



Denglin HammingTM V2

dINNE TVM Quantization

DL-DG/SW-031C-02

2025-02-19

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经上苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼栋5号1101室，江苏，中国

<http://www.denglin.ai>

Email : support@denglin.ai

Change History

Version	Change description
02	The quantization parameters and commands have been supplemented, and for the quantization operator quantized_batch_matmul, its API has been enhanced with the addition of input parameters transpose_a and transpose_b.
01	Initial version.

CONTENTS

- 1 Denglin Quantization OPs documentation 1
 - 1.1 Overview 1
 - 1.2 Quantization OPs API 1
- 2 Denglin Quantization documentation 9
 - 2.1 Overview 9
 - 2.2 Quantization Command Line Tool 10
 - 2.3 Quantization API 14
 - 2.4 Quantization Plugin 19
- Python Module Index 23
- Index 25

CHAPTER ONE

DENGLIN QUANTIZATION OPS DOCUMENTATION

1.1 Overview

To support mixed-precision model, dINNE added a group of quantized OPs which supports uint8 quantization calculation to Relay. To combine quantized OPs with normal OPs, dINNE introduced dl.dequantize. If necessary, dINNE can quantize a tensor with qnn.quantize and send it to a quantized OP and dequantize a quantized tensor and send it to normal float OPs.

1.2 Quantization OPs API

```
dl.relay.op.qnn.dequantize(data, input_scales, input_zero_points, out_dtype='float32',
                           mode='MIN_FIRST')
```

Dequantize.

Parameters

- **data** (tvm.relay.Expr) – The input data to the operator.
- **input_scales** (List[float]) – The scale for the input tensor.
- **input_zero_points** (List[int]) – The zero point for the input tensor.
- **out_dtype** (Optional[str]) – Specifies the output data type for the operator.
- **mode** (Optional[str]) – Compute strategy of the operator.

Returns result – The computed result.

Return type tvn.relay.Expr

```
dl.relay.op.qnn.quantize(data, output_scales, output_zero_points, out_dtype='uint8',
                          mode='MIN_FIRST')
```

Quantize.

Parameters

- **data** (tvm.relay.Expr) – The input data to the operator.
- **output_scales** (List[float]) – The scale for the output tensor.
- **output_zero_points** (List[int]) – The zero point for the output tensor.

- **out_dtype** (Optional[str]) – Specifies the output data type for the operator.
- **mode** (Optional[str]) – Compute strategy of the operator.

Returns result – The computed result.

Return type `tvm.relay.Expr`

```
dl.relay.op.qnn.quantized_add(lhs, rhs, out_dtype, lhs_scales, rhs_scales, scales,
                              lhs_zero_points, rhs_zero_points, zero_points, with_relu,
                              compute_type='float32')
```

Quantized add.

Perform add at the last two dimensions.

Parameters

- **lhs** (`relay.Expr`) – The left hand side input data.
- **rhs** (`relay.Expr`) – The right hand side input data.
- **out_dtype** (`str`) – Specifies the output data type.
- **lhs_scales** (`List[float]`) – The scale for the input tensor a.
- **rhs_scales** (`List[float]`) – The scale for the input tensor b.
- **scales** (`List[float]`) – The scale for the output tensor.
- **lhs_zero_points** (`List[int]`) – The zero point for the input tensor a.
- **rhs_zero_points** (`List[int]`) – The zero point for the input tensor b.
- **zero_points** (`List[int]`) – The zero point for the output tensor.
- **with_relu** (`int`) – int 0 or 1. If it is 1, a relu operation will be performed after addition.
- **compute_type** (`str`) – Possible value is “float16” or “float32”. This op de-quantizes input to float value first before doing add, and dtype of the float value is determined by compute_type.

Returns result – The computed result.

Return type `tvm.relay.Expr`

```
dl.relay.op.qnn.quantized_batch_matmul(inputs, out_dtype, scales_a, scales_b,
                                       scales_product, zero_points_a, zero_points_b,
                                       zero_points_product, transpose_a=False, trans-
                                       pose_b=True)
```

Quantized batch_matmul.

Perform matmul at the last two dimensions.

Parameters

- **inputs** (`Union[List[tvm.relay.Expr], Tuple[tvm.relay.Expr]]`) – The input data to the operator. It can be [a, b] or [a, b, bias].

