An Overview of Quantized Activations

In this second tutorial, we take a deeper look at quantized activation.

We were already introduced to quantized activations in the previous tutorial, when we looked at input and output quantization of `QuantConv2d` with the `Int8ActPerTensorFloat` quantizer. The same result can be obtained with different syntax by coupling `QuantConv2d` with `QuantIdentity` layers, which by default uses the `Int8ActPerTensorFloat` quantizer. As an example, we compare - on the *same input* - the result of `QuantConv2d` with `output_quant` enabled with the result of a `QuantConv2d` followed by a `QuantIdentity`:

[1]:
```
# helpers
def assert_with_message(condition):
    assert condition
    print(condition)
```

[2]:
```
import torch
from brevitas.nn import QuantConv2d, QuantIdentity
from brevitas.quant.scaled_int import Int8ActPerTensorFloat

torch.manual_seed(0)  # set a seed to make sure the random weight init is reproducible
output_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3), output_quant=Int8ActPerTensorFloa

torch.manual_seed(0)  # reproduce the same random weight init as above
default_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3))
output_identity_quant = QuantIdentity()

inp = torch.randn(1, 2, 5, 5)
out_tensor1 = output_quant_conv(inp)
out_tensor2 = output_identity_quant(default_quant_conv(inp))

assert_with_message(out_tensor1.isclose(out_tensor2).all().item())
```

```
True

/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.
```

```
[3]: torch.manual_seed(0)
input_output_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3),
    input_quant=Int8ActPerTensorFloat, output_quant=Int8ActPerTensorFloat)

torch.manual_seed(0)
default_quant_conv = QuantConv2d(
    in_channels=2, out_channels=3, kernel_size=(3,3))
input_identity_quant = QuantIdentity()
output_identity_quant = QuantIdentity()

inp = torch.randn(1, 2, 5, 5)
out_tensor1 = input_output_quant_conv(inp)
out_tensor2 = output_identity_quant(default_quant_conv(input_identity_quant(inp)))

assert_with_message(out_tensor1.isclose(out_tensor2).all().item())
```

```
True
```

From an algorithmic point of view then the two different implementation are doing the same thing. However, as it will become clearer in later tutorials, there are currently some scenarios where picking one style over the other can make a difference when it comes to exporting to a format such as standard ONNX. In the meantime, we can just keep in mind that both alternatives exist.

As it was the case with `QuantConv2d`, when we disable quantization of an activation, the layer behaves as its floating-point variant. In the case of `QuantIdentity`, that means behaving like an identity function:

```
[4]: disabled_quant_identity = QuantIdentity(act_quant=None)
assert_with_message((inp == disabled_quant_identity(inp)).all().item())
```

```
True
```

Again, as it was the case for `QuantConv2d`, quantized activation layers can also return a `QuantTensor`:

```
[5]: return_quant_identity = QuantIdentity(return_quant_tensor=True)
out_tensor = return_quant_identity(inp)
out_tensor
```

```
[5]: IntQuantTensor(value=tensor([[[[-0.4566, -0.5707, -0.5517,  0.5897,  1.5409],
          [ 0.5136, -0.5897, -0.5707,  0.1902, -0.0761],
          [-0.4946, -1.5029, -0.1902,  0.4376,  1.3317],
          [-1.6361,  2.0736,  1.7122,  2.3780, -1.1224],
          [-0.3234, -1.0844, -0.0761, -0.0951, -0.7610]],

         [[-1.5980,  0.0190, -0.7419,  0.1902,  0.6278],
          [ 0.6468, -0.2473, -0.5327,  1.1605,  0.4376],
          [-0.7990, -1.2936, -0.7419, -1.3127, -0.2283],
```

Skip to main content

```
          [-0.3615, -1.2175, -0.6278, -0.4566,  1.9214]]]],
       grad_fn=<MulBackward0>), scale=tensor(0.0190, grad_fn=<AbsBinarySignGradFnBackwa
```

[6]:
```python
assert_with_message(out_tensor.is_valid)
```

```
True
```

As expected, a `QuantIdentity` with quantization disabled behaves like an identity function also when a `QuantTensor` is passed in. However, depending on whather `return_quant_tensor` is set to `False` or not, quantization metadata might be stripped out, i.e. the input `QuantTensor` is going to be returned as an implicitly quantized `torch.Tensor`:

[7]:
```python
out_torch_tensor = disabled_quant_identity(out_tensor)
out_torch_tensor
```

[7]:
```
tensor([[[[-0.4566, -0.5707, -0.5517,  0.5897,  1.5409],
          [ 0.5136, -0.5897, -0.5707,  0.1902, -0.0761],
          [-0.4946, -1.5029, -0.1902,  0.4376,  1.3317],
          [-1.6361,  2.0736,  1.7122,  2.3780, -1.1224],
          [-0.3234, -1.0844, -0.0761, -0.0951, -0.7610]],

         [[-1.5980,  0.0190, -0.7419,  0.1902,  0.6278],
          [ 0.6468, -0.2473, -0.5327,  1.1605,  0.4376],
          [-0.7990, -1.2936, -0.7419, -1.3127, -0.2283],
          [-2.4351, -0.0761,  0.2283,  0.7990, -0.1902],
          [-0.3615, -1.2175, -0.6278, -0.4566,  1.9214]]]],
       grad_fn=<AliasBackward0>)
```

[8]:
```python
return_disabled_quant_identity = QuantIdentity(act_quant=None, return_quant_tensor=True
identity_out_tensor = return_disabled_quant_identity(out_tensor)
identity_out_tensor
```

[8]:
```
IntQuantTensor(value=tensor([[[[-0.4566, -0.5707, -0.5517,  0.5897,  1.5409],
          [ 0.5136, -0.5897, -0.5707,  0.1902, -0.0761],
          [-0.4946, -1.5029, -0.1902,  0.4376,  1.3317],
          [-1.6361,  2.0736,  1.7122,  2.3780, -1.1224],
          [-0.3234, -1.0844, -0.0761, -0.0951, -0.7610]],

         [[-1.5980,  0.0190, -0.7419,  0.1902,  0.6278],
          [ 0.6468, -0.2473, -0.5327,  1.1605,  0.4376],
          [-0.7990, -1.2936, -0.7419, -1.3127, -0.2283],
          [-2.4351, -0.0761,  0.2283,  0.7990, -0.1902],
          [-0.3615, -1.2175, -0.6278, -0.4566,  1.9214]]]],
       grad_fn=<AliasBackward0>), scale=tensor(0.0190, grad_fn=<AbsBinarySignGradFnBack
```

Moving on from `QuantIdentity`, let's take a look at `QuantReLU`. Anything we said so far about `QuantIdentity` also applies to `QuantReLU`. The difference though is that `QuantReLU` implements a ReLU function followed by quantization, while `QuantIdentity` is really just the quantization operator. Additionally, by default `QuantReLU` adopts the `Uint8ActPerTensorFloat`, meaning that the

Skip to main content

```
[9]: from brevitas.nn import QuantReLU

return_quant_relu = QuantReLU(return_quant_tensor=True)
return_quant_relu(inp)
```

```
[9]: IntQuantTensor(value=tensor([[[[0.0000, 0.0000, 0.0000, 0.5974, 1.5402],
          [0.5041, 0.0000, 0.0000, 0.1867, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.4481, 1.3255],
          [0.0000, 2.0817, 1.7083, 2.3804, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0187, 0.0000, 0.1867, 0.6254],
          [0.6348, 0.0000, 0.0000, 1.1668, 0.4387],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.2334, 0.7935, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 1.9230]]]], grad_fn=<MulBackward0>), scale=t
```

`QuantReLU`, like `QuantIdentity`, is also special compared to other non-linear quantized activation layers as it preserves the metadata of an input `QuantTensor` even when quantization is disabled:

```
[10]: return_disabled_quant_relu = QuantReLU(act_quant=None, return_quant_tensor=True)
relu_out_tensor = return_disabled_quant_relu(out_tensor)
assert_with_message(relu_out_tensor.is_valid)
assert_with_message(relu_out_tensor.scale == out_tensor.scale)
assert_with_message(relu_out_tensor.zero_point == out_tensor.zero_point)
assert_with_message(relu_out_tensor.bit_width == out_tensor.bit_width)
```

```
True
tensor(True)
tensor(True)
tensor(True)
```

