An overview of QuantTensor and QuantConv2d

In this initial tutorial, we take a first look at `QuantTensor`, a basic data structure in Brevitas, and at `QuantConv2d`, a typical quantized layer. `QuantConv2d` is an instance of a `QuantWeightBiasInputOutputLayer` (typically imported as `QuantWBIOL`), meaning that it supports quantization of its weight, bias, input and output. Other instances of `QuantWBIOL` are `QuantLinear`, `QuantConv1d`, `QuantConvTranspose1d` and `QuantConvTranspose2d`, and they all follow the same principles.

If we take a look at the `__init__` method of `QuantConv2d`, we notice a few things:

[1]:
```
import inspect
from brevitas.nn import QuantConv2d
from brevitas.nn import QuantIdentity
from IPython.display import Markdown, display
import torch

# helpers
def assert_with_message(condition):
    assert condition
    print(condition)

def pretty_print_source(source):
    display(Markdown('```python\n' + source + '\n```'))

# set manual seed for the notebook
torch.manual_seed(0)

source = inspect.getsource(QuantConv2d.__init__)
pretty_print_source(source)
```

```
def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: Union[int, Tuple[int, int]],
        stride: Union[int, Tuple[int, int]] = 1,
        padding: Union[int, Tuple[int, int]] = 0,
```

Skip to main content

```
        padding_mode: str = 'zeros',
        bias: Optional[bool] = True,
        weight_quant: Optional[WeightQuantType] = Int8WeightPerTensorFloat,
        bias_quant: Optional[BiasQuantType] = None,
        input_quant: Optional[ActQuantType] = None,
        output_quant: Optional[ActQuantType] = None,
        return_quant_tensor: bool = False,
        device: Optional[torch.device] = None,
        dtype: Optional[torch.dtype] = None,
        **kwargs) -> None:
    # avoid an init error in the super class by setting padding to 0
    if padding_mode == 'zeros' and padding == 'same' and (stride > 1 if isinstance(
            stride, int) else any(map(lambda x: x > 1, stride))):
        padding = 0
        is_same_padded_strided = True
    else:
        is_same_padded_strided = False
    Conv2d.__init__(
        self,
        in_channels=in_channels,
        out_channels=out_channels,
        kernel_size=kernel_size,
        stride=stride,
        padding=padding,
        padding_mode=padding_mode,
        dilation=dilation,
        groups=groups,
        bias=bias,
        device=device,
        dtype=dtype)
    QuantWBIOL.__init__(
        self,
        weight_quant=weight_quant,
        bias_quant=bias_quant,
        input_quant=input_quant,
        output_quant=output_quant,
        return_quant_tensor=return_quant_tensor,
        **kwargs)
    self.is_same_padded_strided = is_same_padded_strided
```

`QuantConv2d` is an instance of both `Conv2d` and `QuantWBIOL`. Its initialization method exposes the usual arguments of a `Conv2d`, as well as: an extra flag to support *same padding*; *four* different arguments to set a quantizer for - respectively - *weight*, *bias*, *input*, and *output*; a `return_quant_tensor` boolean flag; the `**kwargs` placeholder to intercept additional arbitrary keyword arguments.

In this tutorial we will focus on how to set the four quantizer arguments and the return flags; arbitrary kwargs will be explained in a separate tutorial dedicated to defining and overriding quantizers.

Skip to main content

By default `weight_quant=Int8WeightPerTensorFloat`, while `bias_quant`, `input_quant` and `output_quant` are set to `None`. That means that by default weights are quantized to *8-bit signed integer with a per-tensor floating-point scale factor* (a very common type of quantization adopted by e.g. the ONNX standard opset), while quantization of bias, input, and output are disabled. We can easily verify all of this at runtime on an example:

[2]:
```
default_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False)
```

[3]:
```
print(f'Is weight quant enabled: {default_quant_conv.weight_quant.is_quant_enabled}')
print(f'Is bias quant enabled: {default_quant_conv.bias_quant.is_quant_enabled}')
print(f'Is input quant enabled: {default_quant_conv.input_quant.is_quant_enabled}')
print(f'Is output quant enabled: {default_quant_conv.output_quant.is_quant_enabled}')
```

```
Is weight quant enabled: True
Is bias quant enabled: False
Is input quant enabled: False
Is output quant enabled: False
```

If we now try to pass in a random floating-point tensor as input, as expected we get the output of the convolution:

[4]:
```
import torch

out = default_quant_conv(torch.randn(1, 2, 5, 5))
out
```

```
/proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10/site
  return super(Tensor, self).rename(names)
[W NNPACK.cpp:53] Could not initialize NNPACK! Reason: Unsupported hardware.
```

[4]:
```
tensor([[[[ 0.2908, -0.1793, -0.9610],
          [-0.6542, -0.3532,  0.6361],
          [ 1.0290,  0.2730,  0.0969]],

         [[-0.3479,  0.6030,  0.4900],
          [ 0.1607,  0.3547, -0.4283],
          [-0.6696,  0.0652,  0.7300]],

         [[-0.0769, -0.2424,  0.1860],
          [ 0.1740, -0.1182, -0.7017],
          [ 0.0963,  0.2375, -0.9439]]]], grad_fn=<ConvolutionBackward0>)
```

In this case we are computing the convolution between an unquantized input tensor and quantized weights, so the output in general is unquantized.

A QuantConv2d with quantization disabled everywhere behaves like a standard `Conv2d`. Again can easily verify this:

Skip to main content

[5]:
```
from torch.nn import Conv2d

torch.manual_seed(0)  # set a seed to make sure the random weight init is reproducible
disabled_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False, weight_quant=None)
torch.manual_seed(0)  # reproduce the same random weight init as above
float_conv = Conv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False)
inp = torch.randn(1, 2, 5, 5)
assert_with_message(torch.isclose(disabled_quant_conv(inp), float_conv(inp)).all().item
```

```
True
```

As we have just seen, Brevitas allows users as much freedom as possible to experiment with quantization, meaning that computation between quantized and unquantized values is considered legal. This allows users to mix Brevitas layers with Pytorch layers with little restrictions.
To make this possible, quantized values are typically represented in *dequantized format*, meaning that - in the case of affine quantization implemented in Brevitas - zero-point and scale factor are applied to their integer values according to the formula **quant_value = (integer_value - zero_point) * scale**.

QuantTensor

We can directly observe the quantized weights by calling the weight quantizer on the layer's weights: `default_quant_conv.weight_quant(quant_conv.weight)`, which for shortness is already implemented as `default_quant_conv.quant_weight()` :

[6]:
```
default_quant_conv.quant_weight()
```

[6]:
```
IntQuantTensor(value=tensor([[[[-0.0018,  0.1273, -0.1937],
              [-0.1734, -0.0904,  0.0627],
              [-0.0055,  0.1863, -0.0203]],

             [[ 0.0627, -0.0720, -0.0461],
              [-0.2251, -0.1568, -0.0978],
              [ 0.0092,  0.0941,  0.1421]]],


            [[[-0.1605, -0.1033,  0.0849],
              [ 0.1956, -0.0480,  0.1771],
              [-0.0387,  0.0258,  0.2140]],

             [[-0.2196, -0.1476, -0.0590],
              [-0.0923,  0.2030, -0.1531],
              [-0.1089, -0.1642, -0.2214]]],
```

Skip to main content

```
[[[-0.1384,  0.2030,  0.1052],
  [ 0.1144,  0.0129, -0.1199],
  [ 0.0406, -0.2196, -0.1697]],

 [[-0.1218,  0.1494,  0.1384],
  [-0.1052, -0.0092,  0.1513],
  [ 0.2343,  0.0941,  0.0314]]]], grad_fn=<MulBackward0>), scale=tensor(0.0018,
```

Notice how the quantized weights are wrapped in a data structure implemented by Brevitas called `QuantTensor`. A `QuantTensor` is a way to represent an affine quantized tensor with all its metadata, meaning: the `value` of the quantized tensor in *dequantized* format, `scale`, `zero_point`, `bit_width`, whether the quantized value it's `signed` or not, and whether the tensor was generated in `training` mode.

