tinyML® Summit

Miniature dreams can come true…

March 28-30, 2022 | San Francisco Bay Area

TINY ML

www.tinyML.org

Google

Model Optimization

with QKeras' Quantization-Aware Training and Vizier's Automatic Neural Architecture Search

Daniele Moro (danielemoro@google.com)

Google ASML Team and the Google Vizier Team

Quantization makes models tiny

~16x reduction in energy to go from **float32 to int8**

~1024x reduction in energy to go from **float32 to binary**

Leads to significant **reduction in latency and memory** usage as well

	ADD Energy (fJ)		MUL Energy (fJ)		MAC	
	45nm	**7nm**	**45nm**	**7nm**	**CPU64**	**ACE**
float32	900	380	3700	1310	1	1024
float16	400	160	1100	340	1	256
bfloat16	-	110	-	210	-	256
int32	100	30	3100	1480	-	1024
int8	30	7	200	70	1/8	64
int4	-	-	-	-	1/16	16
int2	-	-	-	-	1/32	4
binary	-	-	-	-	1/64	1

https://arxiv.org/pdf/2112.00133.pdf

Energy numbers collected from Google TPU hardware and modelled using PokeBNN's ACE metric

How can we quantize models?

Keras: Deep Learning for humans

This repository hosts the development of the Keras library. Read the documentation at keras.io.

About Keras

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result as fast as possible is key to doing good research.*

Keras is:

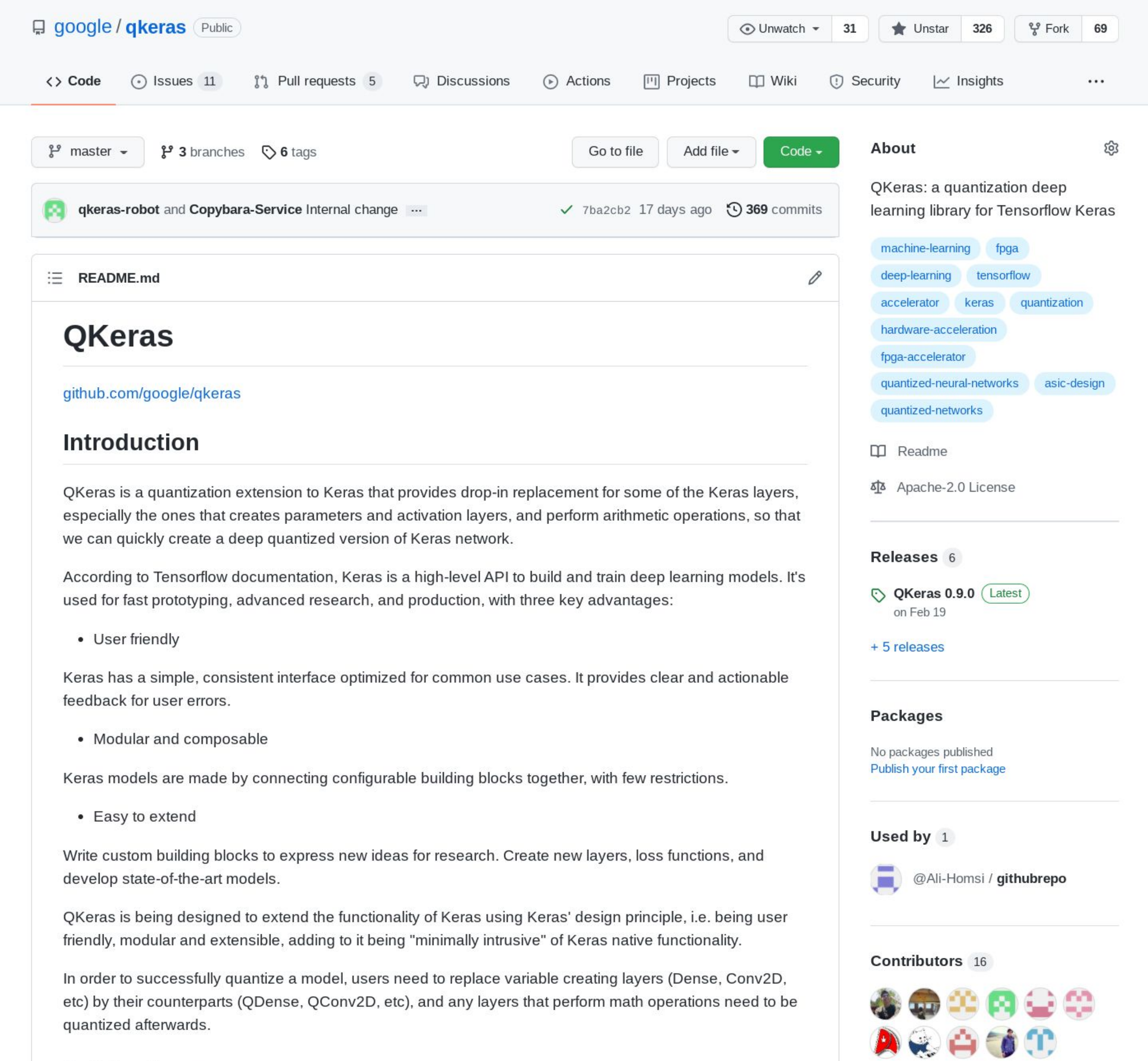
- **Simple** -- but not simplistic. Keras reduces developer *cognitive load* to free you to focus on the parts of the problem that really matter.
- **Flexible** -- Keras adopts the principle of *progressive disclosure of complexity*: simple workflows should be quick and easy, while arbitrarily advanced workflows should be *possible* via a clear path that builds upon what you've already learned.
- **Powerful** -- Keras provides industry-strength performance and scalability: it is used by organizations and companies including NASA, YouTube, or Waymo.

Keras is great for easily building ML models

… but it doesn’t support quantizing these models

QKeras

A ***quantization extension*** *to Keras that provides* ***drop-in replacements*** *for quantizing Keras layers*



google / qkeras Public

Unwatch 31 | Unstar 326 | Fork 69

Code | Issues 11 | Pull requests 5 | Discussions | Actions | Projects | Wiki | Security | Insights

master | 3 branches | 6 tags | Go to file | Add file | Code

qkeras-robot and Copybara-Service Internal change ... 7ba2cb2 17 days ago 369 commits

README.md

QKeras

github.com/google/qkeras

Introduction

QKeras is a quantization extension to Keras that provides drop-in replacement for some of the Keras layers, especially the ones that creates parameters and activation layers, and perform arithmetic operations, so that we can quickly create a deep quantized version of Keras network.

According to Tensorflow documentation, Keras is a high-level API to build and train deep learning models. It's used for fast prototyping, advanced research, and production, with three key advantages:

- User friendly

Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.

- Modular and composable

Keras models are made by connecting configurable building blocks together, with few restrictions.

- Easy to extend

Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models.

QKeras is being designed to extend the functionality of Keras using Keras' design principle, i.e. being user friendly, modular and extensible, adding to it being "minimally intrusive" of Keras native functionality.

In order to successfully quantize a model, users need to replace variable creating layers (Dense, Conv2D, etc) by their counterparts (QDense, QConv2D, etc), and any layers that perform math operations need to be quantized afterwards.

About

QKeras: a quantization deep learning library for Tensorflow Keras

machine-learning, fpga, deep-learning, tensorflow, accelerator, keras, quantization, hardware-acceleration, fpga-accelerator, quantized-neural-networks, asic-design, quantized-networks

