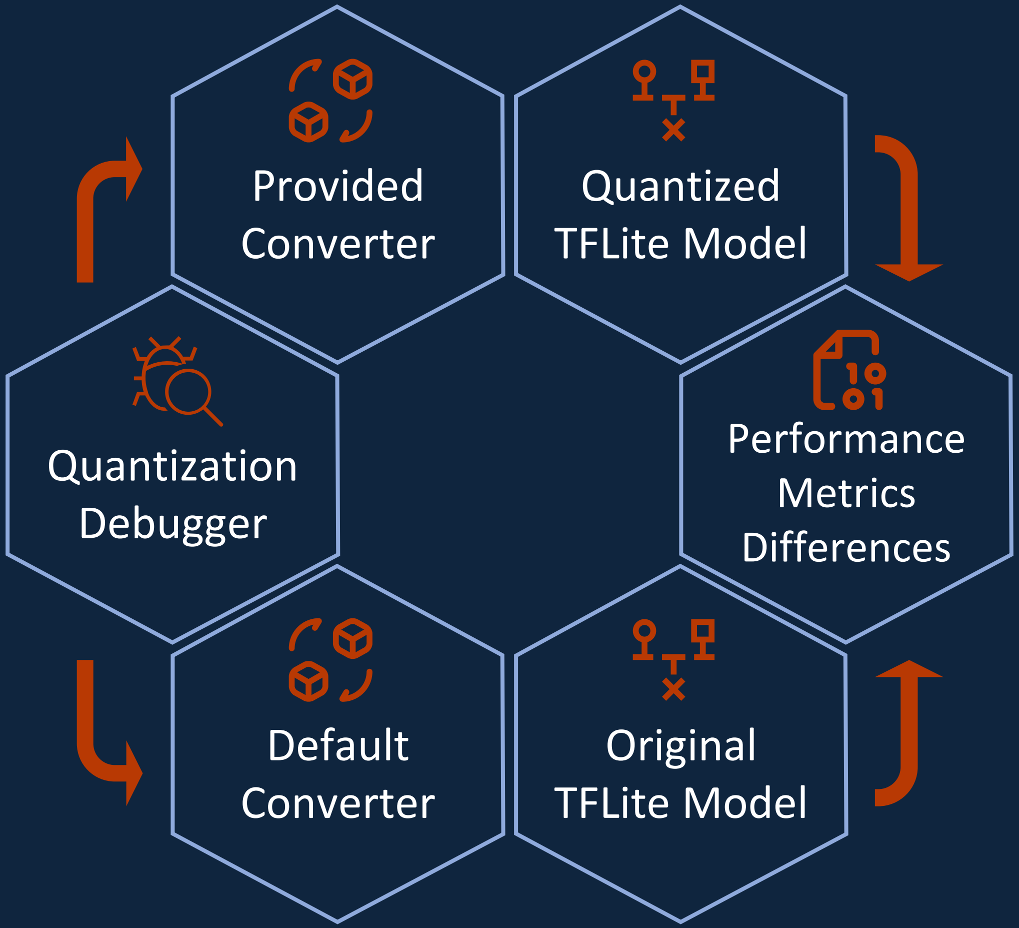
Running the Debugger

```
debugger.run()
RESULTS_FILE = '/tmp/debugger_results.csv'
with open(RESULTS_FILE, 'w') as f:
  debugger.layer_statistics_dump(f)
layer_stats = pd.read_csv(RESULTS_FILE)
layer_stats.head()
```

By calling QuantizationDebugger.run(), the debugger will compute the differences between float tensors and quantized tensors for the same op location, and process them with given metrics.

The QuantizationDebugger.layer_statistics field is a dictionary through which quantization parameters, tensor names, tensor indices and their processed metrics can be accessed.

They could also be stored as a CSV file to analyze them with Pandas or other data processing libraries.