As expected, we have that the quantized value (in dequantized format) can be computer from its integer representation, together with zero-point and scale:

[7]:
```
int_weight = default_quant_conv.quant_weight().int()
zero_point = default_quant_conv.weight_quant.zero_point()
scale = default_quant_conv.weight_quant.scale()
quant_weight_manually = (int_weight - zero_point) * scale

assert_with_message(default_quant_conv.quant_weight().value.isclose(quant_weight_manu
```

```
True
```

A *valid* QuantTensor correctly populates all its fields with values `!= None` and respect the **affine quantization invariant**, i.e. `value / scale + zero_point` is (accounting for rounding errors) an *integer* that can be represented within the interval defined by the `bit_width` and `signed` fields of the `QuantTensor`. A *non-valid* one doesn't. We can observe that the quantized weights are indeed marked as valid:

[8]:
```
assert_with_message(default_quant_conv.quant_weight().is_valid)
```

```
True
```

Calling `is_valid` is relative expensive, so it should be using sparingly, but there are a few cases where a non-valid QuantTensor might be generated that is important to be aware of. Say we have two QuantTensor as output of the same quantized activation, and we want to sum them together:

[9]:
```
quant_act = QuantIdentity(return_quant_tensor=True)

out_tensor_0 = quant_act(torch.randn(1,2,5,5))
out_tensor_1 = quant_act(torch.randn(1,2,5,5))
```

```
assert_with_message(out_tensor_1.is_valid)
print(out_tensor_0.scale)
print(out_tensor_1.scale)
```

```
True
True
tensor(0.0211, grad_fn=<AbsBinarySignGradFnBackward>)
tensor(0.0162, grad_fn=<AbsBinarySignGradFnBackward>)
```

Both QuantTensor are valid but since the quantized activation is in training mode by default, their scale factors are going to be different. It is important to note that the behaviour is different at evaluation time, where the two scale factors will be the same.

[10]:
```
out_tensor = out_tensor_0 + out_tensor_1
print(out_tensor)
```

```
IntQuantTensor(value=tensor([[[[-0.1106,  1.1945, -0.4972, -2.0968,  0.7175],
          [-2.5901,  0.0588, -0.2014,  2.1486,  1.6435],
          [ 0.9067, -2.5212,  2.2193,  0.2352, -0.8395],
          [-0.8351,  0.6341, -0.5551,  0.1040, -3.3151],
          [-0.8979, -0.7092,  3.8232,  1.0875,  0.3954]],

         [[ 1.4363, -1.3973,  1.3249,  2.6914,  0.3660],
          [ 1.5057,  1.8094,  0.5100, -1.6874,  1.9981],
          [ 1.2472, -1.7813,  0.0334, -1.2880, -2.9333],
          [ 0.0180, -1.4298, -2.9978,  0.5494, -1.4548],
          [ 1.6738, -0.3177, -0.3721, -0.1650, -1.1871]]]],
       grad_fn=<AddBackward0>), scale=0.018651068210601807, zero_point=0.0, bit_widt
```

Because we set `training` to `True` for both of them, we are allowed to sum them even if they have different scale factors. The output QuantTensor will have the correct `bit_width`, and a scale which is the average of the two original scale factors. This is done only at training time, in order to propagate gradient information, however the consequence is that the resulting QuantTensor is no longer valid:

[11]:
```
assert_with_message(not out_tensor.is_valid)
```

```
True
```

`QuantTensor` implements `__torch_function__` to handle being called from torch functional operators (e.g. ops under `torch.nn.functional`). Passing a QuantTensor to supported ops that are invariant to quantization, e.g. max-pooling, preserve the the validity of a QuantTensor. Example:

[12]:
```
import torch

quant_identity = QuantIdentity(return_quant_tensor=True)
quant_tensor = quant_identity(torch.randn(1, 3, 4, 4))
torch.nn.functional.max_pool2d(quant_tensor, kernel_size=2, stride=2)
```

[12]:
```
IntQuantTensor(value=tensor([[[[0.5191, 0.6402],
```

Skip to main content

```
               [[2.0417, 0.5883],
                [1.2631, 0.3980]],

               [[0.7959, 0.5191],
                [0.8132, 1.3496]]]], grad_fn=<MaxPool2DWithIndicesBackward0>), scale=tensor(0
```

For ops that are not invariant to quantization, a `QuantTensor` decays into a floating-point `torch.Tensor`. Example:

[13]:
```
torch.tanh(quant_tensor)
```

[13]:
```
tensor([[[[ 0.4770,  0.2212,  0.0691,  0.5650],
          [-0.0346, -0.6618, -0.4635, -0.3482],
          [ 0.9730, -0.7245, -0.5881, -0.5287],
          [-0.0863,  0.8857,  0.5287, -0.4498]],

         [[ 0.9669,  0.5650, -0.6211, -0.4498],
          [-0.2376,  0.6103,  0.5287,  0.2700],
          [-0.6808,  0.8519,  0.2700, -0.5531],
          [-0.0173,  0.8264,  0.3782, -0.1881]],

         [[-0.6211, -0.9764, -0.5993,  0.4770],
          [ 0.5033,  0.6618, -0.1881, -0.6211],
          [-0.8031,  0.1375,  0.5287,  0.8740],
          [-0.6714,  0.6714, -0.5650,  0.8611]]]], grad_fn=<TanhBackward0>)
```

Input Quantization

We can obtain a valid output `QuantTensor` by making sure that both input and weight of `QuantConv2d` are quantized. To do so, we can set a quantizer for `input_quant`. In this example we pick a *signed 8-bit* quantizer with *per-tensor floating-point scale factor*:

[14]:
```
from brevitas.quant.scaled_int import Int8ActPerTensorFloat

input_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False,
    input_quant=Int8ActPerTensorFloat, return_quant_tensor=True)
out_tensor = input_quant_conv(torch.randn(1, 2, 5, 5))
out_tensor
```

[14]:
```
IntQuantTensor(value=tensor([[[[-0.3568, -0.1883,  0.3589],
          [-0.4470,  0.1039, -0.3945],
          [-0.4190,  0.3723,  0.8384]],

         [[-0.0510,  0.5514, -0.2751],
          [-0.5668,  0.5824,  0.2328],
          [ 0.1316, -0.2518,  1.0418]],
```

Skip to main content

```
           [-0.1732,  0.5197,  1.1158],
           [ 0.3771, -0.3810,  0.2008]]]], grad_fn=<ConvolutionBackward0>), scale=tensor
```

[15]:
```
assert_with_message(out_tensor.is_valid)
```

```
True
```

What happens internally is that the input tensor passed to `input_quant_conv` is being quantized before being passed to the convolution operator. That means we are now computing a convolution between two quantized tensors, which mimplies that the output of the operation is also quantized. As expected then `out_tensor` is marked as valid.

Another important thing to notice is how the `bit_width` field of `out_tensor` is relatively high at *21 bits*. In Brevitas, the assumption is always that the output bit-width of an operator reflects the worst-case size of the *accumulator* required by that operation. In other terms, given the *size* of the input and weight tensors and their *bit-widths*, 21 is the bit-width that would be required to represent the largest possible output value that could be generated. This makes sure that the affine quantization invariant is always respected.