Readme

Apache-2.0 License

Releases 6

QKeras 0.9.0 Latest on Feb 19

+ 5 releases

Packages

No packages published
Publish your first package

Used by 1

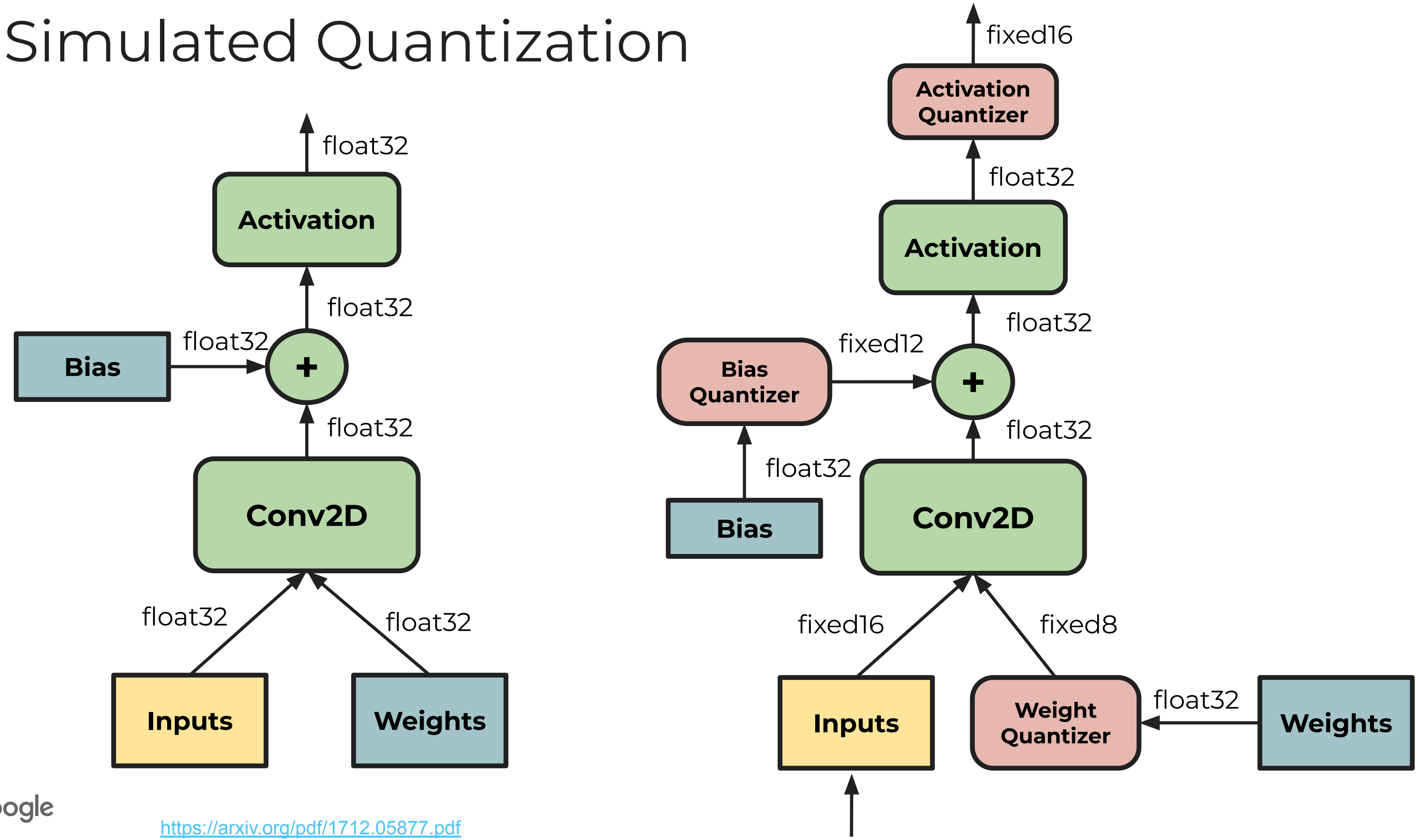
@Ali-Homsi / githubrepo

Contributors 16

Google

	QKeras	TFLite's Post-Training Quantization	TF M.O.T. Keras Functional API	AQT Accurate Quantized Training	LARQ
Works with Keras	✅	✅	✅	❌	✅
Quantization-Aware Training	✅	❌	✅	✅	✅
Heterogenous Quantization	✅	❌	❌	✅	✅
Any-number bit width quantization	✅	❌	❌	✅	✅
Layer-based Authoring API	✅	❌	❌	✅	✅
Variety of quantizers	Most	Few	Few	Few	Some
BN folding / Quantized BN	✅	✅	✅	❌	❌
Training-Aware Activation Quantizer Calibration	✅	❌	❌	✅	❌
Power-Of-Two Quantization	✅	❌	❌	❌	❌
RNN / LSTM quantization	✅	✅	✅	✅	❌
Built-in Energy Estimation	✅	❌	❌	❌	❌
Native QAT	❌	❌	❌	✅	❌
Direct TFLite Support	❌	✅	✅	❌	❌

Simulated Quantization



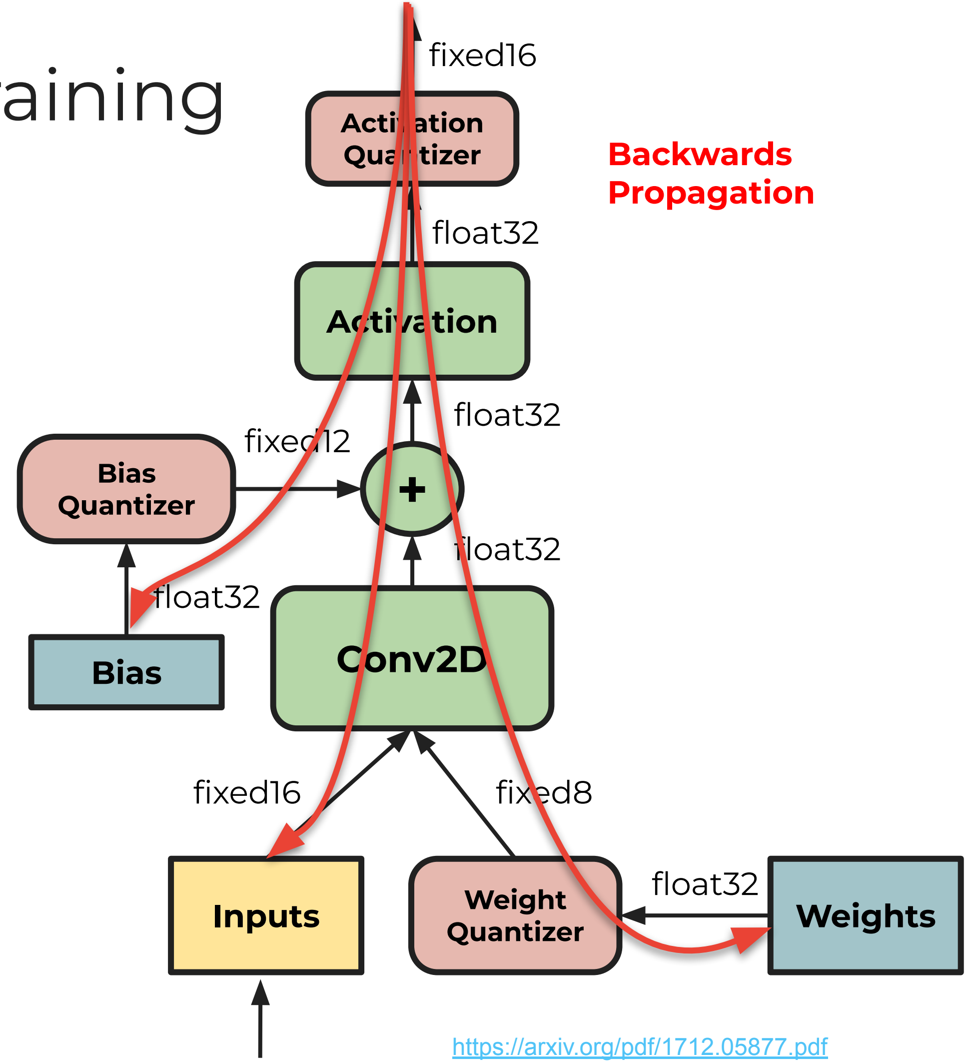
Google

https://arxiv.org/pdf/1712.05877.pdf

Quantization Aware Training

After adding the quantization functions, we need to **train the quantized model** so that the weights can adjust.

Without quantization-aware training, model accuracy is far lower.



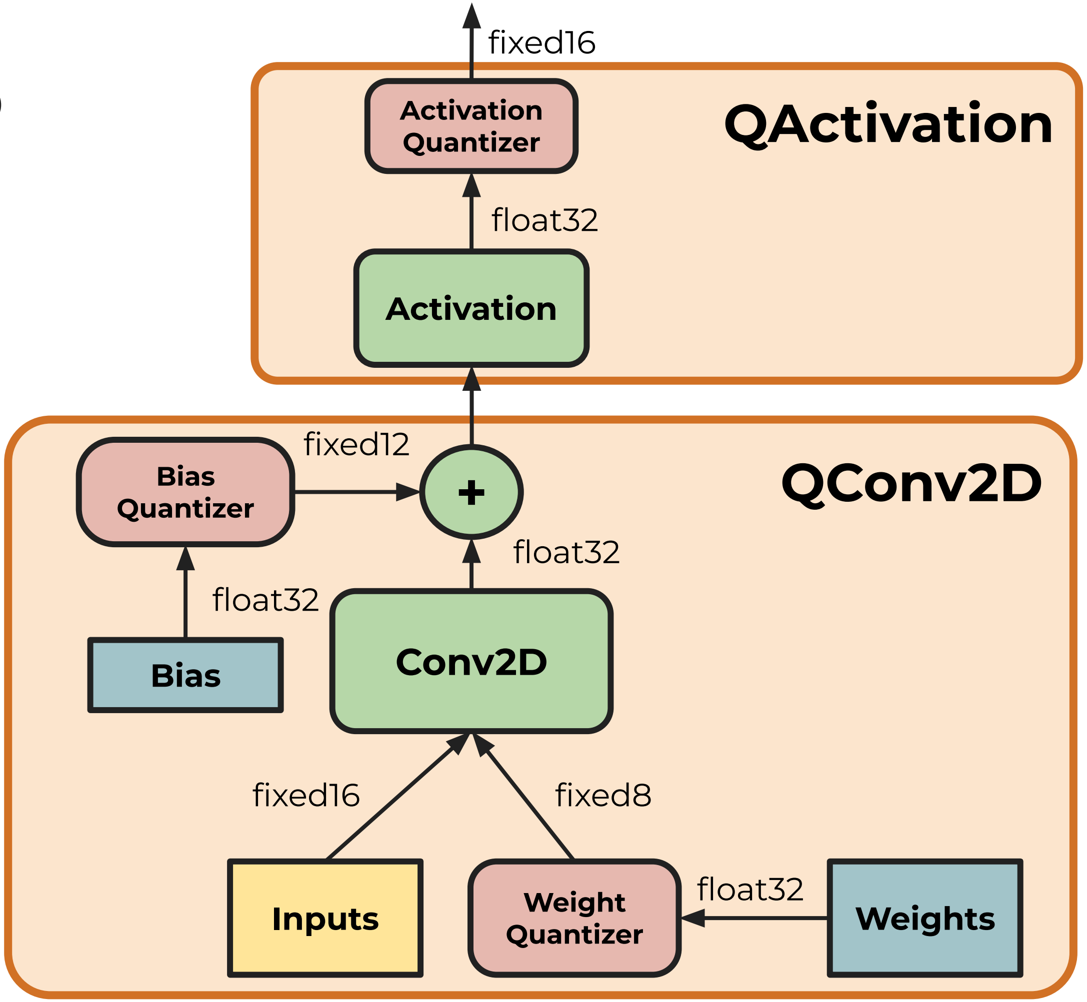
Google

https://arxiv.org/pdf/1712.05877.pdf

QKeras layers wrap Keras layers

Layers Implemented in QKeras

- QDense
- QConv1D
- QConv2D
- QDepthwiseConv2D
- QSeparableConv1D (depthwise + pointwise convolution, without quantizing the activation values after the depthwise step)
- QSeparableConv2D (depthwise + pointwise convolution, without quantizing the activation values after the depthwise step)
- QMobileNetSeparableConv2D (extended from MobileNet SeparableConv2D implementation, quantizes the activation values after the depthwise step)
- QConv2DTranspose
- QActivation
- QAdaptiveActivation
- QAveragePooling2D (in fact, an AveragePooling2D stacked with a QActivation layer for quantization of the result)
- QBatchNormalization (is still in its experimental stage, as we have not seen the need to use this yet due to the normalization and regularization effects of stochastic activation functions.)
- QOctaveConv2D
- QSimpleRNN, QSimpleRNNCell
- QLSTM, QLSTMCell
- QGRU, QGRUCell
- QBidirectional



Quantizers

Divide a continuous space into discrete bins

Need to determine:

- Scale
- Number of bins

Math demo

https://www.desmos.com/calculator/zqggxorrzg

For each layer, quantization is parameterized by the number of quantization levels and clamping range, and is performed by applying point-wise the quantization function q defined as follows:

$$
\begin{aligned}
\operatorname{clamp}(r; a, b) &:= \min\left(\max(x, a), b\right) \\
s(a, b, n) &:= \frac{b - a}{n - 1} \\
q(r; a, b, n) &:= \left\lfloor \frac{\operatorname{clamp}(r; a, b) - a}{s(a, b, n)} \right\rceil s(a, b, n) + a,
\end{aligned}
\tag{12}
$$