That doesn't apply to other layers like, say, `QuantSigmoid`:

```
[11]: from brevitas.nn import QuantSigmoid

return_disabled_quant_sigmoid = QuantSigmoid(act_quant=None, return_quant_tensor=True)
sigmoid_out_tensor = return_disabled_quant_sigmoid(out_tensor)
sigmoid_out_tensor
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[11], line 4
      1 from brevitas.nn import QuantSigmoid
      3 return_disabled_quant_sigmoid = QuantSigmoid(act_quant=None, return_quant_tenso
----> 4 sigmoid_out_tensor = return_disabled_quant_sigmoid(out_tensor)
      5 sigmoid_out_tensor

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.
```

```
-> 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
     51     return out
     52 out = self.act_quant(quant_input)
---> 53 out = self.pack_output(out)
     54 return out

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     99 def pack_output(self, quant_output: Union[Tensor, QuantTensor]) -> Union[Tensor
    100     if self.return_quant_tensor:
--> 101         assert isinstance(quant_output, QuantTensor), 'QuantLayer is not correc
    102         return quant_output
    103     else:

AssertionError: QuantLayer is not correctly configured, check if warnings were raised
```

Something to always keep in mind is that the non-linearity of a quantized activation layer is always called on the *dequantized* representation of the input. For example, let's say we first quantize a floating-point `torch.Tensor` with an unsigned shifted quantizer such as `ShiftedUint8ActPerTensorFloat`, i.e. with zero-point such that the integer representation of its output is non-negative. Then, we pass this tensor as input to a `QuantReLU` with quantization *disabled*. The fact that the input to `QuantReLU` in its integer form is unsigned doesn't mean `QuantReLU` won't have any effect, as ReLU is called on the dequantized representation, which includes both *positive* and *negative* values:

[12]:
```
from brevitas.quant.shifted_scaled_int import ShiftedUint8ActPerTensorFloat

shifted_quant_identity = QuantIdentity(act_quant=ShiftedUint8ActPerTensorFloat, return_
return_disabled_quant_relu = QuantReLU(act_quant=None, return_quant_tensor=True)
return_disabled_quant_relu(shifted_quant_identity(inp))
```

[12]:
```
IntQuantTensor(value=tensor([[[[0.0000, 0.0000, 0.0000, 0.5854, 1.5485],
          [0.5099, 0.0000, 0.0000, 0.1888, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.4532, 1.3219],
          [0.0000, 2.0772, 1.6996, 2.3794, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0189, 0.0000, 0.1888, 0.6232],
          [0.6421, 0.0000, 0.0000, 1.1708, 0.4343],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.2266, 0.7931, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 1.9262]]]], grad_fn=<ReluBackward0>), scale=
```

Let's now consider the very common scenario of a `QuantConv2d` followed by a `ReLU` or `QuantReLU`. In particular, let's say we have a `QuantConv2d` with output quantization *enabled* followed by a `ReLU`:

Skip to main content

```
[13]: torch.manual_seed(0)
      output_quant_conv = QuantConv2d(
          in_channels=2, out_channels=3, kernel_size=(3,3), output_quant=Int8ActPerTensorFloa
      torch.relu(output_quant_conv(inp))
```

```
[13]: tensor([[[[0.0000, 0.0000, 0.0000],
                [1.3134, 1.2557, 1.0392],
                [0.4186, 0.0000, 0.0000]],

               [[0.7361, 0.5340, 0.8516],
                [0.2887, 0.3175, 0.0000],
                [0.8949, 1.6743, 0.0722]],

               [[0.0000, 0.0000, 0.0289],
                [0.0000, 0.0000, 0.2021],
                [0.0000, 0.0000, 0.4907]]]], grad_fn=<ReluBackward0>)
```