- **out_dtype** (Optional[str]) – Specifies the output data type.
- **scales_a** (Optional[List[float]]) – The scale for the input tensor a.
- **scales_b** (Optional[List[float]]) – The scale for the input tensor b.
- **scales_product** (Optional[List[float]]) – The scale for the output tensor product.
- **zero_points_a** (Optional[List[int]]) – The zero point for the input tensor a.
- **zero_points_b** (Optional[List[int]]) – The zero point for the input tensor b.
- **zero_points_product** (Optional[List[int]]) – The zero point for the output tensor product.
- **transpose_a** (Optional[bool]) – Whether the first input should be transposed, it's False as default.
- **transpose_b** (Optional[bool]) – whether the second input should be transposed, it's True as default.

Returns result – The computed result.

Return type `tvm.relay.Expr`

```
dl.relay.op.qnn.quantized_conv2d(inputs, strides=1, 1, pad_width=0, 0, 0, 0, 0, 0,
                                0, 0, dilation=1, 1, groups=1, channels=None,
                                kernel_size=None, data_layout='NCHW',
                                kernel_layout='OIHW', out_dtype='uint8',
                                data_scales=None, data_zero_points=None,
                                weight_scales=None, weight_zero_points=None,
                                scales=None, zero_points=None)
```

Quantized 2D convolution.

This operator convolves quantized data with quantized kernel and adds quantized bias (optional). It can output requantized integer data or float data, according to `out_dtype`. If `out_dtype` is `[int8, uint8]`, it will requantize the result with `scales` and `zero_points`.

Parameters

- **inputs** (Union[Tuple[tvm.relay.Expr], List[tvm.relay.Expr]]) – The input tensors to the operator. It can be `[data, weights]` or `[data, weights, bias]`. dtype of data and weight can be `[int8, uint8]`. dtype of bias can be `[int32]`.
- **strides** (Optional[Tuple[int]]) – The strides of convolution.
- **pad_width** (Optional[Tuple[Tuple[int], Tuple[int], Tuple[int], Tuple[int]]]) – The padding of convolution on both sides of inputs before convolution.
- **dilation** (Optional[Tuple[int]]) – Specifies the dilation rate to be used for dilated convolution.

- **groups** (Optional[int]) – Number of groups for grouped convolution.
- **channels** (Optional[int]) – Number of output channels of this convolution.
- **kernel_size** (Optional[Tuple[int]]) – The spatial scale of the convolution kernel.
- **data_layout** (Optional[str]) – Layout of the input.
- **kernel_layout** (Optional[str]) – Layout of the weight.
- **out_dtype** (Optional[str]) – Specifies the output data type for mixed precision conv2d.
- **data_scales** (Optional[List[float]]) – The scale for the input tensor.
- **data_zero_points** (Optional[List[int]]) – The zero point for the input tensor.
- **weight_scales** (Optional[List[float]]) – The scale for the weight tensor.
- **weight_zero_points** (Optional[List[int]]) – The zero point for the weight tensor.
- **scales** (Optional[List[float]]) – The scale for the output tensor.
- **zero_points** (Optional[List[int]]) – The zero point for the output tensor.

Returns **result** – The computed result.

Return type `tvm.relay.Expr`

```
dl.relay.op.qnn.quantized_conv2d_backprop_input(inputs, dilation, strides, input_sizes, padding, data_format, out_dtype, out_backprop_scales, out_backprop_zero_points, filter_scales, filter_zero_points, scales, zero_points, padding_array=0, 0, channels=None, groups=1, output_padding=0, 0, kernel_size=None)
```

Quantized conv2d_backprop_input.

Parameters

- **inputs** (Union(List[relay.Expr], Tuple[relay.Expr], TupleWrapper[relay.Expr])) – The input tensors [data, weight, bias] or [data, weight]. The layout of weight must be HWIO.
- **dilation** (Optional[List[int]]) – Specifies the dilation rate to be used for dilated convolution.
- **strides** (Optional[List[int]]) – The strides of convolution.