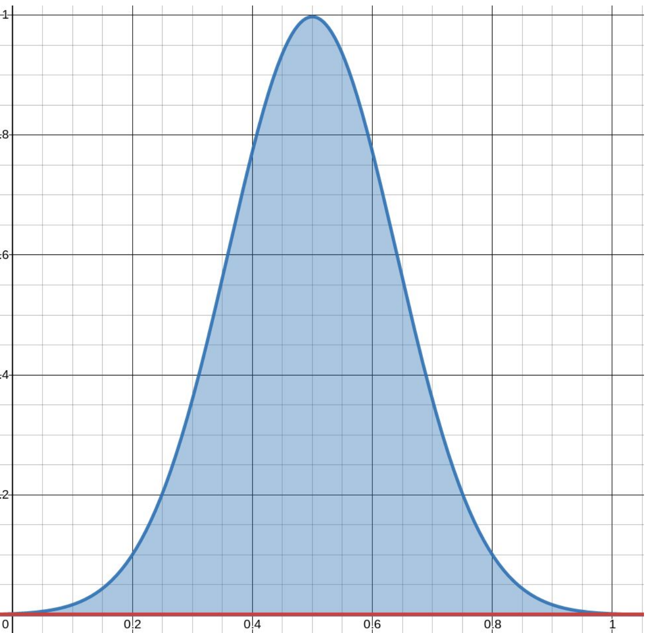
where r is a real-valued number to be quantized, $[a; b]$ is the quantization range, n is the number of quantization levels, and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. n is fixed for all layers in our experiments, e.g. $n = 2^8 = 256$ for 8 bit quantization.

https://arxiv.org/pdf/1712.05877.pdf

Quantizer Scale

No Scale

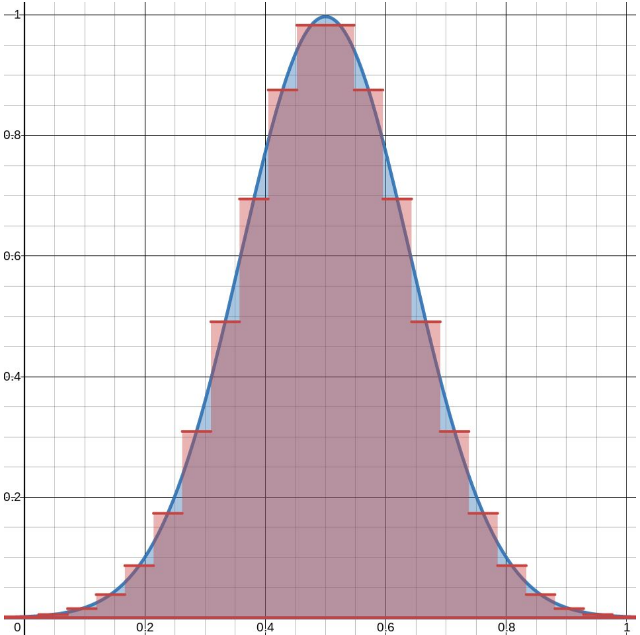
$$q(x) = \mathrm{int}(x)$$



Auto Scale

$$s = \max(x) \,/\, (n\text{-}1) = 0.047619$$

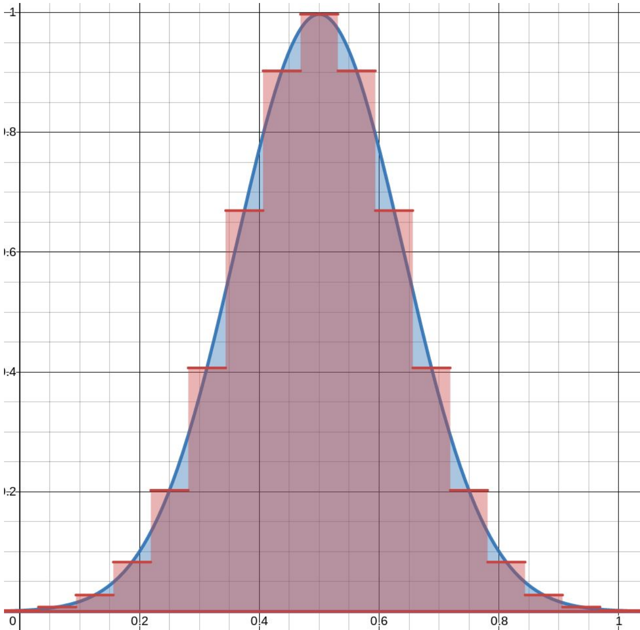
$$q(x) = \mathrm{int}(x/s) * s$$



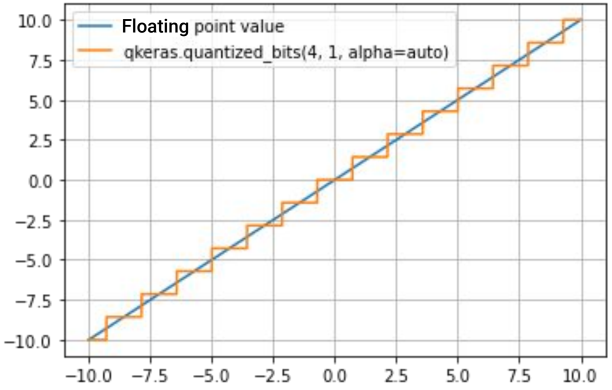
Power-of-two Scale

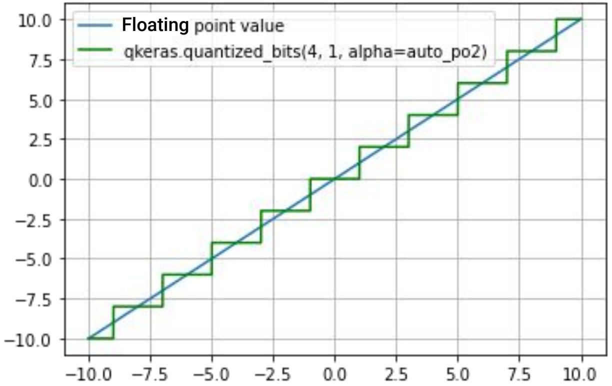
$$s = 1 \,/\, (2\text{^}\mathrm{bits}) = 0.0625$$

$$q(x) = \mathrm{int}(x >> \log2(s)) << \log2(s)$$



quantized_bits



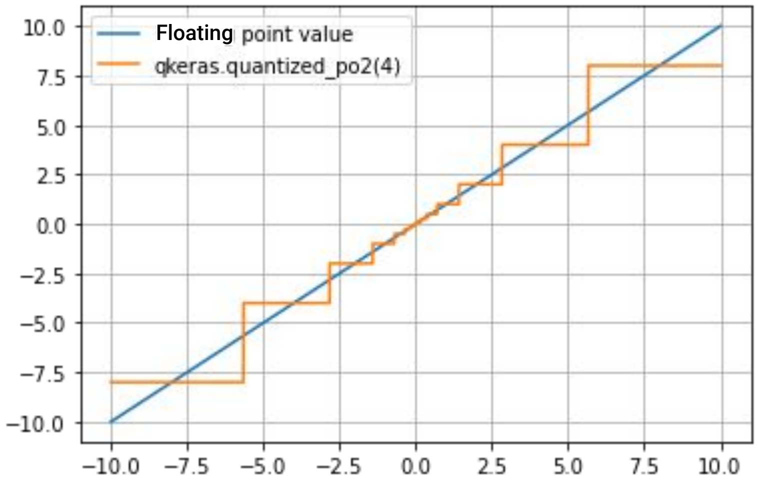



```
class quantized_bits(BaseQuantizer):  # pylint: disable=invalid-name
  """Quantizes the number to a number of bits.

  In general, we want to use a quantization function like:

  a = (pow(2,bits) - 1 - 0) / (max(x) - min(x))
  b = -min(x) * a

```

```
  Attributes:
    bits: number of bits to perform quantization.
    integer: number of bits to the left of the decimal point.
    symmetric: if true, we will have the same number of values for positive
      and negative numbers.
    alpha: a tensor or None, the scaling factor per channel.
      If None, the scaling factor is 1 for all channels.
    keep_negative: if true, we do not clip negative numbers.
    use_stochastic_rounding: if true, we perform stochastic rounding.
    scale_axis: which axis to calculate scale from
    qnoise_factor: float. a scalar from 0 to 1 that represents the level of
      quantization noise to add. This controls the amount of the quantization
      noise to add to the outputs by changing the weighted sum of
      (1 - qnoise_factor)*unquantized_x + qnoise_factor*quantized_x.
    var_name: String or None. A variable name shared between the tf.Variables
      created in the build function. If None, it is generated automatically.
    use_ste: Bool. Whether to use "straight-through estimator" (STE) method or
        not.
    use_variables: Bool. Whether to make the quantizer variables to be dynamic
      tf.Variables or not.

  Returns:
    Function that computes fixed-point quantization with bits.
  """
```

quantized_po2

Non-uniform quantization preserves both precision AND range



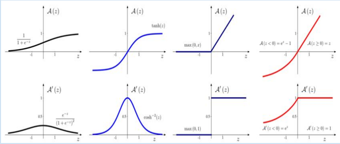

```
class quantized_po2(BaseQuantizer):  # pylint: disable=invalid-name
  """Quantizes to the closest power of 2.

  Attributes:
    bits: An integer, the bits allocated for the exponent, its sign and the sign
      of x.
    max_value: An float or None. If None, no max_value is specified.
      Otherwise, the maximum value of quantized_po2 <= max_value
    use_stochastic_rounding: A boolean, default is False, if True, it uses
      stochastic rounding and forces the mean of x to be x statstically.
    quadratic_approximation: A boolean, default is False if True, it forces the
      exponent to be even number that closted to x.
    log2_rounding: A string, log2 rounding mode. "rnd" and "floor" currently
      supported, corresponding to tf.round and tf.floor respectively.
    qnoise_factor: float. a scalar from 0 to 1 that represents the level of
      quantization noise to add. This controls the amount of the quantization
      noise to add to the outputs by changing the weighted sum of
      (1 - qnoise_factor)*unquantized_x + qnoise_factor*quantized_x.
    var_name: String or None. A variable name shared between the tf.Variables
      created in the build function. If None, it is generated automatically.
    use_ste: Bool. Whether to use "straight-through estimator" (STE) method or
        not.
    use_variables: Bool. Whether to make the quantizer variables to be dynamic
      tf.Variables or not.
  """
```

QKeras' Many Quantizers and Layers

Layer types

Dense (Fully Connected)
Conv1D, Conv2D
Conv2DTranspose
2-way Separable Conv2D
(depthwise + pointwise)
3-way Separable Conv2D
(1xN + Nx1 + pointwise)
MaxPooling, AveragePooling
GlobalAveragePooling
OctaveConv2D
SimpleRNN, LSTM, GRU
BiDirectional