We could have obtained a similar result by directly passing as input a QuantTensor. In this example we are directly defining a QuantTensor ourselves, but it could also be the output of a previous layer

[16]:
```
from brevitas.quant_tensor import IntQuantTensor

scale = 0.0001
bit_width = 8
zero_point = 0.
int_value = torch.randint(low=- 2 ** (bit_width - 1), high=2 ** (bit_width - 1) - 1, si
quant_value = (int_value - zero_point) * scale
quant_tensor_input = IntQuantTensor(
    quant_value,
    scale=torch.tensor(scale),
    zero_point=torch.tensor(zero_point),
    bit_width=torch.tensor(float(bit_width)),
    signed=True,
    training=True)
quant_tensor_input
```

[16]:
```
IntQuantTensor(value=tensor([[[[ 7.2000e-03, -3.7000e-03,  7.7000e-03, -2.4000e-03, -8.
           [-1.2000e-02, -8.1000e-03,  7.2000e-03, -1.1300e-02, -9.7000e-03],
           [-1.0000e-03,  1.0100e-02,  3.8000e-03, -1.1900e-02,  6.9000e-03],
           [ 8.3000e-03,  1.0000e-04, -6.9000e-03,  3.9000e-03, -5.4000e-03],
           [ 1.1300e-02, -6.0000e-03,  9.7000e-03,  0.0000e+00,  1.0900e-02]],

          [[-1.0900e-02,  1.1400e-02, -6.4000e-03,  9.2000e-03,  7.1000e-03],
           [-6.0000e-04,  9.2000e-03, -8.5000e-03,  5.0000e-03,  6.5000e-03],
```

Skip to main content

```
        [-2.1000e-03,  9.5000e-03,  3.0000e-04, -2.9000e-03, -6.5000e-03],
        [-1.1800e-02, -4.8000e-03,  5.4000e-03, -2.5000e-03,  9.0000e-04]]]]), scale=
```

[17]:
```
assert_with_message(quant_tensor_input.is_valid)
```

```
True
```

Note: how we are explicitly forcing `value`, `scale`, `zero_point` and `bit_width` to be floating-point `torch.Tensor`, as this is expected by Brevitas but it's currently not enforced automatically at initialization time.

If we now pass in `quant_tensor_input` to `return_quant_conv`, we will see that indeed the output is a valid 21-bit `QuantTensor`:

[18]:
```
return_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False, return_quant_tensor=T
out_tensor = return_quant_conv(quant_tensor_input)
out_tensor
```

[18]:
```
IntQuantTensor(value=tensor([[[[-0.0019,  0.0049, -0.0012],
          [-0.0012,  0.0050, -0.0074],
          [-0.0023, -0.0035, -0.0033]],

         [[-0.0031,  0.0028,  0.0116],
          [ 0.0079,  0.0046,  0.0022],
          [ 0.0021, -0.0004,  0.0011]],

         [[-0.0045, -0.0010,  0.0002],
          [-0.0044,  0.0027,  0.0025],
          [-0.0009,  0.0040, -0.0044]]]], grad_fn=<ConvolutionBackward0>), scale=tensor
```

[19]:
```
assert_with_message(out_tensor.is_valid)
```

```
True
```

We can also pass in an input `QuantTensor` to a layer that has `input_quant` enabled. In that case, the input gets re-quantized:

[20]:
```
input_quant_conv(quant_tensor_input)
```

[20]:
```
IntQuantTensor(value=tensor([[[[-0.0073,  0.0040, -0.0011],
          [-0.0033,  0.0078, -0.0028],
          [ 0.0005, -0.0025, -0.0008]],

         [[ 0.0021, -0.0021,  0.0035],
          [ 0.0012, -0.0016, -0.0023],
          [-0.0010, -0.0015,  0.0040]],

         [[-0.0010,  0.0047,  0.0025],
```

Skip to main content

```
        [-0.0014,  0.0021, -0.0039],
        [ 0.0036, -0.0003,  0.0026]]]], grad_fn=<ConvolutionBackward0>), scale=tensor
```

Output Quantization

Let's now look at would have happened if we instead enabled output quantization:

[21]:
```
from brevitas.quant.scaled_int import Int8ActPerTensorFloat

output_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=False,
    output_quant=Int8ActPerTensorFloat, return_quant_tensor=True)
out_tensor = output_quant_conv(torch.randn(1, 2, 5, 5))
out_tensor
```

[21]:
```
IntQuantTensor(value=tensor([[[[-0.2117, -0.4811,  0.0385],
          [-0.5100, -0.2502, -0.2213],
          [-0.5773,  0.0192, -0.5485]],

         [[ 0.1347,  0.8179, -1.2316],
          [-0.6062,  0.4426, -0.3849],
          [ 0.1732, -0.5100, -0.1251]],

         [[ 1.0873,  0.2406, -0.2887],
          [-0.4330, -0.4907, -0.2021],
          [ 0.6447,  0.4811,  0.1347]]]], grad_fn=<MulBackward0>), scale=tensor(0.0096,
```

[22]:
```
assert_with_message(out_tensor.is_valid)
```

```
True
```

We can see again that the output is a valid `QuantTensor`. However, what happened internally is quite different from before.

Previously, we computed the convolution between two quantized tensors, and got a quantized tensor as output.

In this case instead, we compute the convolution between a quantized and an unquantized tensor, we take its unquantized output and we quantize it.

The difference is obvious once we look at the output `bit_width`. In the previous case, we had that the `bit_width` reflected the size of the output accumulator. In this case instead, we have `bit_width=tensor(8.)`, which is what we expected since `output_quant` had been set to an *Int8* quantizer.

Bias Quantization

There is an important scenario where the various options we just saw make a practical difference, and it's quantization of *bias*. In many contexts, such as in the ONNX standard opset and in FINN, bias is assumed to be quantized with scale factor equal to *input scale* weight scale*, which means that we need a valid quantized input somehow. A predefined bias quantizer that reflects that assumption is `brevitas.quant.scaled_int.Int8Bias`. If we simply tried to set it to a `QuantConv2d` without any sort of input quantization, we would get an error:

[23]:
```
from brevitas.quant.scaled_int import Int8Bias

bias_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    bias_quant=Int8Bias, return_quant_tensor=True)
bias_quant_conv(torch.randn(1, 2, 5, 5))
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[23], line 6
      1 from brevitas.quant.scaled_int import Int8Bias
      3 bias_quant_conv = QuantConv2d(
      4     in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
      5     bias_quant=Int8Bias, return_quant_tensor=True)
----> 6 bias_quant_conv(torch.randn(1, 2, 5, 5))

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
   1190 # If we don't have any hooks, we want to skip the rest of the logic in
   1191 # this function, and just call forward.
   1192 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks
   1193         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1194     return forward_call(*input, **kwargs)
   1195 # Do not call functions when jit is used
   1196 full_backward_hooks, non_full_backward_hooks = [], []

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    193 def forward(self, input: Union[Tensor, QuantTensor]) -> Union[Tensor, QuantTens
--> 194     return self.forward_impl(input)

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    148 compute_output_quant_tensor = isinstance(quant_input, QuantTensor) and isinstan
    149     quant_weight, QuantTensor)
    150 if not (compute_output_quant_tensor or
    151         self.output_quant.is_quant_enabled) and self.return_quant_tensor:
--> 152     raise RuntimeError("QuantLayer is not correctly configured")
    154 if self.bias is not None:
    155     quant_bias = self.bias_quant(self.bias, quant_input, quant_weight)

RuntimeError: QuantLayer is not correctly configured
```

