
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.

Publications

- Claudionor N. Coelho Jr, Aki Kuusela, Shan Li, Hao Zhuang, Jennifer Ngadiuba, Thea Klæboe Aarrestad, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors”, Nature Machine Intelligence (2021), <https://www.nature.com/articles/s42256-021-00356-5>

-
- Claudionor N. Coelho Jr., Aki Kuusela, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Sioni Summers, “Ultra Low-latency, Low-area Inference Accelerators using Heterogeneous Deep Quantization with QKeras and hls4ml”, <http://arxiv.org/abs/2006.10159v1>
 - Erwei Wang, James J. Davis, Daniele Moro, Piotr Zielinski, Claudionor Coelho, Satrajit Chatterjee, Peter Y. K. Cheung, George A. Constantinides, “Enabling Binary Neural Network Training on the Edge”, <https://arxiv.org/abs/2102.04270>

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

It is worth noting that not all functionality is safe at this time to be used with other high-level operations, such as with layer wrappers. For example, Bidirectional layer wrappers are used with RNNs. If this is required, we encourage users to use quantization functions invoked as strings instead of the actual functions as a way through this, but we may change that implementation in the future.

A first attempt to create a safe mechanism in QKeras is the adoption of QActivation is a wrap-up that provides an encapsulation around the activation functions so that we can save and restore the network architecture, and duplicate them using Keras interface, but this interface has not been fully tested yet.

Activation Layers Implemented in QKeras

- `smooth_sigmoid(x)`
- `hard_sigmoid(x)`
- `binary_sigmoid(x)`
- `binary_tanh(x)`
- `smooth_tanh(x)`
- `hard_tanh(x)`
- `quantized_bits(bits=8, integer=0, symmetric=0, keep_negative=1)(x)`
- `bernoulli(alpha=1.0)(x)`
- `stochastic_ternary(alpha=1.0, threshold=0.33)(x)`
- `ternary(alpha=1.0, threshold=0.33)(x)`
- `stochastic_binary(alpha=1.0)(x)`
- `binary(alpha=1.0)(x)`
- `quantized_relu(bits=8, integer=0, use_sigmoid=0, negative_slope=0.0)(x)`
- `quantized_ulaw(bits=8, integer=0, symmetric=0, u=255.0)(x)`
- `quantized_tanh(bits=8, integer=0, symmetric=0)(x)`
- `quantized_po2(bits=8, max_value=-1)(x)`
- `quantized_relu_po2(bits=8, max_value=-1)(x)`

The `stochastic_*` functions, `bernoulli` as well as `quantized_relu` and `quantized_tanh` rely on stochastic versions of the activation functions. They draw a random number with uniform distribution from `_hard_sigmoid` of the input `x`, and result is based on the expected value of the activation function.

Please refer to the papers if you want to understand the underlying theory, or the documentation in `qkeras/qlayers.py`.

The parameters “bits” specify the number of bits for the quantization, and “integer” specifies how many bits of “bits” are to the left of the decimal point. Finally, our experience in training networks with `QSeparableConv2D`, both `quantized_bits` and `quantized_tanh` that generates values between `[-1, 1]`, required symmetric versions of the range in order to properly converge and eliminate the bias.

Every time we use a quantization for weights and bias that can generate numbers outside the range `[-1.0, 1.0]`, we need to adjust the `*_range` to the number. For example, if we have a `quantized_bits(bits=6, integer=2)` in a weight of a layer, we need to set the weight range to `2**2`, which is equivalent to Catapult HLS `ac_fixed<6, 3, true>`. Similarly, for quantization functions that accept an `alpha` parameter, we need to specify a range of `alpha`, and for `po2` type of quantizers, we need to specify the range of `max_value`.

Example

Suppose you have the following network.