Activation functions



Smooth/hard/binary sigmoid
Smooth/hard/binary tanh
ReLu
Softmax
ulaw
Hard Swish (h-swish)

Batch normalization

Separate
Folded

Quantization functions

Quantized bits
Bernoulli
Binary
Stochastic Binary
Ternary
Stochastic Ternary
Power-of-2
Quantized ReLu
Quantized ulaw
Quantized h-swish

Heterogeneous quantizer configuration

Each listed area can have an independent quantizer configuration

Per layer:
- Weights
- Biases
- Activations
- Scales

Arithmetic precision
Independent for each quantizer

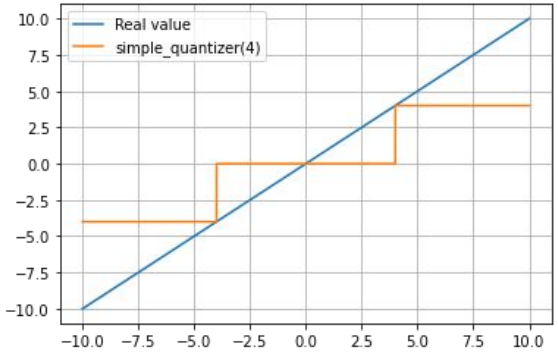
Integer 1-32 bits
Fixed point 2-32 bits

Custom Quantizers

You can make your own custom quantizers:

1. Create a class in ***quantizers.py***
2. Make the class extend ***BaseQuantizer***
3. Create an ***__init__*** method to initialize state (such as number of bits)
4. Create a ***__call__*** method that takes a floating-point inputs and returns a quantized output
5. Implement ***max*** and ***min*** methods of your quantizer
6. Implement ***from_config*** and ***get_config*** methods so that your quantizer is correctly saved and loaded when the model is saved and loaded

```
class simple_quantizer(qkeras.quantizers.BaseQuantizer):

  def __init__(self, alpha):
    self.alpha = tf.constant(alpha, dtype=tf.double)
    self.bits = 2

  def __call__(self, x):
    return tf.where(tf.logical_and(
                      tf.greater(x, -self.alpha),
                      tf.less(x, self.alpha)),
                    tf.zeros(x.shape, dtype=tf.float64),
                    tf.multiply(tf.sign(x), self.alpha))
  def max(self):
    return self.alpha
  def min(self):
    return -self.alpha
  def from_config(cls, config):
    return cls(**config)
  def get_config(self):
    return {"alpha": self.alpha}
```



Keras Example

```
x = x_in = Input((28,28,1), name="input")

x = Conv2D(filters=32, kernel_size=(2, 2), strides=(2,2))(x)

x = Activation("relu")(x)

x = Conv2D(filters=64, kernel_size=(3, 3), strides=(2,2))(x)

x = Activation("relu")(x)

x = Flatten()(x)
x = Dense(10)(x)

x = Activation("softmax")(x)
```

QKeras Heterogeneous Quantization Example

```
x = x_in = Input((28,28,1), name="input")
x = QActivation("quantized_bits(8,1)")(x)

x = QConv2D(filters=32, kernel_size=(2,2), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=8, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=12, integer=4, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=16, integer=4)")(x)

x = QConv2D(filters=64, kernel_size=(3,3), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=8, integer=2)")(x)

x = Flatten()(x)
x = QDense(units=10, kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
           bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = Activation("softmax", name="softmax")(x)
```

QKeras Heterogeneous Quantization Example

```
x = x_in = Input((28,28,1), name="input")
x = QActivation("quantized_bits(8,1)")(x)

x = QConv2D(filters=32, kernel_size=(2,2), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=8, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=12, integer=4, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=16, integer=4)")(x)

x = QConv2D(filters=64, kernel_size=(3,3), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=8, integer=2)")(x)

x = Flatten()(x)
x = QDense(units=10, kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
           bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = Activation("softmax", name="softmax")(x)
```

1. Change Keras layers to the corresponding QKeras layers

QKeras Heterogeneous Quantization Example

```
x = x_in = Input((28,28,1), name="input")
x = QActivation("quantized_bits(8,1)")(x)

x = QConv2D(filters=32, kernel_size=(2,2), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=8, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=12, integer=4, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=16, integer=4)")(x)

x = QConv2D(filters=64, kernel_size=(3,3), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=8, integer=2)")(x)

x = Flatten()(x)
x = QDense(units=10, kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
           bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = Activation("softmax", name="softmax")(x)
```

1. Change Keras layers to the corresponding QKeras layers
2. Add weight and bias quantizers

QKeras Heterogeneous Quantization Example

```
x = x_in = Input((28,28,1), name="input")
x = QActivation("quantized_bits(8,1)")(x)

x = QConv2D(filters=32, kernel_size=(2,2), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=8, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=12, integer=4, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=16, integer=4)")(x)

x = QConv2D(filters=64, kernel_size=(3,3), strides=(2,2),
            kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
            bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = QActivation("quantized_relu(bits=8, integer=2)")(x)

x = Flatten()(x)
x = QDense(units=10, kernel_quantizer=quantized_bits(
                 bits=4, alpha="auto_po2", symmetric=True, keep_negative=True),
           bias_quantizer=quantized_bits(
                 bits=6, integer=2, symmetric=True, keep_negative=True))(x)
x = Activation("softmax", name="softmax")(x)
```

1. Change Keras layers to the corresponding QKeras layers
2. Add weight and bias quantizers
3. Quantize the activations

Keras to QKeras

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 14, 14, 32)        160

 activation_1 (Activation)   (None, 14, 14, 32)        0

 conv2d_1 (Conv2D)           (None, 6, 6, 64)          18496

 activation_2 (Activation)   (None, 6, 6, 64)          0

 flatten_5 (Flatten)         (None, 2304)              0

 dense (Dense)               (None, 10)                23050

 activation_3 (Activation)   (None, 10)                0

=================================================================
Total params: 41,706
Trainable params: 41,706
Non-trainable params: 0
_________________________________________________________________
```

```
Model: "model_3"
_________________________________________________________________________________________________
 Layer (type)                                                                 Output Shape
=================================================================================================
 input (InputLayer)                                                           [(None, 28, 28, 1)]

 q_activation_16 (QActivation)                                                (None, 28, 28, 1)

 q_conv2d_6 (QConv2D)                                                         (None, 14, 14, 32)

 q_activation_17 (QActivation)                                                (None, 14, 14, 32)

 q_conv2d_7 (QConv2D)                                                         (None, 6, 6, 64)

 q_activation_18 (QActivation)                                                (None, 6, 6, 64)

 flatten_4 (Flatten)                                                          (None, 2304)

 q_dense_2 (QDense)                                                           (None, 10)

 activation (Activation)                                                      (None, 10)

=================================================================================================
Total params: 41,706
Trainable params: 41,706
Non-trainable params: 0
_________________________________________________________________________________________________

Number of operations in model:
    q_conv2d_6                    : 25088 (smult_8_8)
    q_conv2d_7                    : 663552 (smult_4_16)
    q_dense_2                     : 23040 (smult_4_8)

Number of operation types in model:
    smult_4_16                    : 663552
    smult_4_8                     : 23040
    smult_8_8                     : 25088

Weight profiling:
    q_conv2d_6_weights             : 128   (8-bit unit)
    q_conv2d_6_bias                : 32    (12-bit unit)
    q_conv2d_7_weights             : 18432 (4-bit unit)
    q_conv2d_7_bias                : 64    (6-bit unit)
    q_dense_2_weights              : 23040 (4-bit unit)
    q_dense_2_bias                 : 10    (6-bit unit)
    ----------------------------------------
    Total Bits                     : 167740

Weight sparsity:
... quantizing model
    q_conv2d_6                     : 0.0063
    q_conv2d_7                     : 0.1745
    q_dense_2                      : 0.2167
    ----------------------------------------
    Total Sparsity                 : 0.1971
```

Folded Batch Normalization

Batch normalization layers are important for modern deep neural networks

Problem: Batch normalization includes several operations that are expensive to run on optimized hardware, such as high-precision division

Solution: Fold the batch normalization operations into normal quantized convolutional layers. We can do this automatically with the function ***convert_to_folded_model***

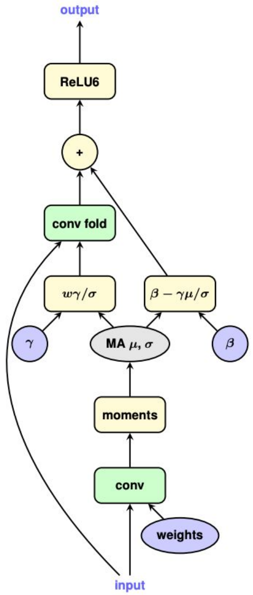


Figure C.7: Convolutional layer with batch normalization: training graph, folded

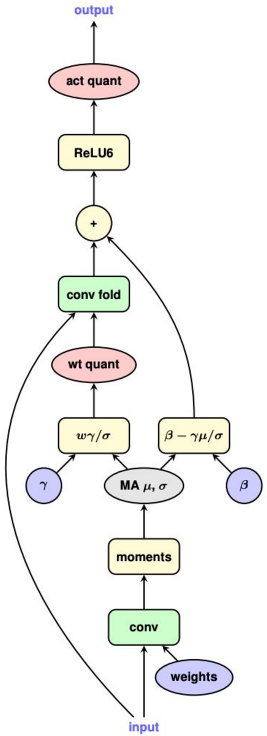


Figure C.8: Convolutional layer with batch normalization: training graph, folded and quantized

https://arxiv.org/pdf/1712.05877.pdf

Google

QAdaptiveActivation

Problem: We can't find the scale of activation quantizers because the data distribution keeps changing every batch

Solution: find the exponential moving average of the data distribution so that we always have a good scale for the activation quantizers.

```python
class QAdaptiveActivation(Layer, PrunableLayer):
  """[EXPERIMENTAL] Implements an adaptive quantized activation layer using EMA.

  This layer calculates an exponential moving average of min and max of the
  activation values to automatically determine the scale (integer bits) of
  the quantizer used in this layer.
  """

  def __init__(self,
               activation,
               total_bits,
               current_step=None,
               symmetric=True,
               quantization_delay=0,
               ema_freeze_delay=None,
               ema_decay=0.9999,
               per_channel=False,
               po2_rounding=False,
               relu_neg_slope=0.0,
               relu_upper_bound=None,
               **kwargs):
    """Initializes this QAdaptiveActivation layer.