Skip to main content

[24]:
```
bias_quant_conv(quant_tensor_input)
```

[24]:
```
IntQuantTensor(value=tensor([[[[-2.4238e-03, -5.6598e-03,  5.1882e-03],
          [-6.5582e-03,  8.9274e-03,  4.9640e-04],
          [ 9.6283e-03, -1.7466e-03, -4.8311e-03]],

         [[ 2.9322e-03, -3.1358e-03, -6.2727e-04],
          [ 2.8723e-06, -3.7981e-03,  1.0973e-02],
          [-4.1031e-03,  6.5909e-03, -4.2369e-03]],

         [[ 4.1967e-03, -7.0733e-03,  1.6456e-03],
          [ 1.8197e-03, -3.1683e-03,  4.8200e-03],
          [-3.2585e-04,  3.1055e-03,  1.9703e-03]]]],
       grad_fn=<ConvolutionBackward0>), scale=tensor([[[[1.7953e-07]]]], grad_fn=<MulBa
```

Or by enabling input quantization and then passing in a float a `torch.Tensor` or a `QuantTensor`:

[25]:
```
input_bias_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    input_quant=Int8ActPerTensorFloat, bias_quant=Int8Bias, return_quant_tensor=True)
input_bias_quant_conv(torch.randn(1, 2, 5, 5))
```

[25]:
```
IntQuantTensor(value=tensor([[[[-0.2816, -0.5271, -0.1748],
          [-0.4247, -0.1575,  0.0681],
          [ 0.6528, -0.5346, -0.0657]],

         [[ 0.2993, -0.3383,  0.3035],
          [-0.4595, -0.6796, -0.9720],
          [-0.1948, -0.5169, -0.2175]],

         [[ 0.5586,  0.0665, -0.5807],
          [ 0.5565,  0.1780, -0.0555],
          [-0.1080,  0.0791, -0.2262]]]], grad_fn=<ConvolutionBackward0>), scale=tensor
```

[26]:
```
input_bias_quant_conv(quant_tensor_input)
```

[26]:
```
IntQuantTensor(value=tensor([[[[-0.0058,  0.0030,  0.0030],
          [-0.0013, -0.0002,  0.0043],
          [-0.0061,  0.0033, -0.0001]],

         [[ 0.0013, -0.0008, -0.0015],
          [ 0.0011,  0.0012, -0.0012],
          [-0.0013, -0.0020,  0.0002]],

         [[-0.0061,  0.0053, -0.0004],
          [ 0.0028,  0.0031, -0.0037],
          [ 0.0027, -0.0048, -0.0044]]]], grad_fn=<ConvolutionBackward0>), scale=tensor
```

Notice how the output `bit_width=tensor(22.)`. This is because, in the worst-case, summing a *21-bit* integer (the size of the accumulator before bias is added) and an *8-bit* integer (the size of quantized bias) gives a *22-bit* integer.

Skip to main content

Let's try now to enable output quantization instead of input quantization. That wouldn't have solved the problem with bias quantization, as output quantization is performed after bias is added:

[27]:
```
output_bias_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    output_quant=Int8ActPerTensorFloat, bias_quant=Int8Bias, return_quant_tensor=True)
output_bias_quant_conv(torch.randn(1, 2, 5, 5))
```

```
/proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10/site
  warn(
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[27], line 4
      1 output_bias_quant_conv = QuantConv2d(
      2     in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
      3     output_quant=Int8ActPerTensorFloat, bias_quant=Int8Bias, return_quant_tenso
----> 4 output_bias_quant_conv(torch.randn(1, 2, 5, 5))

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
   1190 # If we don't have any hooks, we want to skip the rest of the logic in
   1191 # this function, and just call forward.
   1192 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks
   1193         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1194     return forward_call(*input, **kwargs)
   1195 # Do not call functions when jit is used
   1196 full_backward_hooks, non_full_backward_hooks = [], []

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    193 def forward(self, input: Union[Tensor, QuantTensor]) -> Union[Tensor, QuantTens
--> 194     return self.forward_impl(input)

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    152     raise RuntimeError("QuantLayer is not correctly configured")
    154 if self.bias is not None:
--> 155     quant_bias = self.bias_quant(self.bias, quant_input, quant_weight)
    156 else:
    157     quant_bias = None

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
   1190 # If we don't have any hooks, we want to skip the rest of the logic in
   1191 # this function, and just call forward.
   1192 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks
   1193         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1194     return forward_call(*input, **kwargs)
   1195 # Do not call functions when jit is used
   1196 full_backward_hooks, non_full_backward_hooks = [], []

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    362     input_scale = self.scale()
    363     if input_scale is None:
--> 364         raise RuntimeError("Input scale required")
```