- **input_sizes** ([List[int]]) – The output size. This parameter is obsolete and output shape of this op is determined by padding_array, output_padding, channels, groups, kernel_size and shape of input.
- **padding** (Optional[str]) – The padding type of convolution. This parameter is obsolete, please use padding_array and output_padding to control padding.
- **data_format** (Optional[str]) – Layout of the input.
- **out_dtype** (Optional[str]) – Specifies the output data type for mixed precision conv2d_backprop_input.
- **out_backprop_scales** (Optional[List[float]]) – The scale for the input tensor.
- **out_backprop_zero_points** (Optional[List[int]]) – The zero point for the input tensor.
- **filter_scales** (Optional[List[float]]) – The scale for the filter tensor.
- **filter_zero_points** (Optional[List[int]]) – The zero point for the filter/weight tensor.
- **scales** (Optional[List[float]]) – The scale for the output tensor.
- **zero_points** (Optional[List[int]]) – The zero point for the output tensor.
- **explicit_padding** (Tuple[int], optional) – The padding of convolution on both sides of inputs.
- **channels** (int, optional) – Number of output channels of this convolution.
- **groups** (int, optional) – Number of groups for grouped convolution.
- **output_padding** (Tuple[int], optional) – Used to disambiguate the output shape.
- **kernel_size** (Optional[Tuple[int]]) – The spatial scale of the convolution kernel.

Returns **result** – The computed result.

Return type `tvm.relay.Expr`

```
dl.relay.op.qnn.quantized_conv3d(inputs, data_zero_points, weight_zero_points,
                                  data_scales, weight_scales, scales, zero_points, dilation,
                                  strides, padding, data_layout, out_dtype, pad_width=[[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]],
                                  groups=1, channels=None, kernel_size=None)
```

Quantized 3d convolution.

Parameters

- **inputs** (Tuple[tvm.relay.Expr]) – The input data to the operator. It can be [data, weights] or [data, weights, bias].
- **strides** (Optional[Tuple[int]]) – The strides of convolution.
- **padding** (Optional[Tuple[Tuple[int], Tuple[int], Tuple[int], Tuple[int]]]) – The padding of convolution on both sides of inputs before convolution.
- **dilation** (Optional[Tuple[int]]) – Specifies the dilation rate to be used for dilated convolution.
- **data_layout** (Optional[str]) – Layout of the input.
- **out_dtype** (Optional[str]) – Specifies the output data type for mixed precision conv2d.
- **data_scales** (Optional[List[float]]) – The scale for the input tensor.
- **data_zero_points** (Optional[List[int]]) – The zero point for the input tensor.
- **weight_scales** (Optional[List[float]]) – The scale for the weight tensor.
- **weight_zero_points** (Optional[List[int]]) – The zero point for the weight tensor.
- **scales** (Optional[List[float]]) – The scale for the output tensor.
- **zero_points** (Optional[List[int]]) – The zero point for the output tensor.
- **groups** (Optional[int]) – Number of groups for grouped convolution.
- **channels** (Optional[int]) – Number of output channels of this convolution.
- **kernel_size** (Optional[Tuple[int]]) – The spatial scale of the convolution kernel.

Returns **result** – The computed result.

Return type `tvm.relay.Expr`

`dl.relay.op.qnn.quantized_embedding_bag(embedding_matrix, scales, zero_points, indices, offsets, include_last_offset, mode)`

Quantized embedding_bag.

Parameters

- **embedding_matrix** (relay.Expr) – The embedding matrix with the number of rows equal to the maximum possible index + 1, and the number of columns equal to the embedding size.
- **scales** (relay.Expr) – per_channel scales of embedding_matrix.

- **zero_points** (relay.Expr) – per_channel zero_points of embedding_matrix.
- **indices** (relay.Expr) – Tensor containing bags of indices into the embedding matrix.
- **offsets** (relay.Expr) – offsets determine the starting index position of each bag (sequence) in input.
- **include_last_offset** (bool) – if True, offsets have one additional element, where the last element is equivalent to the size of indices.
- **mode** (str) – “sum” Specifies the way to reduce the bag, only “sum” is supported.

Returns result – [embedding_bag, offset2bag, bag_size, max_indices]

Return type relay.TupleWrapper

Note: dtype of output is float

`dl.relay.op.qnn.quantized_matmul(inputs, out_dtype, scales_a, scales_b, scales_product, zero_points_a, zero_points_b, zero_points_product)`

Quantized matmul.

out_dtype can be “uint8”, “int8”, “float16”. If out_dtype is integer, it will requantize the result with scales and zero_points.