    Args:
      activation: Str. The activation quantizer type to use for this activation
        layer, such as 'quantized_relu'. Should be a string with no params.
      total_bits: Int. The total bits that can be used by the quantizer
      current_step: tf.Variable specifying the current step in training.
        You can find this by passing model.optimizer.iterations
        (see tf.keras.optimizers.Optimizer.iterations). If set to None, the
        layer will attempt to estimate the current step itself, but please note
        that this number may not always match the optimizer step.
      symmetric: Bool. If to enforce symmetry about the origin in the quantized
        bit representation of the value. When using linear activation, this
        should be True for best results.
      quantization_delay: Int. How many training steps to wait until quantizing
        the activation values.
      ema_freeze_delay: Int. Steps to wait until stopping the update of the
        exponential moving average values. Set to None for an infinite delay.
      ema_decay: Float. The decay value used for exponential moving average (see
        tf.keras.backend.moving_average_update)
```

QTools

Purpose:

- To assist hardware implementation of the quantized model and model size estimation;
- Automatically generate the data type map for weights, bias, multiplier, adder, etc. of each layer.
- Data type map includes operation type, variable size, quantizer type and bits, etc.

Input: a given quantized model; a list of input quantizers for the model

Output: json file that list the data type map of each layer (stored in qtools_instance._output_dict)

QTools Example

```python
x = x_in = keras.layers.Input((784,), name="input")
x = keras.layers.Dense(300, name="d0")(x)
x = keras.layers.Activation("relu", name="d0_act")(x)
x = QDense(100, kernel_quantizer=quantizers.quantized_po2(4),
           bias_quantizer=quantizers.quantized_po2(4),
           name="d1")(x)
x = QActivation("quantized_relu(4,0)", name="d1_qr4")(x)
x = QDense(
    10, kernel_quantizer=quantizers.quantized_po2(4),
    bias_quantizer=quantizers.quantized_po2(4),
    name="d2")(x)
x = keras.layers.Activation("softmax", name="softmax")(x)

return keras.Model(inputs=[x_in], outputs=[x])
```

```python
input_quantizer_list = [quantizers.quantized_bits(8, 0, 1)]
reference_internal = "int8"
reference_accumulator = "int32"

# generate QTools object which contains model data type map in json format
q = run_qtools.QTools(
    in_model,
    # energy calculation using a given process
    process="horowitz",
    # quantizers for model inputs
    source_quantizers=input_quantizer_list,
    # training or inference with a pre-trained model
    is_inference=False,
    # path to pre-trained model weights
    weights_path=None,
    # keras_quantizer to quantize weight/bias in non-quantized keras layers
    keras_quantizer=reference_internal,
    # keras_accumulator to quantize MAC in un-quantized keras layers
    keras_accumulator=reference_accumulator,
    # calculating baseline energy or not
    for_reference=False)

# print data type map
q.qtools_stats_print()
```

```
{
    "source_quantizers": [
        {
            "quantizer_type": "quantized_bits",
            "bits": 8,
            "int_bits": 0,
            "is_signed": true
        }
    ],
    "d0": {
        "layer_type": "Dense",
        "input_quantizer_list": [
            {
                "quantizer_type": "quantized_bits",
                "bits": 8,
                "int_bits": 0,
                "is_signed": true
            }
        ],
        "weight_quantizer": {
            "quantizer_type": "quantized_bits",
            "bits": 8,
            "int_bits": 0,
            "is_signed": true,
            "shape": [
                784,
                300
            ]
        },
        "bias_quantizer": {
            "quantizer_type": "quantized_bits",
            "bits": 8,
            "int_bits": 0,
            "is_signed": true,
            "shape": 300
        },
        "multiplier": {
            "quantizer_type": "quantized_bits",
            "bits": 32,
            "int_bits": 10,
            "is_signed": true,
            "op_type": "mul"
        },
        "accumulator": {
            "quantizer_type": "quantized_bits",
            "bits": 32,
```

```
        "accumulator": {
            "quantizer_type": "quantized_bits",
            "bits": 32,
            "int_bits": 10,
            "is_signed": true,
            "op_type": "add"
        },
        "output_quantizer": {
            "quantizer_type": "quantized_bits",
            "bits": 32,
            "int_bits": 10,
            "is_signed": true,
            "shape": [
                -1,
                300
            ]
        },
        "operation_count": 235200
    },
    "d0_act": {
        "layer_type": "Activation",
        "input_quantizer_list": [
            {
                "quantizer_type": "quantized_bits",
                "bits": 32,
                "int_bits": 10,
                "is_signed": true
            }
        ],
        "output_quantizer": {
            "quantizer_type": "quantized_bits",
            "bits": 8,
            "int_bits": 0,
            "is_signed": true,
            "shape": [
                -1,
                300
            ]
        },
        "operation_count": 300
    },
    "d1": {
        "layer_type": "QDense",
```