Skip to main content

```
RuntimeError: Input scale required
```

Not all scenarios require bias quantization to depend on the scale factor of the input. In those cases, biases can be quantized the same way weights are quantized, and have their own scale factor. In Brevitas, a predefined quantizer that reflects this other scenario is `Int8BiasPerTensorFloatInternalScaling`. In this case then a valid quantized input is not required:

[28]:
```
from brevitas.quant.scaled_int import Int8BiasPerTensorFloatInternalScaling

bias_internal_scale_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    bias_quant=Int8BiasPerTensorFloatInternalScaling, return_quant_tensor=False)
bias_internal_scale_quant_conv(torch.randn(1, 2, 5, 5))
```

[28]:
```
tensor([[[[-0.4360, -0.2674, -0.4194],
          [-0.2412, -0.6360, -0.6838],
          [-0.5227, -0.0199, -0.1445]],

         [[-0.3524,  0.8025,  0.2844],
          [ 0.9945, -0.4782,  0.8064],
          [ 0.5732,  0.1249,  0.3110]],

         [[ 0.3223,  0.2530,  0.2753],
          [ 0.5764, -0.2533, -0.0181],
          [-0.4147,  0.2049, -0.9944]]]], grad_fn=<ConvolutionBackward0>)
```

There are a couple of situations to be aware of concerning bias quantization that can lead to changes in the output `zero_point`.

Let's consider the scenario where we compute the convolution between a quantized input tensor and quantized weights. In the first case, we then add an *unquantized* bias on top of the output. In the second one, we add a bias quantized with its own scale factor, e.g. with the `Int8BiasPerTensorFloatInternalScaling` quantizer. In both cases, in order to make sure the output `QuantTensor` is valid (i.e. the affine quantization invariant is respected), the output `zero_point` becomes non-zero:

[29]:
```
unquant_bias_input_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    input_quant=Int8ActPerTensorFloat, return_quant_tensor=True)
out_tensor = unquant_bias_input_quant_conv(torch.randn(1, 2, 5, 5))
out_tensor
```

[29]:
```
IntQuantTensor(value=tensor([[[[-0.6912,  0.0086,  0.1628],
          [-0.4786, -0.8073,  0.5224],
          [ 0.4157,  0.4686,  0.2560]],
```

Skip to main content

```
 [-0.1115,  0.6974, -0.0452]],

[[-0.6168, -0.5241, -0.6593],
 [ 0.6408,  0.2674,  0.4537],
 [-0.3744, -0.7771, -0.2848]]]], grad_fn=<ConvolutionBackward0>), scale=tensor

[[-4597.1797]],

[[-3452.3713]]]], grad_fn=<DivBackward0>), bit_width=tensor(21.), signed_t=ten
```

[30]:
```
assert_with_message(out_tensor.is_valid)
```

```
True
```

Finally, an important point about `QuantTensor`. With the exception of learned bit-width (which will be the subject of a separate tutorial) and some of the bias quantization scenarios we have just seen, usually returing a `QuantTensor` is not necessary and can create extra complexity. This is why currently `return_quant_tensor` defaults to `False`. We can easily see it in an example:

[31]:
```
bias_input_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), bias=True,
    input_quant=Int8ActPerTensorFloat, bias_quant=Int8Bias)
bias_input_quant_conv(torch.randn(1, 2, 5, 5))
```

[31]:
```
tensor([[[[-0.2327,  0.9267,  0.6294],
          [ 0.0901,  0.1027, -0.0727],
          [-0.5614,  0.6182,  0.5394]],

         [[ 0.4179, -0.5184, -0.2016],
          [ 0.1390, -0.3925, -0.6171]
```