Parameters

- **inputs** (Union[List[tvm.relay.Expr], Tuple[tvm.relay.Expr]]) – The input data to the operator. It can be [a, b] or [a, b, bias].
- **out_dtype** (Optional[str]) – Specifies the output data type quantized_matmul.
- **scales_a** (Optional[List[float]]) – The scale for the input tensor a.
- **scales_b** (Optional[List[float]]) – The scale for the input tensor b.
- **scales_product** (Optional[List[float]]) – The scale for the output tensor product.
- **zero_points_a** (Optional[List[int]]) – The zero point for the input tensor a.
- **zero_points_b** (Optional[List[int]]) – The zero point for the input tensor b.
- **zero_points_product** (Optional[List[int]]) – The zero point for the output tensor product.

Returns result – The computed result.

Return type tvn.relay.Expr

CHAPTER TWO

DENGLIN QUANTIZATION DOCUMENTATION

2.1 Overview

dlnNE TVM provides an automatic quantization toolkit.

2.1.1 Workflow

This is a standalone toolkit which does not depend on DengLin hardware. User should use this tool to quantize a float model into a quantized model, then compile the quantized model with dlnNE.

2.1.2 Inputs

Trained float model

To use the quantization toolkit, user must convert a trained float model into a `tvm.IRModule`.

Calibration dataset

User has to provide calibration dataset for calibrating the quantized model. The calibration dataset must be preprocessed and can be fed into the trained float model directly.

2.1.3 Output

Output of this toolkit is a `tvm.IRModule`.

Serialize and store

Before being sent to dlnNE, a quantized module needs to be serialized and stored in a file. The quantized module can be serialized with `tvm.ir.save_json()`, and then written to a file, and the file name should end with `rlym`. This file is not compatible between different SDKs.

```
1 mod_name = "quantized_xxx.rlym"
2 with open(mod_name, "w") as f:
3     f.write(tvm.ir.save_json(mod))
```

2.2 Quantization Command Line Tool

dlTVM CLI provides tools to quantize a model

2.2.1 Convert

```
1 # Convert test.onnx
2 python3 -m dl convert test.onnx --output-model test.rlym
```

2.2.2 Quantize

```
1 # Quantize test.rlym, the plugin include data preprocess and postprocess .
2 python3 -m dl quantize test.rlym --dataset /your-path/dataset --plugin /your-path/
  ↪ quantization_plugin.py --output-model test.quantized.rlym
```

Options

check-acc: Whether should we do the acc comparation for the original and quantized model, while the quantization completed .

Example:

```
python3 -m dl quantize test.rlym --check-acc ...
```

check-acc-for-tragets: when we do the check-acc, we can freely set the target for original model and quantized model .

Example:

```
python3 -m dl quantize test.rlym --check-acc-for-tragets 'llvm llvm' ...
```

optimize: Optimize the quantized model .

Example:

```
python3 -m dl quantize --optimize test.rlym ...
```

calibrate-mode: The calibration mode, ‘min_max’ or ‘kl_divergence’, ‘kl_divergence’ as default, min_max: find the min and max in the activation output, kl_divergence: find scales by kl divergence on the dataset .

Example:

```
python3 -m dl quantize --calibrate-mode "min_max" test.rlym ...
```

calibrate-device: The target for quantization, only supports ‘llvm’ ‘cuda’ and ‘nne’.

Example:

```
python3 -m dl quantize --calibrate-device "nne" test.rlym ...
```

input-data-dir: Directory in which input datas are stored, for the format, [a.npy,b.npy] [a_1.npy,b_1.npy] to [a_n.npy,b_n.npy], a and b are input names .

Example:

```
python3 -m dl quantize test.rlym --input-data-dir /home/user/dat ...
```

regions-configure: It’s about quantization region settings, if it set to ‘generate’, the quantize will generate the region configure file, if it set to a file path, the quantize will run according to the configure file .

Example:

```
python3 -m dl quantize test.rlym --regions-configure "generate" ...  
python3 -m dl quantize test.rlym --regions-configure /home/user/region_config.json ...
```

dump-regions: Dump quantization regions .

Example:

```
python3 -m dl quantize test.rlym --dump-regions ...
```

arch: The arch of hardware for which model is quantized. Possible value are [“v1”, “v2”] and default value is “v1” .