```
"d2": {
    "layer_type": "QDense",
    "input_quantizer_list": [
        {
            "quantizer_type": "quantized_relu",
            "bits": 4,
            "int_bits": 0,
            "is_signed": 0
        }
    ],
    "weight_quantizer": {
        "quantizer_type": "quantized_po2",
        "bits": 4,
        "is_signed": 1,
        "max_value": -1,
        "shape": [
            100,
            10
        ]
    },
    "bias_quantizer": {
        "quantizer_type": "quantized_po2",
        "bits": 4,
        "is_signed": 1,
        "max_value": -1,
        "shape": 10
    },
    "multiplier": {
        "quantizer_type": "quantized_bits",
        "bits": 12,
        "int_bits": 3,
        "is_signed": 1,
        "op_type": "shifter"
    },
    "accumulator": {
        "quantizer_type": "quantized_bits",
        "bits": 19,
        "int_bits": 10,
        "is_signed": 1,
        "op_type": "add"
    },
    "output_quantizer": {
        "quantizer_type": "quantized_bits",
        "bits": 19,
        "int_bits": 10,
```

QEnergy - Computing Energy Estimate

energy_estimate(layer) = energy(input) + energy(parameter) + energy(op) + energy(output)

actual_energy(layer) = k1 * energy_estimate(layer) + k2

$$energy(bits) = a\ bits^2 + b\ bits + c$$

	a	b	c	
fixed_point_add		0.0031	0	pJ/bit
fixed_point_multiply	0.0030	0.0010	0	pJ/bit
FP16 add			0.4	pJ/bit
FP16 multiply			1.1	pJ/bit
FP32 add			0.9	pJ/bit
FP32 multiply			3.7	pJ/bit
SRAM access	0.02455/64	-0.2656/64	0.8661/64	pJ/bit
DRAM access		20.3125	0	pJ/bit

		inputs (x)					
		fp	m-quant	e-quant	-1,0,+1	-1,+1	0,1
weights (w)	fp	fp	fp	fp	?/+-(fp)	?/+-(fp)	&(fp)
	m-quant	fp	*(m)/+(m')	<<>>/+(m')	?/+-(m')	?/+-(m')	&
	e-quant	fp	<<>>/+(m')	+/+(m')	?/+-(m')	?/+-(m')	&/+(m')
	-1,0,+1	?/+-(fp)	?/+-(m')	?/+-(m')	?/+-1	?/+-1	&/+-1
	-1,+1	?/+-(fp)	?/+-(m')	?/+-(m')	?/+-1	^/+-1	&/+-1
	0,1	&(fp)	&	&/+(m')	&/+-1	&/+-1	&/+1

energy(*, bits) = fixed_point_multiply(bits)
energy(+, bits) = fixed_point_add(bits)
energy(?, bits) = alpha(?) * fixed_point_add(bits)

M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, San Francisco, CA, 2014, pp. 10-14.

Design Space Options

All of these parameters influence the tradeoff between accuracy and cost

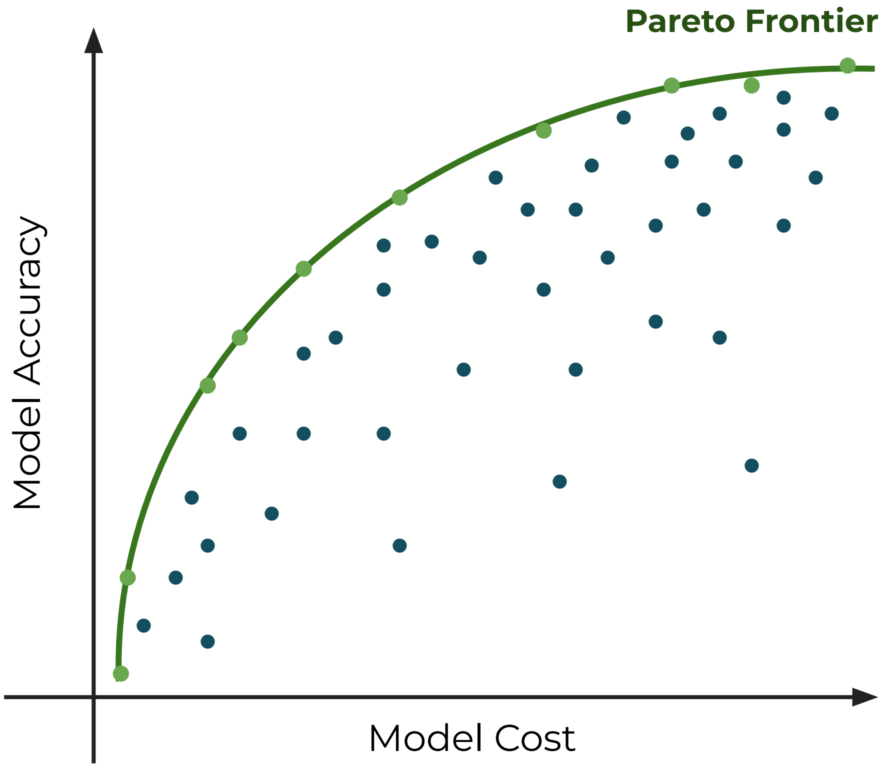
1. **Number of bits** / quantization levels for each weight / bias / activation quantizer
2. **Scale** for each weight / bias / activation quantizer
3. **Quantization function** for every quantizer
4. Number of **channels** for every layer
5. Number of **layers**
6. **Connections** between the layers
7. **Layer types** (filter sizes, strides, dense versus conv, etc)

Finding the Pareto Frontier

To find the right **tradeoff between model accuracy and cost**, we can build a Pareto Frontier of optimal models.

We can then use the model with **highest possible accuracy for any cost budget**.

The **model cost** can be a theoretical energy number, and/or a **trained cost model** that predicts power & latency on a target HW platform.



How do we find the Pareto Frontier?

Google Vizier: A Service for Black-Box Optimization

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, D. Sculley
{dgg, bsolnik, smoitra, gpk, karro, dsculley}@google.com
Google Research
Pittsburgh, PA, USA

ABSTRACT

Any sufficiently complex system acts as a black box when it becomes easier to experiment with than to understand. Hence, black-box optimization has become increasingly important as systems have become more complex. In this paper we describe *Google Vizier*, a Google-internal service for performing black-box optimization that has become the de facto parameter tuning engine at Google. Google Vizier is used to optimize many of our machine learning models and other systems, and also provides core capabilities to Google's Cloud Machine Learning *HyperTune* subsystem. We discuss our requirements, infrastructure design, underlying algorithms, and advanced features such as transfer learning and automated early stopping that the service provides.

KEYWORDS

Black-Box Optimization, Bayesian Optimization, Gaussian Processes, Hyperparameters, Transfer Learning, Automated Stopping

1 INTRODUCTION

In this paper we discuss a state-of-the-art system for black–box optimization developed within Google, called *Google Vizier*, named after a high official who offers advice to rulers. It is a service for black-box optimization that supports several advanced algorithms. The system has a convenient Remote Procedure Call (RPC) interface, along with a dashboard and analysis tools. Google Vizier is a research project, parts of which supply core capabilities to our Cloud Machine Learning *HyperTune*[1] subsystem. We discuss the architecture of the system, design choices, and some of the algorithms used.

1.1 Related Work

Black–box optimization makes minimal assumptions about the problem under consideration, and thus is broadly applicable across many domains and has been studied in multiple scholarly fields under names including Bayesian Optimization [2, 25, 26], Derivative–free optimization [7, 24], Sequential Experimental Design [5], and assorted variants of the multiarmed bandit problem [13, 20, 29].

Several classes of algorithms have been proposed for the problem. The simplest of these are non-adaptive procedures

How Vizier Works

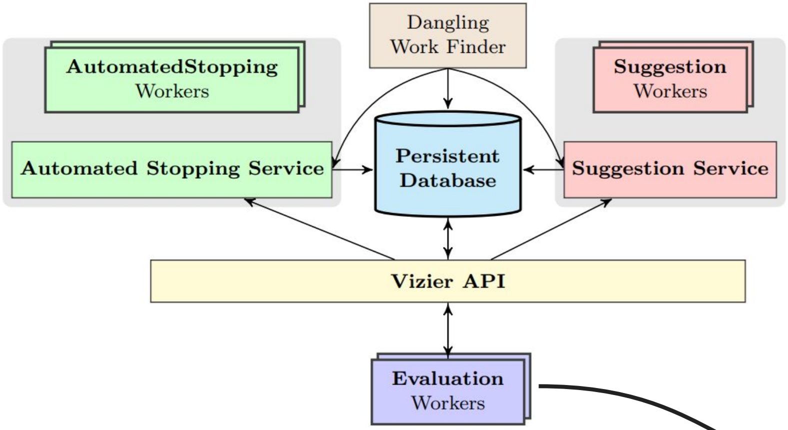


Figure 1: Architecture of Vizier service: Main components are (1) Dangling work finder (restarts work lost to preemptions) (2) Persistent Database holding the current state of all Studies (3) Suggestion Service (creates new Trials), (4) Early Stopping Service (helps terminate a Trial early) (5) Vizier API (JSON, validation, multiplexing) (6) Evaluation workers (provided and owned by the user).

Dynamically create workers to search many configurations in parallel

Worker 1

Trains a model with quantization schema 1

Worker 2

Trains a model with quantization schema 2

Worker 3

Trains a model with quantization schema 3

Worker 4

Trains a model with quantization schema

Optimizing multiple objectives

Send feedback

Run this tutorial as a notebook in Colab

View the notebook on GitHub

This tutorial demonstrates AI Platform Optimizer multi-objective optimization.

Objective

The goal is to `minimize` the objective metric: `y1 = r*sin(theta)`

and simultaneously `maximize` the objective metric: `y2 = r*cos(theta)`

that you will evaluate over the parameter space:

- `r` in [0,1],
- `theta` in [0, pi/2]

AI Platform Vizier

AI Platform Vizier documentation

Product overview

All AI Platform documentation

Getting started

Introduction to AI Platform

Vizier overview

Getting started: Optimizing a machine learning model

Getting started: Optimizing multiple objectives

Using AI Platform Vizier

Monitoring and security

Viewing audit logs

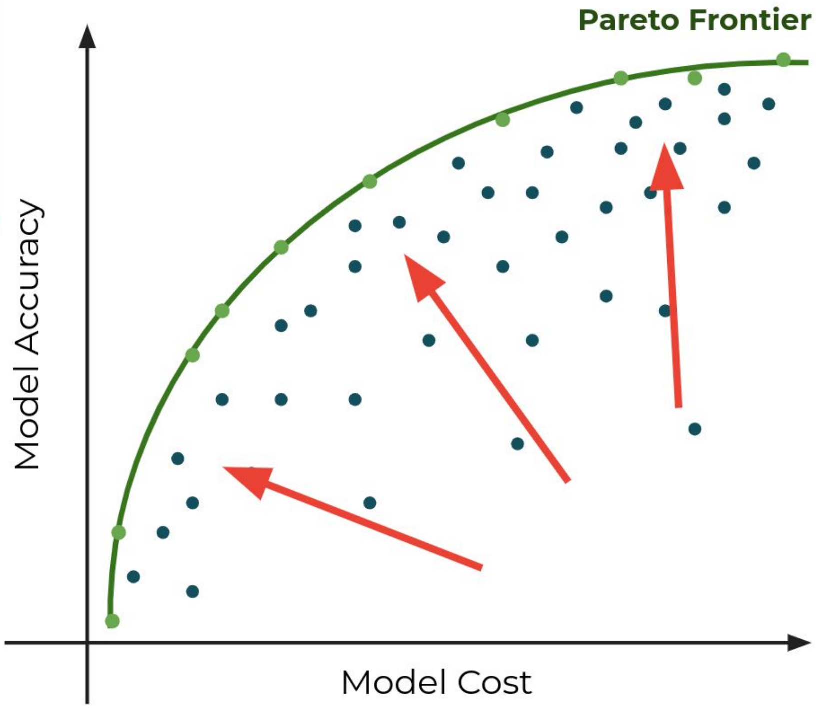
Access control

AI Platform > AI Platform Vizier > Documentation

Was this helpful?

Optimizing multiple objectives

Send feedback



Goo

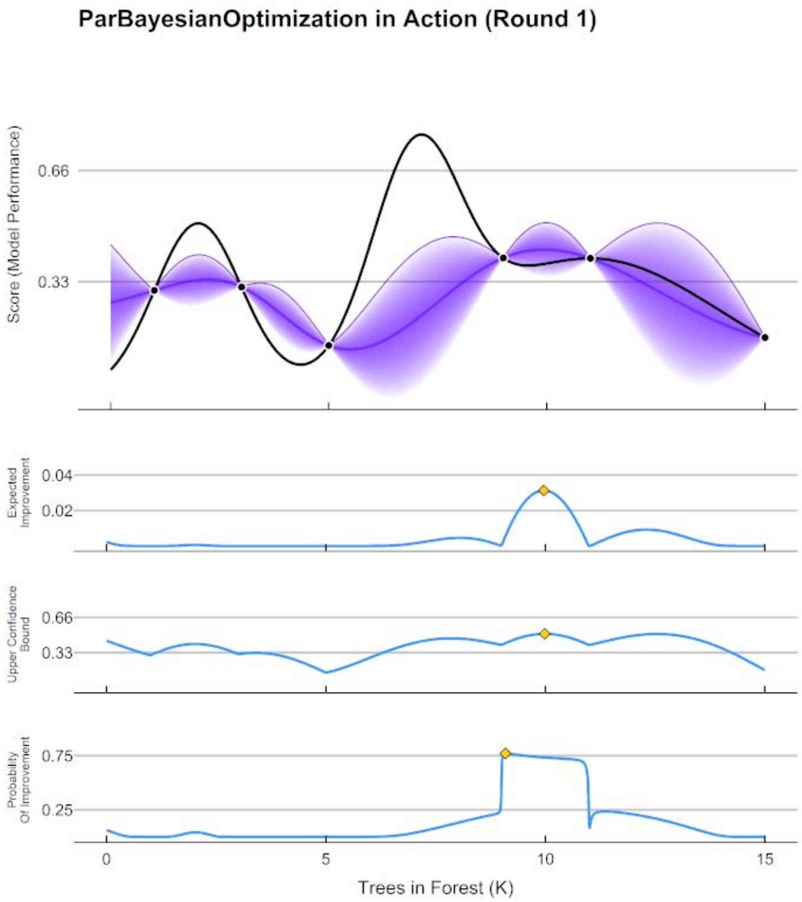
Vizier Search Algorithms

Search algorithms

If you do not specify an algorithm, Vertex AI Vizier uses the default algorithm. The default algorithm applies Bayesian optimization to arrive at the optimal solution with a more effective search over the parameter space.

The following values are available:

- `ALGORITHM_UNSPECIFIED`: Results in the same behavior as when you don't specify a search algorithm. Vertex AI Vizier uses a default algorithm, which applies Bayesian optimization to search the space of possible values, resulting in the most effective technique for your set of parameters.
- `GRID_SEARCH`: A simple grid search within the feasible space. This option is useful if you want to specify a quantity of trials that is greater than the number of points in the feasible space. In such cases, if you do not specify a grid search, the default algorithm can generate duplicate suggestions. To use grid search, all parameters must be of type `INTEGER`, `CATEGORICAL`, or `DISCRETE`.
- `RANDOM_SEARCH`: A simple random search within the feasible space.

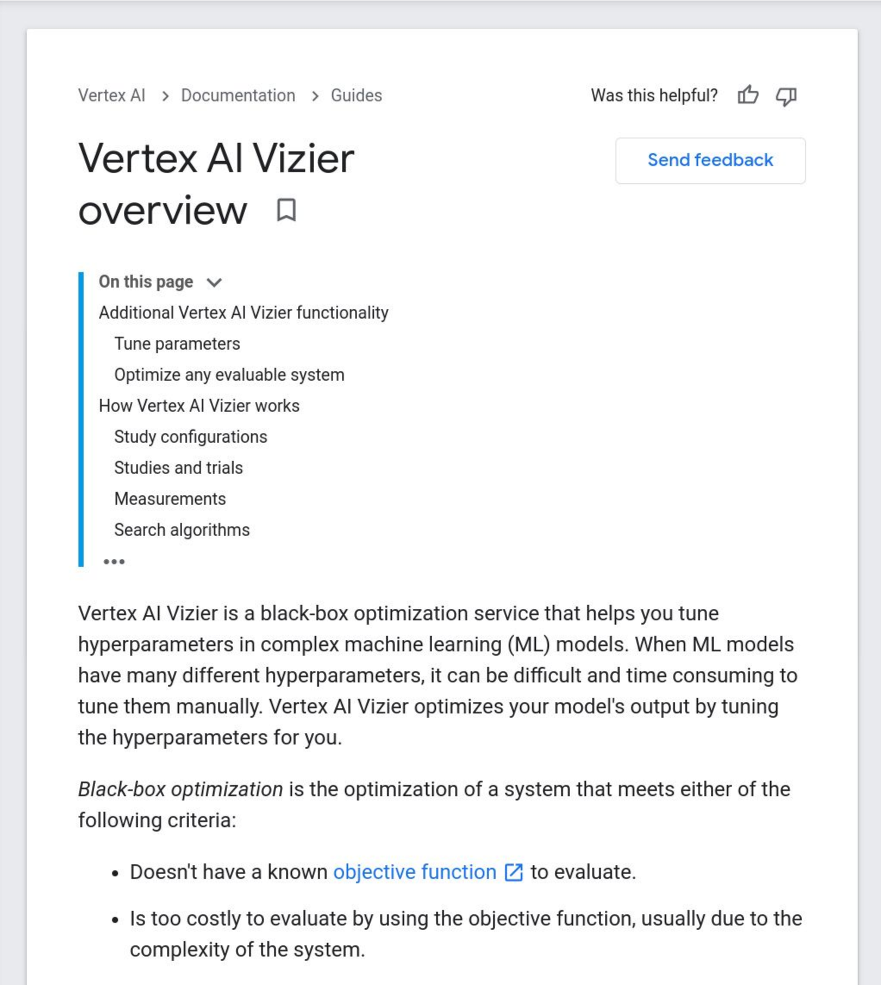


https://en.wikipedia.org/wiki/Bayesian_optimization

How to use Vizier

1. With Google Cloud
   - Called “***Vertex AI Vizier***”
   - Free tier available

2. Upcoming open source version
   - GitHub Link: github.com/google/vizier
   - Supports custom search algorithms



Vertex AI > Documentation > Guides

Was this helpful?

Send feedback

Vertex AI Vizier overview

On this page

Vertex AI Vizier is a black-box optimization service that helps you tune hyperparameters in complex machine learning (ML) models. When ML models have many different hyperparameters, it can be difficult and time consuming to tune them manually. Vertex AI Vizier optimizes your model's output by tuning the hyperparameters for you.

Black-box optimization is the optimization of a system that meets either of the following criteria:

- Doesn't have a known objective function to evaluate.
- Is too costly to evaluate by using the objective function, usually due to the complexity of the system.

Creating a Vizier Study