Data Analysis of FIQ Model's metrics

	op_name	tensor_idx	num_elements	stddev	mean_error	max_abs_error	mean_squared_error	scale	zero_point	tensor_name
0	CONV_2D	79	47616.0	0.009758	0.000169	0.074279	0.000096	0.054382	-128	Ordonez2016DeepOriginal/conv2d/Relu;Ordonez201...
1	CONV_2D	83	46080.0	0.034338	-0.004216	0.175095	0.001232	0.305846	-128	Ordonez2016DeepOriginal/conv2d_1/Relu;Ordonez2...
2	CONV_2D	87	44544.0	0.114489	-0.006365	0.600643	0.015137	1.067677	-128	Ordonez2016DeepOriginal/conv2d_2/Relu;Ordonez2...
3	CONV_2D	91	43008.0	0.540942	-0.025463	3.032404	0.346328	5.625286	-128	Ordonez2016DeepOriginal/conv2d_3/Relu;Ordonez2...
4	RESHAPE	95	43008.0	0.000000	0.000000	0.000000	0.000000	5.625286	-128	Ordonez2016DeepOriginal/reshape/Reshape
5	UNIDIRECTIONAL_SEQUENCE_LSTM	111	14336.0	0.033766	-0.000542	0.849846	0.001784	0.007843	-1	tfl.unidirectional_sequence_lstm
6	UNIDIRECTIONAL_SEQUENCE_LSTM	127	14336.0	0.005990	-0.000052	0.158288	0.000053	0.007843	0	tfl.unidirectional_sequence_lstm1
7	STRIDED_SLICE	131	128.0	0.000000	0.000000	0.000000	0.000000	0.007843	0	tfl.strided_slice
8	FULLY_CONNECTED	135	6.0	0.047605	-0.004853	0.088034	0.003024	0.164402	11	Ordonez2016DeepOriginal/act_smx/MatMul;Ordonez...
9	SOFTMAX	139	6.0	0.001321	-0.000573	0.003476	0.000002	0.003906	-128	StatefulPartitionedCall:0

Data Analysis of FIQ Model's metrics

	op_name	range	rmse/scale
0	CONV_2D	13.867427	0.180051
1	CONV_2D	77.990845	0.114751
2	CONV_2D	272.257660	0.115232
3	CONV_2D	1434.447930	0.104616
4	RESHAPE	1434.447930	0.000000
5	UNIDIRECTIONAL_SEQUENCE_LSTM	2.000000	5.385343
6	UNIDIRECTIONAL_SEQUENCE_LSTM	1.999981	0.932456
7	STRIDED_SLICE	1.999981	0.000000
8	FULLY_CONNECTED	41.922566	0.334514
9	SOFTMAX	0.996094	0.380070

The ratio

$$\frac{RMSE}{S}$$

should approximate $1/\sqrt{12}$ ≈ 0.289 that occurs when quantized distribution is similar to the original float distribution, indicating a good quantized model.

The larger the value is, it's more likely for the layer not being quantized well. For example RMSE/S > 0.7 for the following layers:

	op_name	range	rmse/scale	tensor_name
5	UNIDIRECTIONAL_SEQUENCE_LSTM	2.000000	5.385343	tfl.unidirectional_sequence_lstm
6	UNIDIRECTIONAL_SEQUENCE_LSTM	1.999981	0.932456	tfl.unidirectional_sequence_lstm1

Selective Quantization

- Selective quantization skips quantization for some nodes, so that the calculation can happen in the original floating-point domain.
- We can expect some model quality recovery at the cost of increased latency and model size.
- Quantization debugger's option accepts denylisted_nodes and denylisted_ops options for skipping quantization for specific layers, or all instances of specific ops.
- Issues will arise if the model is planned to run on integer-only accelerators.

```
debug_options = tf.lite.experimental.QuantizationDebugOptions(
    denylisted_nodes=suspected_layers
    denylisted_ops=suspected_ops
)
debugger = tf.lite.experimental.QuantizationDebugger(
    converter=converter,
    debug_dataset=representative_dataset(ds),
    debug_options=debug_options
)
```

Combination of FIQ and DRQ

The Quantization Debugger allows partial Quantization of a TFLite Model.

In the next slides, this will be explored further to quantize the model partially with Full-Integer Quantization and partially with Dynamic Range Quantization.

Challenges

The Quantization Debugger demands the existence of a Representative Dataset in the provided converter.

Thus, should a TFLite model be examined using the converter that produced it, then it could not have been quantized with a scheme other than FIQ.