Example:

```
python3 -m dl quantize --arch=v2 test.rlym ...
```

quantize_swish: This option controls whether the swish op will be quantized, stored true .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize --quantize_swish test.rlym ...
```

quantize_hardswish: This option controls whether the hard-swish op will be quantized, stored true .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize --quantize_hardswish test.rlym ...
```

quantize_mish: This option controls whether the mish op will be quantized, stored true .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize --quantize_mish test.rlym ...
```

resize2d_to_tu: This option controls whether convert resize2d to convolution which will be quantized .

Example:

```
python3 -m dl quantize --resize2d_to_tu test.rlym ...
```

downcast: Downcast region ops to float16 .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize test.rlym --downcast ...
```

downcast_add: Downcast add to float16, if “quantize –downcast”, downcast_add defaults to True, otherwise it is False, and you should use it like this “quantize –downcast_add”.

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize test.rlym --downcast or python3 -m dl quantize test.rlym --downcast_
↪add ...
```

downcast_concat: Downcast concatenate to float16, if “quantize –downcast”, downcast_concat defaults to True, otherwise it is False, and you should use it like this “quantize –downcast_concat” .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize test.rlym --downcast or python3 -m dl quantize test.rlym --downcast_
↪concat ...
```

low_precision_float_regions: convert all ops except the quantization ops to float16 .

Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Example:

```
python3 -m dl quantize test.rlym --low_precision_float_regions ...
```

perchannel: Make dense, matmul and batch_matmul weights support perchannel quantization, stored True .

Example:

```
python3 -m dl quantize test.rlym --perchannel ...
```

just_insert_dump_for_specified_spans: Only insert dump operator for the specified spans .

Example:

```
python3 -m dl quantize test.rlym --just_insert_dump_for_specified_spans "span1 span2" ...
```

logic_insert_dump_for_float32: Insert dump for the float model, stored True .

Example:

```
python3 -m dl quantize test.rlym --logic_insert_dump_for_float32 ...
```

logic_insert_dump_for_int8: Insert dump for the quantized model, stored True .

Example:

```
python3 -m dl quantize test.rlym --logic_insert_dump_for_int8 ...
```

analyze_dump_info_in_dir: Analyze the dump info in directory .

Example:

```
python3 -m dl quantize test.rlym --analyze_dump_info_in_dir build/data ...
```

no-concatenate_quantize_correct: When both the input and output of concatenate op require quantization, whether should we align their scales for the quantize and dequantize elimination, stored False .

Notice: After enabling this option, while it may improve accuracy, it could also lead to a decrease in performance, as the quantize and dequantize operators associated with concatenate might not be able to eliminate, thereby increasing computation .

Example:


```
python3 -m dl.quantize.test.rlym --no-concatenate_quantize_correct ...
```

2.3 Quantization API

2.3.1 API Definition

`dl.quantize.quantize.quantize(mod, params, dataset, to_onnx=False)`

Convert a float model into a mixed-precision model.

Parameters

- **mod** (Module) – The original module.
- **params** (dict of str to NDAarray) – Input parameters to the graph that do not change during inference time. Used for constant folding.

For a model whose format is supported by TVM, user can use `tvmc.frontends.load_model` to get mod and params.

- **dataset** (list of dict of Str -> NDAarray or CalibrationDataset) – The calibration dataset.

Returns ret

Return type Module or tuple[Module, dict{str,dict{str,float}}]

Example

Use `qconfig` to control quantization

with `qconfig(quantized_input=True, quantize_add=False)`: `qmod = quantize(mod, params, dataset)`

or

with `qconfig(quantize_add=False)`: `qmod = quantize(mod, params, dataset)`

if `quantized_input` is `True`: `qmod, qinfo = qmod`

`dl.quantize.quantize.qconfig(**kwargs)`

Configure the quantization behavior by setting config variables.

Parameters

- **arch** (str) – The arch of hardware for which model is quantized. Possible values are ["v1", "v2"] and default value is "v1".
- **calibrate_mode** (str) – The calibration mode. 'min_max' or 'kl_divergence'. `min_max`: find the min and max in the activation output. `kl_divergence`: find scales by kl divergence on the dataset.