```
param_layer1_weightquantizer_bits = {
    'parameter_id': layer1_weightquantizer_bits,
    'integer_value_spec': {
        'min_value': 4,
        'max_value': 12
    }
}

metric_accuracy = {
    'metric_id': 'accuracy',
    'goal': 'MAXIMIZE'
}

metric_cost = {
    'metric_id': ‘cost',
    'goal': 'MINIMIZE'
}

study = {
    'display_name': STUDY_DISPLAY_NAME,
    'study_spec': {
      'parameters': [param_layer1_weightquantizer_bits,
                     param_layer2_weightquantizer_bits,…],
      'metrics': [metric_accuracy, metric_cost],
    }
}
```

```
vizier_client = aiplatform.gapic.VizierServiceClient(

  client_options=dict(api_endpoint=ENDPOINT)),

  study=vizier_client.create_study(parent=PARENT,
                                   study=study)

STUDY_NAME = study.name
```

Google

Running a Vizier Study

```
# EVERY WORKER RUNS THIS CODE

# Get a suggested set of parameters from the Vizier search
# algorithm
suggest_response = vizier_client.suggest_trials({
    'parent': STUDY_NAME,
    'suggestion_count': 1,
    'client_id': CLIENT_ID
    })


# Current model configuration
trial_config = suggest_response.result().trials[0]

# Create the QKeras model and run the training
qkeras_model = create_qkeras_model(trial_config)
qkeras_model.train()
experiment_results = qkeras_model.evaluate()