Inversely, a model that has been quantized with DRQ cannot be inspected through the quantization debugger.

```
self._data_gen = debug_dataset
self._debug_options = debug_options or QuantizationDebugOptions()
self.converter = None
self.calibrated_model = None
self.float_model = None
self._float_interpreter = None
if converter is not None:
  if self._debug_options.model_debug_metrics:
    old_optimizations = converter.optimizations
    self.converter = self._set_converter_options_for_float(converter)
    self.float_model = self.converter.convert()
    converter.optimizations = old_optimizations

  self.converter = self._set_converter_options_for_calibration(converter)
  self.calibrated_model = self.converter.convert()
  # Converter should be already set up with all options
  self._init_from_converter(
      self._debug_options,
      self.converter,
      self.calibrated_model,
      float_model=self.float_model)
else:
  self._quant_interpreter = _interpreter.Interpreter(
      quant_debug_model_path,
      quant_debug_model_content,
      experimental_preserve_all_tensors=(
          self._debug_options.layer_direct_compare_metrics is not None))
  if self._debug_options.model_debug_metrics:
    self._float_interpreter = _interpreter.Interpreter(
        float_model_path, float_model_content)
self._initialize_stats()
```

Challenges

This can be validated by examining the Quantization Debugger's constructor (right).

It is evident that if a converter is provided, it is necessary to have the optimizations and representative_dataset fields registered, as seen in the code segment below.

```
def _set_converter_options_for_calibration(
    self, converter: TFLiteConverter) -> TFLiteConverter:
  """Verify converter options and set required experimental options."""
  if not converter.optimizations:
    raise ValueError(
        'converter object must set optimizations to lite.Optimize.DEFAULT')
  if not converter.representative_dataset:
    raise ValueError('converter object must set representative_dataset')

  converter.experimental_mlir_quantizer = True
  converter._experimental_calibrate_only = True  # pylint: disable=protected-access
  return converter
```

```
self._data_gen = debug_dataset
self._debug_options = debug_options or QuantizationDebugOptions()
self.converter = None
self.calibrated_model = None
self.float_model = None
self._float_interpreter = None
if converter is not None:
  if self._debug_options.model_debug_metrics:
    old_optimizations = converter.optimizations
    self.converter = self._set_converter_options_for_float(converter)
    self.float_model = self.converter.convert()
    converter.optimizations = old_optimizations

  self.converter = self._set_converter_options_for_calibration(converter)
  self.calibrated_model = self.converter.convert()
  # Converter should be already set up with all options
  self._init_from_converter(
      self._debug_options,
      self.converter,
      self.calibrated_model,
      float_model=self.float_model)
else:
  self._quant_interpreter = _interpreter.Interpreter(
      quant_debug_model_path,
      quant_debug_model_content,
      experimental_preserve_all_tensors=(
          self._debug_options.layer_direct_compare_metrics is not None))
  if self._debug_options.model_debug_metrics:
    self._float_interpreter = _interpreter.Interpreter(
        float_model_path, float_model_content)
self._initialize_stats()
```

Challenges

This could be possible through strategic manipulation of the other input arguments in the Quantization Debugger's constructor:

```
class QuantizationDebugger(
    quant_debug_model_path: str | None = None,
    quant_debug_model_content: bytes | None = None,
    float_model_path: str | None = None,
    float_model_content: bytes | None = None,
    debug_dataset: (() -> Iterable[Sequence[ndarray]]) | None = None,
    debug_options: QuantizationDebugOptions | None = None,
    converter: Any | None = None
)
```

```
self._data_gen = debug_dataset
self._debug_options = debug_options or QuantizationDebugOptions()
self.converter = None
self.calibrated_model = None
self.float_model = None
self._float_interpreter = None
if converter is not None:
  if self._debug_options.model_debug_metrics:
    old_optimizations = converter.optimizations
    self.converter = self._set_converter_options_for_float(converter)
    self.float_model = self.converter.convert()
    converter.optimizations = old_optimizations

  self.converter = self._set_converter_options_for_calibration(converter)
  self.calibrated_model = self.converter.convert()
  # Converter should be already set up with all options
  self._init_from_converter(
      self._debug_options,
      self.converter,
      self.calibrated_model,
      float_model=self.float_model)
else:
  self._quant_interpreter = _interpreter.Interpreter(
      quant_debug_model_path,
      quant_debug_model_content,
      experimental_preserve_all_tensors=(
          self._debug_options.layer_direct_compare_metrics is not None))
  if self._debug_options.model_debug_metrics:
    self._float_interpreter = _interpreter.Interpreter(
        float_model_path, float_model_content)
self._initialize_stats()
```

Quantization Aware Training

A possible solution to combat the drop in accuracy after FIQ compression.