We compare it against a `QuantConv2d` with default settings (i.e. output quantization *disabled*), followed by a `QuantReLU` with default settings (i.e. activation quantization *enabled*):

```
[14]: torch.manual_seed(0)
      default_quant_conv = QuantConv2d(
          in_channels=2, out_channels=3, kernel_size=(3,3))
      default_quant_relu = QuantReLU()
      default_quant_relu(default_quant_conv(inp))
```

```
[14]: tensor([[[[0.0000, 0.0000, 0.0000],
                [1.3078, 1.2555, 1.0397],
                [0.4185, 0.0000, 0.0000]],

               [[0.7454, 0.5427, 0.8566],
                [0.2943, 0.3269, 0.0000],
                [0.8893, 1.6674, 0.0785]],

               [[0.0065, 0.0000, 0.0262],
                [0.0000, 0.0000, 0.1962],
                [0.0000, 0.0000, 0.4839]]]], grad_fn=<MulBackward0>)
```

We can see the results are close but not quite the same.
In the first case, we quantized the output of `QuantConv2d` with an 8-bit signed quantizer, and then we passed it through a `ReLU`, meaning that half of the numerical range covered by the signed quantizer is now lost, and by all practical means the output can now be treated as a 7-bit unsigned number (although it's not explicitly marked as such). In the second case, we perform unsigned 8-bit quantization after `ReLU`. Because the range covered by the quantizer now includes only non-negative numbers, we don't waste a bit as in the previous case.

Regarding some premade activation quantizers, such as `Uint8ActPerTensorFloat`, `ShiftedUint8ActPerTensorFloat`, and `Int8ActPerTensorFloat`, a word of caution that anticipates

Skip to main content

zero-point by collecting statistics for a number of training steps (by default 30). This can be seen as a sort of very basic calibration step, although it typically happens during training and with quantization already enabled. These statistics are accumulated in an exponential moving average that at end of the collection phase is used to initialize a learned *parameter*. During the collection phase then, the quantizer behaves differently between `train()` and `eval()` mode. In `train()` mode, the statistics for that particular batch are returned. In `eval()` mode, the exponential moving average is returned. After the collection phase is over the learned parameter is returned in both execution modes. We can easily observe this behaviour with an example. Let's first define a quantized activation and two random input tensors:

[15]:
```
quant_identity = QuantIdentity(return_quant_tensor=True)
inp1 = torch.randn(3, 3)
inp2 = torch.randn(3, 3)
```

We then compare the output scale factor of the two tensors between `train()` and `eval()` mode. The ones in train mode in general are different. The ones in eval mode are the same.

[16]:
```
out1_train = quant_identity(inp1)
out2_train = quant_identity(inp2)
assert_with_message(not out1_train.scale.isclose(out2_train.scale).item())
```

```
True
```

[17]:
```
quant_identity.eval()
out1_eval = quant_identity(inp1)
out2_eval = quant_identity(inp2)
assert_with_message(out1_eval.scale.isclose(out2_eval.scale).item())
```

```
True
```

By default, the only layer that is an exception to this is `QuantHardTanh`. That is because the interface to `torch.nn.HardTanh` already requires users to manually specify `min_val` and `max_val`, so Brevitas preserves that both when quantization is enabled or disabled. With quantization enabled, by default those values are used for initialization, but then the range is learned. Let's look at an example. Run the cell below, and we expect it to throw an error because of missing attributes:

[18]:
```
from brevitas.nn import QuantHardTanh

QuantHardTanh()
```

```
---------------------------------------------------------------------------
DependencyError                           Traceback (most recent call last)
Cell In[18], line 3
      1 from brevitas.nn import QuantHardTanh
----> 3 QuantHardTanh()
```