# Report the measurements
vizier_client.add_trial_measurement({
        'trial_name': TRIAL_ID,
        'measurement': {
            'metrics': [**experiment_results]
        }
    })
```

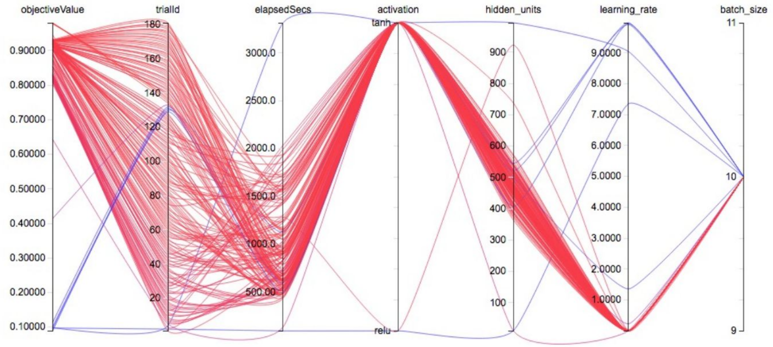
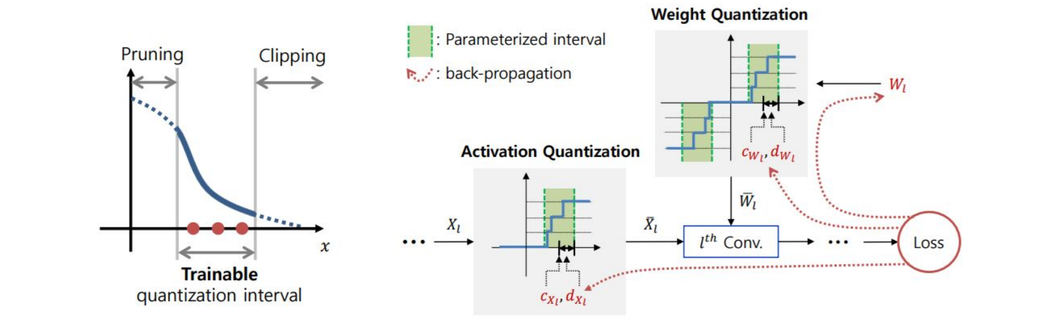


Figure 4: The Parallel Coordinates visualization [18] is used for examining results from different Vizier runs. It has the benefit of scaling to high dimensional spaces (∼15 dimensions) and works with both numerical and categorical parameters. Additionally, it is interactive and allows various modes of slicing and dicing data.

Learnable Quantizers

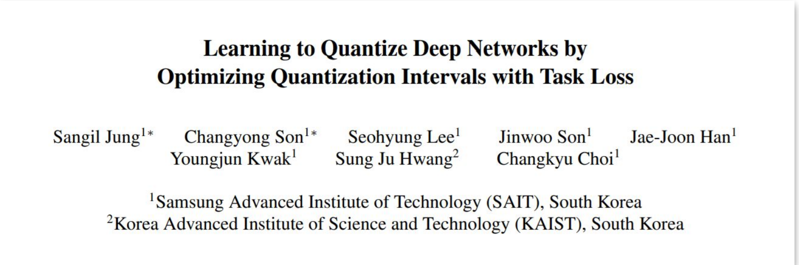
What if we could get the model to **learn** how to assign bits while training?

We have implemented this in QKeras as an experimental quantizer with promising results



(a) Quantization Interval (b) A convolutional layer of our low bit-width network

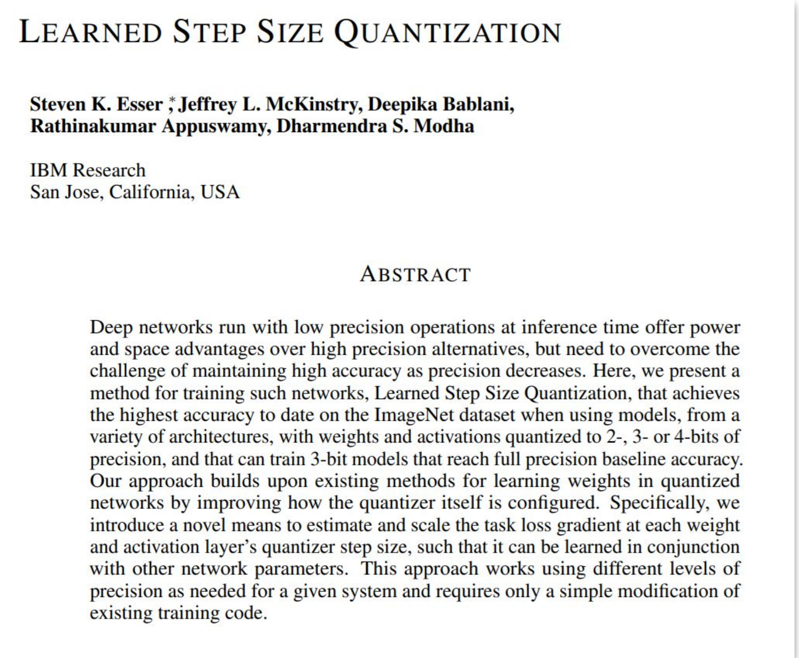
Figure 1. Illustration of our trainable quantizer. (a) Our trainable quantization interval, which performs pruning and clipping simultaneously. (b) The l^{th} convolution layer of our low-precision network. Given bit-width, the quantized weights $\bar{W}_l$ and activations $\bar{X}_l$ are acquired using the parameterized intervals. The interval parameters $(c_{W_l}, d_{W_l}, c_{X_l}, d_{X_l})$ are trained jointly with the full-precision weights W_l during backpropagation.



Learning to Quantize Deep Networks by Optimizing Quantization Intervals with Task Loss

Sangil Jung[1*] Changyong Son[1*] Seohyung Lee[1] Jinwoo Son[1] Jae-Joon Han[1]
Youngjun Kwak[1] Sung Ju Hwang[2] Changkyu Choi[1]

[1]Samsung Advanced Institute of Technology (SAIT), South Korea
[2]Korea Advanced Institute of Science and Technology (KAIST), South Korea



LEARNED STEP SIZE QUANTIZATION

Steven K. Esser *, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, Dharmendra S. Modha

IBM Research
San Jose, California, USA

ABSTRACT

Deep networks run with low precision operations at inference time offer power and space advantages over high precision alternatives, but need to overcome the challenge of maintaining high accuracy as precision decreases. Here, we present a method for training such networks, Learned Step Size Quantization, that achieves the highest accuracy to date on the ImageNet dataset when using models, from a variety of architectures, with weights and activations quantized to 2-, 3- or 4-bits of precision, and that can train 3-bit models that reach full precision baseline accuracy. Our approach builds upon existing methods for learning weights in quantized networks by improving how the quantizer itself is configured. Specifically, we introduce a novel means to estimate and scale the task loss gradient at each weight and activation layer's quantizer step size, such that it can be learned in conjunction with other network parameters. This approach works using different levels of precision as needed for a given system and requires only a simple modification of existing training code.

Google
https://arxiv.org/pdf/1808.05779.pdf

nature machine intelligence

Explore content ⌄ About the journal ⌄ Publish with us ⌄ Subscribe

nature › nature machine intelligence › articles › article

Article | Published: 21 June 2021

Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors

Claudionor N. Coelho Jr, Aki Kuusela, Shan Li, Hao Zhuang, Jennifer Ngadiuba, Thea Klaeboe Aarrestad, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol & Sioni Summers

Nature Machine Intelligence **3**, 675–686 (2021) | Cite this article

Google

The task is *"discrimination of jets, a collimated spray of particles, stemming from the decay and/or hadronization of five different particles. [We must detect the presence of a] quark (q), gluon (g), W boson, Z boson, and top (t) jets, each represented by 16 physics-motivated high-level features"*

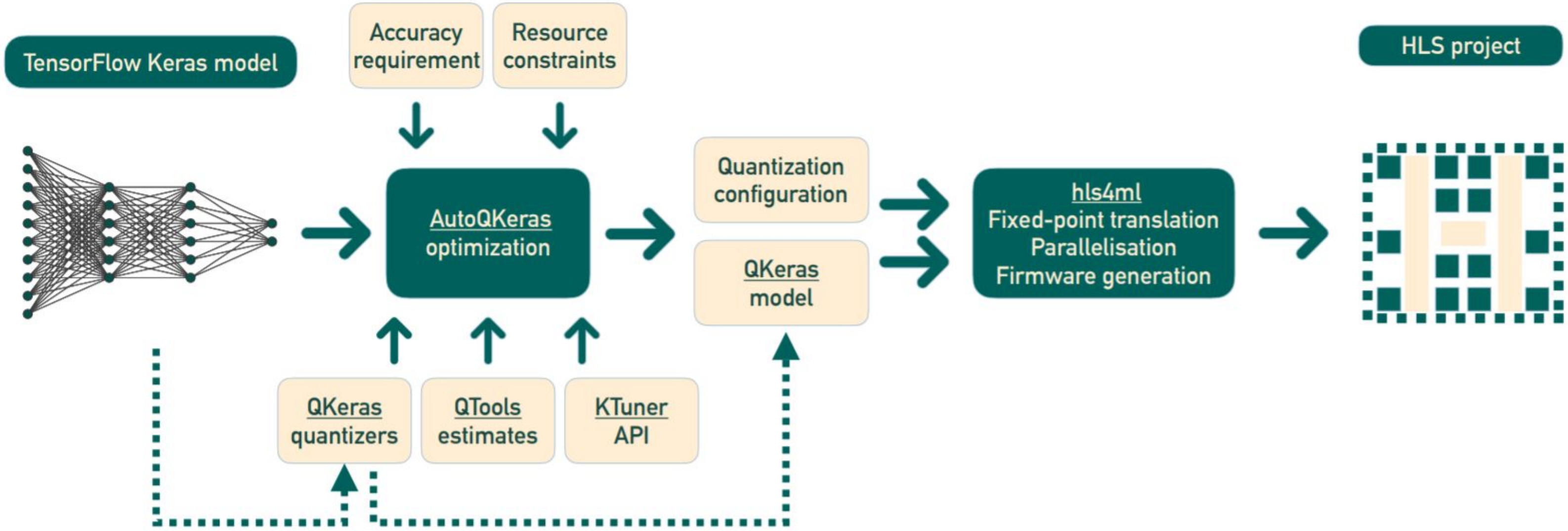


FIG. V. The full workflow starting from a baseline TensorFlow Keras Model, which is then converted into an optimally quantized equivalent through QKeras and AutoQKeras. This model is then translated into highly parallel firmware with hls4ml.

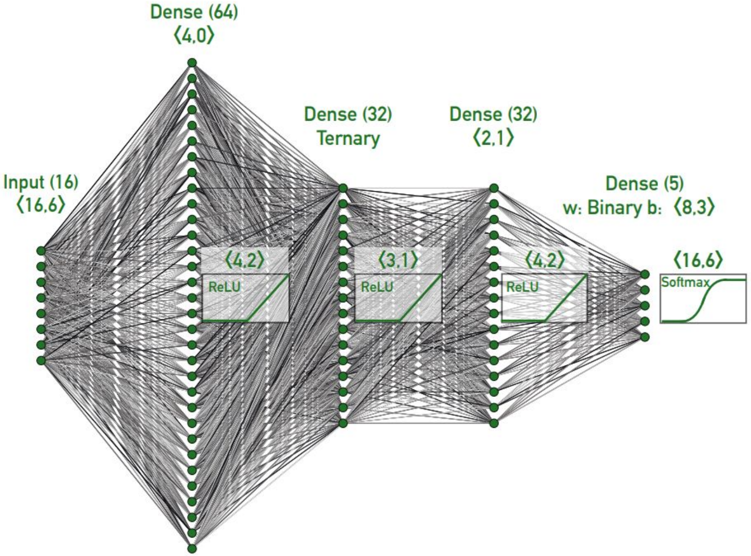


TABLE II. Per-layer quantization configuration and the relative model energy consumption for the AutoQKeras Energy Optimized (QE) and AutoQKeras Bits Optimized (QB) models, compared to the simple homogeneously quantized model, Q6.

Model	Acc. [%]	Precision								$\frac{E}{E_{Q6}}$	$\frac{Bits}{Bits_{Q6}}$
		Dense	ReLU	Dense	ReLU	Dense	ReLU	Dense	Softmax		
QE	72.3	$\langle 4, 0\rangle$	$\langle 4, 2\rangle$	Ternary	$\langle 3, 1\rangle$	$\langle 2, 1\rangle$	$\langle 4, 2\rangle$	w: Stoc. Bin. b: $\langle 8, 3\rangle$	$\langle 16, 6\rangle$	0.27	0.18
QB	72.8	$\langle 4, 0\rangle$	$\langle 4, 2\rangle$	Stoc. Bin.	$\langle 4, 2\rangle$	Ternary	$\langle 3, 1\rangle$	Stoc. Bin.	$\langle 16, 6\rangle$	0.25	0.17
Q6	74.8	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	$\langle 6, 0\rangle$	1.00	1.00

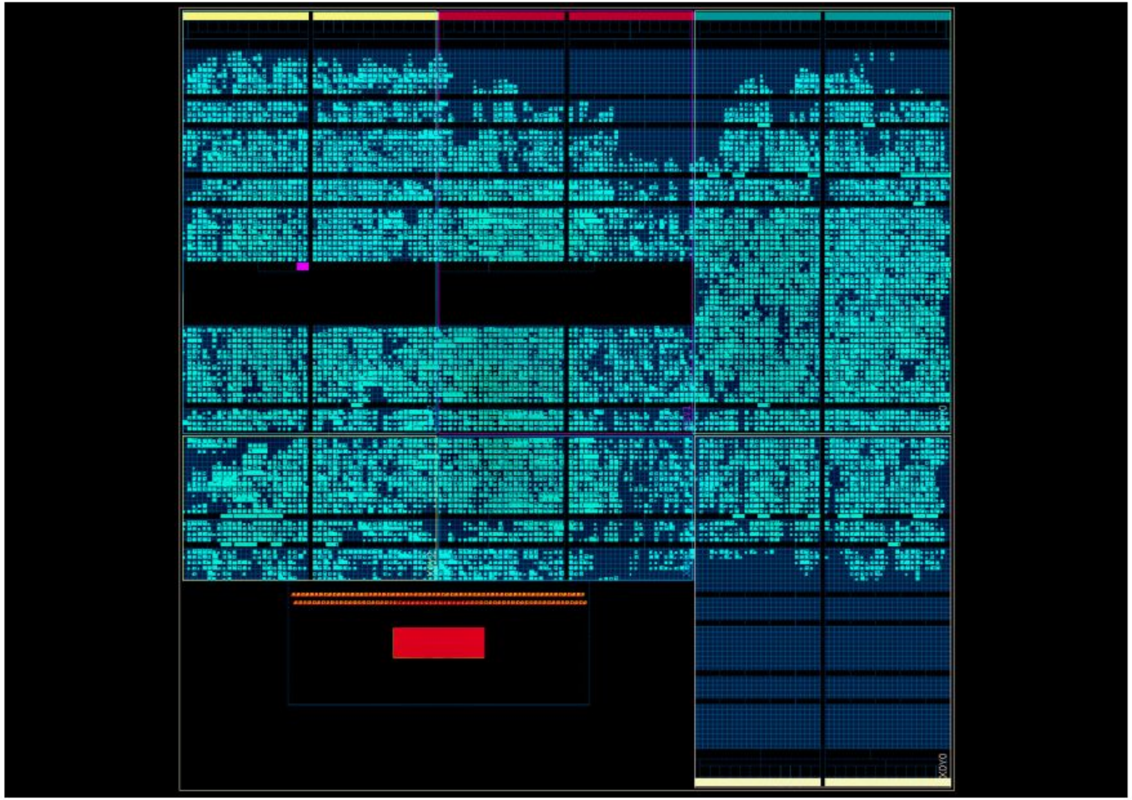


FIG. I. An ultra-compressed deep neural network for particle identification on a Xilinx FPGA.

Google

Thank you!

Any Questions?

Daniele Moro - danielemoro@google.com

Google

TINY ML SUMMIT

tinyML Summit 2022 Sponsors

- ANALOG DEVICES – AHEAD OF WHAT'S POSSIBLE™
- AONdevices
- arm
- ASPINITY
- brainchip – The Neuromorphic Computing Company
- CEVA®
- Deeplite
- EDGE IMPULSE
- emza visual sense
- FotaHub
- GREENWAVES TECHNOLOGIES
- Grovety Inc.
- Himax
- HOTG
- imagimob
- Infineon
- itemis
- KLIKA TECH – GLOBAL IOT SOLUTIONS
- LatentAI
- LATTICE SEMICONDUCTOR
- Micro.ai
- OmniML
- NXP
- PXI
- Plumerai
- PROPHESEE
- Qeexo
- Qualcomm
- Rackner
- Reality AI® – Engineering Solutions for the Edge
- REEXEN technology
- RENESAS
- SAP
- seeed studio – The IoT Hardware Enabler
- SensiML™
- Sony Semiconductor Solutions Corporation
- ST – life.augmented
- STREAM ANALYZE
- synaptics
- SynSense
- SYNTIANT
- Tensil.ai
- TensorFlow
- XMOS

Copyright Notice