- **calibrate_device** (str) – The target for quantization, only support ‘llvm’, ‘cuda’ and ‘nne’
- **regions_configure** (str) – It’s about quantization region settings, if it set to ‘generate’, the quantize will generate the region configure file, if it set to path of file, the quantize will run according to the configure file.
- **dump_regions** (bool) – Dump quantization regions, False by default.
- **quantize_add** (bool) – Whether to quantize the operator add, True by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **quantize_pooling** (bool) – Whether to quantize operators global_avg_pool2d, max_pool2d and avg_pool2d which composes a whole quantization region, True by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **quantize_concat** (bool) – Whether to quantize the operator concatenate, True by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **quantize_swish** (bool) – Whether to quantize the operator swish, False by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **quantize_hardswish** (bool) – Whether to quantize the operator hardswish, False by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **quantize_mish** (bool) – Whether to quantize the operator mish, False by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **downcast_pooling** (bool) – Whether to force input of standalone avg_pool2d to be float16, True by default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

Relation between quantize_pooling and downcast_pooling:

quantize_pooling	downcast_pooling	avg_pool2d
True	True	fp16
True	False	int
False	True	fp16
False	False	fp32

- **downcast** (bool) – Whether to downcast the model to a mixed precision float16 model. If this parameter is enabled, dataset is not needed, user can pass None. The output is a float16 and float32 mixed precision

model. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.

- **downcast_add** (bool) – Downcast add to float16, if “quantize –downcast”, downcast_add defaults to True, otherwise it is False, and you should use it like this “quantize –downcast_add”. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **downcast_concat** (bool) – Downcast concatenate to float16, False as default. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **low_precision_float_regions** (bool) – convert all ops except the quantization ops to float16. Notice: this option, while aiming to enhance performance, may potentially lead to a decrease in accuracy.
- **perchannel** (bool) – Make dense, matmul and batch_matmul weights support perchannel quantization, False as default.
- **just_insert_dump_for_specified_spans** (list) – Only insert dump operator for the specified spans. for example, just_insert_dump_for_specified_spans=['span1','span2']
- **logic_insert_dump_for_float32** (bool) – Insert dump for the float model, False as default.
- **logic_insert_dump_for_int8** (bool) – Insert dump for the quantized model, False as default.
- **concatenate_quantize_correct** (bool) – While both the input and output of concatenate op need to be quantized, whether we need to flatten the scales for quantize and dequantize elimination. By default, it's True, while python3 -m dl quantize –no-concatenate_quantize_correct, it's False. Notice: while it's False, this may lead to a decrease of performance, but also may improve the accuracy, because this will cause the concat-related quantize and dequantize op to not be eliminated.

Returns **config** – The quantization configuration.

Return type QConfig

`dl.quantize.quantize.query_regions_configure(mod) → Dict[str, Dict[str, int]]`

Query regions configure of a module

Parameters **mod** (Module) – The original module.

Returns

config – Key is the index of layer Value is a Dict whose key is name of configure item and value is configure value

Format of config is described in “dlnNE TVM Quantization” 2.2.3

Return type Dict[str, Dict[str, int]]

2.3.2 Sample

```

1  tvmc_model = tvmc.frontends.load_model(model_path)
2
3  if isinstance(tvmc_model, tuple):
4      mod, params = tvmc_model
5  else:
6      mod = tvmc_model.mod
7      params = tvmc_model.params
8
9  dataset = [{"input": np_data}]
10
11 with qconfig(calibrate_mode="min_max"):
12     quantized_mod = quantize(mod, params, dataset)

```

2.3.3 Configuring Quantization Regions

Quantization regions

Quantization regions are the minimum sections of a model which can be quantized.

How to configure quantization regions

User can pass a region configuration file to *quantize* to control quantization options for each region.

Format

```

1  {
2      "0": {
3          "quantize": 1,
4          "float16": 0,
5          "activation": 1,
6          "pooling": 1
7      },
8  }

```

The top level object is configured for each quantization region.

“0”: Name of the quantization region.

“quantize”:

Whether to quantize this region, the default value is 1.

“float16”:

Works when “quantize” is 0, and if “float16” is 1, this region will be downcasted to float16.

“activation”:

Whether to keep activation and later operators in this region.

“pooling”:

Whether to keep pooling and later operators in this region.

Get a template config file

```
1 with qconfig(regions_configure="generate"):
2     quantize(...)
```

When `regions_configure="generate"`, it generates a file named `dl_quantize_config.json`, which enables all the regions by default; it also generates a file named `dl_quantization_regions_info.log`, which includes quantization region information.