Skip to main content

```
     90 def __init__(
     91         self,
     92         act_quant: Optional[ActQuantType] = Int8ActPerTensorFloatMinMaxInit,
     93         input_quant: Optional[ActQuantType] = None,
     94         return_quant_tensor: bool = False,
     95         **kwargs):
---> 96     QuantNLAL.__init__(
     97         self,
     98         act_impl=nn.Hardtanh,
     99         passthrough_act=True,
    100         input_quant=input_quant,
    101         act_quant=act_quant,
    102         return_quant_tensor=return_quant_tensor,
    103         **kwargs)

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     32 QuantLayerMixin.__init__(self, return_quant_tensor)
     33 QuantInputMixin.__init__(self, input_quant, **kwargs)
---> 34 QuantNonLinearActMixin.__init__(self, act_impl, passthrough_act, act_quant, **k

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     55 def __init__(
     56         self,
     57         act_impl: Optional[Type[Module]],
   (...)
     61         act_kwargs_prefix='',
     62         **kwargs):
     63     prefixed_kwargs = {
     64         act_kwargs_prefix + 'act_impl': act_impl,
     65         act_kwargs_prefix + 'passthrough_act': passthrough_act}
---> 66     QuantProxyMixin.__init__(
     67         self,
     68         quant=act_quant,
     69         proxy_prefix=act_proxy_prefix,
     70         kwargs_prefix=act_kwargs_prefix,
     71         proxy_protocol=ActQuantProxyProtocol,
     72         none_quant_injector=NoneActQuant,
     73         **prefixed_kwargs,
     74         **kwargs)

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     50     quant_injector = quant
     51     quant_injector = quant_injector.let(**filter_kwargs(kwargs_prefix, kwargs))
---> 52     quant = quant_injector.proxy_class(self, quant_injector)
     53 else:
     54     if not isinstance(quant, proxy_protocol):

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    220 def __init__(self, quant_layer, quant_injector):
--> 221     super().__init__(quant_layer, quant_injector)
    222     self.cache_class = _CachedIO

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     93 def __init__(self, quant_layer, quant_injector):
---> 94     QuantProxyFromInjector.__init__(self, quant_layer, quant_injector)
```

Skip to main content

```
File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
     78 # Use a normal list and not a ModuleList since this is a pointer to parent modu
     79 self.tracked_module_list = []
---> 80 self.add_tracked_module(quant_layer)
     81 self.disable_quant = False
     82 # Torch.compile compatibility requires this

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    118     self.tracked_module_list.append(module)
    119     self.update_tracked_modules()
--> 120     self.init_tensor_quant()
    121 else:
    122     raise RuntimeError("Trying to add None as a parent module.")

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    148 def init_tensor_quant(self):
--> 149     tensor_quant = self.quant_injector.tensor_quant
    150     if 'act_impl' in self.quant_injector:
    151         act_impl = self.quant_injector.act_impl

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10
    126     else:
    127         message = "{!r} can not resolve attribute {!r}".format(
    128             cls.__name__, current_attr)
--> 129     raise DependencyError(message)
    131 marker, attribute, args, have_defaults = spec
    133 if set(args).issubset(cached):

DependencyError: 'Int8ActPerTensorFloatMinMaxInit' can not resolve attribute 'max_val
```

As expected, we get an error concering a missing `max_val` attribute. Let's try to pass it then, together with `min_val`:

```
[19]: quant_hard_tanh = QuantHardTanh(max_val=1.0, min_val=-1.0, return_quant_tensor=True)
```

The layer is now correctly initialized. We can see that the output scale factors are all the same between `train()` and `eval()` mode:

```
[20]: out1_train = quant_hard_tanh(inp1)
      quant_hard_tanh.eval()
      out2_eval = quant_hard_tanh(inp2)
      assert_with_message(out1_train.scale.isclose(out2_eval.scale).item())
```

```
True
```

In all of the examples that have currently been looked at in this tutorial, we have used per-tensor quantization. I.e., the output tensor of the activation, if quantized, was always quantized on a per-tensor level, with a single scale and zero-point quantization parameter per output tensor. However, one can also do per-channel quantization, where each output channel of the tensor has its own tensor that has 3 channels and 256 elements in the height and width dimensions. We purposely mutate the 1st channel to have its dynamic range be 3 times larger than the other 2 channels. We then feed it through a `QuantReLU`, whose default behavior is to quantize at a per-tensor granularity.

[21]:
```
out_channels = 3
inp3 = torch.rand(1, out_channels, 256, 256)  # (B, C, H, W)
inp3[:, 0, :, :] *= 3

per_tensor_quant_relu = QuantReLU(return_quant_tensor=True)
out_tensor = per_tensor_quant_relu(inp3)
out_tensor.scale * ((2**8) -1)
```

[21]:
```
tensor(2.9998, grad_fn=<MulBackward0>)
```

We can see that the per-tensor scale parameter has calibrated itself to provide a full quantization range of 3, matching that of the channel with the largest dynamic range.