An example of a very simple network is given below in Keras.

```
1 from keras.layers import *
2
3 x = x_in = Input(shape)
4 x = Conv2D(18, (3, 3), name="first_conv2d")(x)
5 x = Activation("relu")(x)
6 x = SeparableConv2D(32, (3, 3))(x)
7 x = Activation("relu")(x)
8 x = Flatten()(x)
9 x = Dense(NB_CLASSES)(x)
10 x = Activation("softmax")(x)
```

You can easily quantize this network as follows:

```
1 from keras.layers import *
2 from qkeras import *
3
4 x = x_in = Input(shape)
5 x = QConv2D(18, (3, 3),
6           kernel_quantizer="stochastic_ternary",
7           bias_quantizer="ternary", name="first_conv2d")(x)
8 x = QActivation("quantized_relu(3)")(x)
9 x = QSeparableConv2D(32, (3, 3),
10                    depthwise_quantizer=quantized_bits(4, 0, 1),
11                    pointwise_quantizer=quantized_bits(3, 0, 1),
12                    bias_quantizer=quantized_bits(3),
13                    depthwise_activation=quantized_tanh(6, 2, 1))(x)
```

```
14 x = QActivation("quantized_relu(3)")(x)
15 x = Flatten()(x)
16 x = QDense(NB_CLASSES,
17           kernel_quantizer=quantized_bits(3),
18           bias_quantizer=quantized_bits(3))(x)
19 x = QActivation("quantized_bits(20, 5)")(x)
20 x = Activation("softmax")(x)
```

The last QActivation is advisable if you want to compare results later on. Please find more cases under the directory examples.

QTools

The purpose of QTools is to assist hardware implementation of the quantized model and model energy consumption estimation. QTools has two functions: data type map generation and energy consumption estimation.

- **Data Type Map Generation:** QTools automatically generate the data type map for weights, bias, multiplier, adder, etc. of each layer. The data type map includes operation type, variable size, quantizer type and bits, etc. Input of the QTools is:

- 1) a given quantized model;
- 2) a list of input quantizers for the model. Output of QTools json file that list the data type map of each layer (stored in `qtools_instance._output_dict`) Output methods include: `qtools_stats_to_json`, which is to output the data type map to a json file; `qtools_stats_print` which is to print out the data type map.

- **Energy Consumption Estimation:** Another function of QTools is to estimate the model energy consumption in Pico Joules (pJ). It provides a tool for QKeras users to quickly estimate energy consumption for memory access and MAC operations in a quantized model derived from QKeras, especially when comparing power consumption of two models running on the same device.

As with any high-level model, it should be used with caution when attempting to estimate the absolute energy consumption of a model for a given technology, or when attempting to compare different technologies.

This tool also provides a measure for model tuning which needs to consider both accuracy and model energy consumption. The energy cost provided by this tool can be integrated into a total loss function which combines energy cost and accuracy.

- **Energy Model:** The best work referenced by the literature on energy consumption was first computed by Horowitz M.: "1.1 computing's energy problem (and what we can do about it)"; IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014

In this work, the author attempted to estimate the energy consumption for accelerators, and for 45 nm process, the data points he presented has since been used whenever someone wants to compare accelerator performance. QTools energy consumption on a 45nm process is based on the data published in this work.

- Examples: Example of how to generate data type map can be found in `qkeras/qtools/examples/example_generate_json.py`. Example of how to generate energy consumption estimation can be found in `qkeras/qtools/examples/example_get_energy.py`

AutoQKeras

AutoQKeras allows the automatic quantization and rebalancing of deep neural networks by treating quantization and rebalancing of an existing deep neural network as a hyperparameter search in Keras-Tuner using random search, hyperband or gaussian processes.

In order to contain the explosion of hyperparameters, users can group tasks by patterns, and perform distribute training using available resources.

Extensive documentation is present in `notebook/AutoQKeras.ipynb`.

Related Work

QKeras has been implemented based on the work of “B.Moons et al. - Minimum Energy Quantized Neural Networks”, Asilomar Conference on Signals, Systems and Computers, 2017 and “Zhou, S. et al. - DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” but the framework should be easily extensible. The original code from QNN can be found below.

<https://github.com/BertMoons/QuantizedNeuralNetworks-Keras-Tensorflow>

QKeras extends QNN by providing a richer set of layers (including SeparableConv2D, DepthwiseConv2D, ternary and stochastic ternary quantizations), besides some functions to aid the estimation for the accumulators and conversion between non-quantized to quantized networks. Finally, our main goal is easy of use, so we attempt to make QKeras layers a true drop-in replacement for Keras, so that users can easily exchange non-quantized layers by quantized ones.

Acknowledgements

Portions of QKeras were derived from QNN.

<https://github.com/BertMoons/QuantizedNeuralNetworks-Keras-Tensorflow>

Copyright (c) 2017, Bert Moons where it applies