Understand `dl_quantization_regions_info.log`

It contains all information about quantization regions, for example:

```
1 %23 = fn (%FunctionVar_53_0:, %FunctionVar_53_1, %bias5, PartitionedFromPattern="nn.conv2d_
↪nn.bias_add_nn.relu_nn.max_pool2d_", Index=0) {
2     %20 = nn.conv2d(%FunctionVar_53_0, %FunctionVar_53_1) /* conv1 */;
3     %21 = nn.bias_add(%20, %bias5);
4     %22 = nn.relu(%21) /* conv1_relu */;
5     nn.max_pool2d(%22) /* pool1 */
6 };
```

- `Index=0` means it is the first quantization region, “0” is the name of the quantization region.
- `conv1` is the span information, which means this Call is converted from original model node (layer) conv1.

Note:

1. Some Calls are not marked with the original layer name because they are generated during quantization.
 2. Some original layers may disappear completely in the converted relay module.
 3. For an in-place caffe layer, its top name is replaced with the layer name to avoid the same name for different calls.
 4. The span information can be changed because of many reasons, so it is only for debugging, and production code should not depend on the span information.
- `PartitionedFromPattern="nn.conv2d_nn.bias_add_nn.relu_nn.max_pool2d_"` means there are conv2d, bias_add, relu and max_pool2d in this region.

Use a config file

```
1 with qconfig(regions_configure="your_config.json"):
2     quantize(...)
```

You have to use another name instead of `dl_quantize_config.json`, because the default file may be overwritten unintentionally.

Query and apply a modified regions configure

```
1 with qconfig(quantize_gelu=True):
2     config = query_regions_configure(mod)
3
4     # The keys are all str, the values are all number
5     # Set layer "0" as "float16"
6     config["0"]["float16"] = 1
7     config["0"]["quantize"] = 0
8
9     # Keep other quantization configures the same as the one used for querying
10    with qconfig(quantize_gelu=True, regions_configure=config):
11        quantize(mod, None, [{"x": data_np}])
```

2.4 Quantization Plugin

2.4.1 What is a quantization plugin

A quantization plugin is implemented as a standalone python file whose path can be passed to dlTVM CLI. dlTVM CLI calls the plugin to get calibration dataset and uses the metric function provided by the plugin to calculate accuracy.

```
1 # Quantize test.rlym
2 python3 -m dl quantize test.rlym --plugin quantization_plugin.py
3
4 # Get accuracy of test.quantized.rlym
5 python3 -m dl acc test.quantized.rlym --plugin quantization_plugin.py
```

2.4.2 Entry callback of a quantization plugin

In the plugin python file, a function called `get_quantization_plugin()` must be implemented, its definition should be

```
1 def get_quantization_plugin(**kwargs) -> dl.quantize.plugin_interface.PluginBase:
```

and it must return an object of a class derived from `dl.quantize.plugin_interface.PluginBase`.

2.4.3 Implement the Plugin class

The Plugin class is used to generate datasets and a metric. It should be derived from `dl.quantize.plugin_interface.PluginBase`. There are three members which must be implemented: `get_metric`, `get_calibration_dataset`, `get_validate_dataset`.

2.4.4 Implement the Metric class

The Metric class is used to calculate accuracy. It should be derived from `dl.quantize.plugin_interface.MetricBase`.

2.4.5 Implement the CalibrationDataset class

The CalibrationDataset class is used to get a dataset for calibration and validation. It should be derived from `dl.quantize.plugin_interface.CalibrationDatasetBase`. Normally, there should be a CalibrationDataset class for calibration and a CalibrationDataset class for validation.

2.4.6 Plugin template

```

1  from dl.quantize.plugin_interface import MetricBase, CalibrationDatasetBase, PluginBase
2
3  class Metric(MetricBase):
4      def __init__(self, model):
5          super().__init__()
6
7      def update(self, label, preds):
8
9      def get(self):
10
11     def reset(self):
12
13     class CalibrationDataset(CalibrationDatasetBase):
14         def __init__(self, inputs_names, model, is_calibrate, num_samples):
15             super().__init__()
16
17         def __iter__(self):
18
19         def __next__(self) -> dict:
20
21         def __len__(self) -> int:
22
23         def reset(self):
24
25         def get_sample(self, index, rand=0):
26
27
28     class QuatizationPlugIn(PluginBase):
29         def __init__(self, model_name, inputs_names, dataset_path):

```

(continues on next page)