Modification of the model's structure

```python
def Ordonez2016DeepOriginal(inp_shape, out_shape) -> tf.keras.Model:
    """
    @article{ordonez2016deep,
        title={Deep convolutional and {LSTM} recurrent neural networks for multimodal
        author={Ord{\'o}{\~n}ez, Francisco and Roggen, Daniel},
        journal={Sensors},
        volume={16},
        number={1},
        pages={115},
        year={2016},
        publisher={Multidisciplinary Digital Publishing Institute}
    }
    """
    nb_filters = 64
    drp_out_dns = .5
    nb_dense = 128

    inp = Input(inp_shape)

    x = Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(inp)
    x = Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = Reshape((x.shape[1],x.shape[2]*x.shape[3]))(x)
    act = LSTM(nb_dense, return_sequences=True, activation='tanh', name="lstm_1")(x)
    act = Dropout(drp_out_dns, name= "dot_1")(act)
    act = LSTM(nb_dense, activation='tanh', name="lstm_2")(act)
    act = Dropout(drp_out_dns, name= "dot_2")(act)
    out_act = Dense(out_shape, activation='softmax',  name="act_smx")(act)

    model = Model(inputs=inp, outputs=out_act, name="Ordonez2016DeepOriginal")
    return model
```

```python
def Ordonez2016DeepQuantAware(inp_shape, out_shape) -> tf.keras.Model:
    nb_filters = 64
    drp_out_dns = .5
    nb_dense = 128

    inp = keras.Input(inp_shape)

    x = keras.layers.Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(inp)
    x = keras.layers.Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = keras.layers.Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = keras.layers.Conv2D(nb_filters, kernel_size = (5,1),
               strides=(1,1), padding='valid', activation='relu')(x)
    x = keras.layers.Reshape((x.shape[1],x.shape[2]*x.shape[3]))(x)
    act = quantize_annotate_layer(
        keras.layers.LSTM(nb_dense, return_sequences=True, activation='tanh', name="lstm_1"),
        quantize_config=MyLSTMQuantizationConfig(8)
    )(x)
    act = keras.layers.Dropout(drp_out_dns, name= "dot_1")(act)
    act = quantize_annotate_layer(
        keras.layers.LSTM(nb_dense, activation='tanh', name="lstm_2"),
        quantize_config=MyLSTMQuantizationConfig(8)
    )(act)
    act = keras.layers.Dropout(drp_out_dns, name= "dot_2")(act)
    out_act = keras.layers.Dense(out_shape, activation='softmax',  name="act_smx")(act)

    model = keras.Model(inputs=inp, outputs=out_act, name="Ordonez2016DeepQuantAware")
    return model
```

The LSTM layers have been wrapped in the quantize_annotate_layer object that specifies a custom quantization configuration.

Encapsulation in TFLiteAPI.py

Exporting	Conversion	Evaluation	Analysis	Quantization Debugging
Simple interface for exporting a keras Sequential or Functional model to a “pb” model (SavedModel) format.	Easy construction of converter with pre-defined settings (flags, data types, representative dataset, etc.)	Inference testing of the produced TFLite model. Evaluates performance using multiple metrics (accuracy, precision, recall, confusion matrix, etc.)	Analyzes the TFLite model’s structre and exports it to a log file for ease-of-use inspection by the user.	Deploy the quantization debugger for the previously specified converter using a new representative dataset.

Thank you

Yfantidis Dimitrios

ydimitri@csd.auth.gr

School of Informatics,

Faculty of Sciences – AUTh

References

- TensorFlow – Official Guide:
  “Post-training quantization”
  “TensorFlow Lite 8-bit quantization specification”
  “Inspecting Quantization Errors with Quantization Debugger”
  “TensorFlow Lite 8-bit quantization specification”
- Mehrdad Zakershahrak (2023)
  “A Closer Look at Deep Learning Quantization Techniques”
- Benoit Jacob; Skirmantas Kligys; Bo Chen; Menglong Zhu; Matthew Tang; Andrew Howard; Hartwig Adam; Dmitry Kalenichenko (2017)
  “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”