(continued from previous page)

```

30     super().__init__()
31
32     def get_validate_dataset(self, num_samples):
33         return CalibrationDataset(self.inputs_names, self.model, False, num_samples)
34
35     def get_calibration_dataset(self, num_samples):
36         return CalibrationDataset(self.inputs_names, self.model, True, num_samples)
37
38     def get_metric(self):
39         return Metric(self.model)
40
41
42     def get_quantization_plugin(**kwargs):
43         model_name = kwargs["model_name"]
44         inputs_names = kwargs["inputs_names"]
45         dataset_path = kwargs["dataset_path"]
46         return QuantizationPlugin(model_name, inputs_names, dataset_path)

```

2.4.7 API Definition

class dl.quantize.plugin_interface.**MetricBase**
Base of all metric classes.

get()
Gets the current evaluation result.

Returns

- **names** (*list of str*) – Names of the metrics.
- **values** (*list of float*) – Values of the evaluations.

update(label, preds)
Updates the internal evaluation result.

Parameters

- **labels** (*list of NDArray*) – The labels of the data.
- **preds** (*list of NDArray*) – Predicted values.

class dl.quantize.plugin_interface.**CalibrationDatasetBase**
Base of all calibration dataset classes.

class dl.quantize.plugin_interface.**PluginBase**
Base of all plugin classes.

Plugin author should implement a class derived from PluginBase, and override its members.

get_calibration_dataset(model_name, inputs_names, dataset_path) →
[*dl.quantize.plugin_interface.CalibrationDatasetBase*](#)
 Get a dataset which will be used for calibration.

Parameters

- **model_name** (str) – Name of the model.
- **inputs_names** (list of str) – List of input names.
- **dataset_path** (str) – Path of the dataset.

Returns dataset – An object of a class derived from CalibrationDatasetBase.

Return type *CalibrationDatasetBase*

get_metric() → *dl.quantize.plugin_interface.MetricBase*

Get a metric which is used to calculate accuracy.

Returns metric – An object of a class derived from MetricBase.

Return type *MetricBase*

get_validate_dataset(model_name, inputs_names, dataset_path) →
dl.quantize.plugin_interface.CalibrationDatasetBase

Get a dataset which will be used for validation.

Parameters

- **model_name** (str) – Name of the model.
- **inputs_names** (list of str) – List of input names.
- **dataset_path** (str) – Path of the dataset.

Returns dataset – An object of a class derived from CalibrationDatasetBase.

Return type *CalibrationDatasetBase*

PYTHON MODULE INDEX

d

`dl.quantize.plugin_interface`, [21](#)
`dl.quantize.quantize`, [14](#)
`dl.relay.op.qnn`, [1](#)

INDEX

C

CalibrationDatasetBase (class in
dl.quantize.plugin_interface), 21

D

dequantize() (in module *dl.relay.op.qnn*), 1
dl.quantize.plugin_interface
 module, 21
dl.quantize.quantize
 module, 14
dl.relay.op.qnn
 module, 1

G

get() (*dl.quantize.plugin_interface.MetricBase*
 method), 21
 get_calibration_dataset()
 (*dl.quantize.plugin_interface.PluginBase*
 method), 21
 get_metric()
 (*dl.quantize.plugin_interface.PluginBase*
 method), 22
 get_validate_dataset()
 (*dl.quantize.plugin_interface.PluginBase*
 method), 22

M

MetricBase (class in
dl.quantize.plugin_interface), 21
 module
dl.quantize.plugin_interface, 21
dl.quantize.quantize, 14
dl.relay.op.qnn, 1

P

PluginBase (class in
dl.quantize.plugin_interface), 21

Q

qconfig() (in module *dl.quantize.quantize*), 14
 quantize() (in module *dl.quantize.quantize*), 14
 quantize() (in module *dl.relay.op.qnn*), 1
 quantized_add() (in module *dl.relay.op.qnn*), 2
 quantized_batch_matmul() (in module
dl.relay.op.qnn), 2
 quantized_conv2d() (in module
dl.relay.op.qnn), 3
 quantized_conv2d_backprop_input() (in mod-
 ule *dl.relay.op.qnn*), 4
 quantized_conv3d() (in module
dl.relay.op.qnn), 5
 quantized_embedding_bag() (in module
dl.relay.op.qnn), 6
 quantized_matmul() (in module
dl.relay.op.qnn), 7
 query_regions_configure() (in module
dl.quantize.quantize), 16

U

update()
 (*dl.quantize.plugin_interface.MetricBase*
 method), 21