We can take a look at the `QuantReLU` object, and in particular look at what the `scaling_impl` object is composed of. It is responsible for gathering statistics for determining the quantization parameters, and we can see that its `stats_input_view_shape_impl` attribute is set to be an instance of `OverTensorView`. This is defined here, and serves to flatten out the observed tensor into a 1D tensor and, in this case, use the `AbsPercentile` observer to calculate the quantization parameters during the gathering statistics stage of QAT.

[22]:
```
per_tensor_quant_relu
```

[22]:
```
QuantReLU(
  (input_quant): ActQuantProxyFromInjector(
    (_zero_hw_sentinel): StatelessBuffer()
  )
  (act_quant): ActQuantProxyFromInjector(
    (_zero_hw_sentinel): StatelessBuffer()
    (fused_activation_quant_proxy): FusedActivationQuantProxy(
      (activation_impl): ReLU()
      (tensor_quant): RescalingIntQuant(
        (int_quant): IntQuant(
          (float_to_int_impl): RoundSte()
          (tensor_clamp_impl): TensorClamp()
          (delay_wrapper): DelayWrapper(
            (delay_impl): _NoDelay()
          )
          (input_view_impl): Identity()
        )
        (scaling_impl): ParameterFromRuntimeStatsScaling(
          (stats_input_view_shape_impl): OverTensorView()
          (stats): _Stats(
            (stats_impl): AbsPercentile()
          )
```

```
                )
                (restrict_threshold): _RestrictValue(
                  (restrict_value_impl): FloatRestrictValue()
                )
                (clamp_scaling): _ClampValue(
                  (clamp_min_ste): ScalarClampMinSte()
                )
                (restrict_inplace_preprocess): Identity()
                (restrict_scaling_pre): Identity()
                (restrict_threshold_pre): Identity()
              )
              (int_scaling_impl): IntScaling()
              (zero_point_impl): ZeroZeroPoint(
                (zero_point): StatelessBuffer()
              )
              (msb_clamp_bit_width_impl): BitWidthConst(
                (bit_width): StatelessBuffer()
              )
            )
          )
        )
      )
```

Next, we initialise a new `QuantReLU` instance, but this time we specify that we desire per-channel quantization i.e. `scaling_per_output_channel=True`. This will implictly call `scaling_stats_input_view_shape_impl`, defined here, and will change the `QuantReLU` from using a per-tensor view when gathering stats to a per output channel view (`` `OverOutputChannelView ``
<Xilinx/brevitas>`__). This simply permutes the tensor into a 2D tensor, with dim 0 equal to the number of output channels.

To accomplish this, we also need to give it some extra information: `scaling_stats_permute_dims` and `per_channel_broadcastable_shape`. `scaling_stats_permute_dims` is responsible for defining how we do the permutation. `per_channel_broadcastable_shape` is necessary to understand along which dimensions the scale factor has to be broadcasted, so that the scale factor values are applied along the channel dimensions of the input. By default, PyTorch will broadcast along the first rightmost dimension for which the shapes of the two tensors match. To make sure that we apply the scale factor in our desired output channel dimension, we need to tell PyTorch how to correctly broadcast the scale factors. Therefore the scale factor will have as many dimensions as the input tensors, with all the shapes equal to 1 apart from the channel dimension.

Below, we can see that in the per-channel `QuantReLU` instance, the `stats_input_view_shape_impl` is now `OverOutputChannelView`, and uses a `PermuteDims` instance to do the permutation of the tensor to, in this case, a 2D tensor.

```
[23]: per_chan_quant_relu = QuantReLU(return_quant_tensor=True,
```

```
                                   scaling_stats_permute_dims=(1, 0, 2, 3),
                                   )
per_chan_quant_relu
```

[23]:
```
QuantReLU(
  (input_quant): ActQuantProxyFromInjector(
    (_zero_hw_sentinel): StatelessBuffer()
  )
  (act_quant): ActQuantProxyFromInjector(
    (_zero_hw_sentinel): StatelessBuffer()
    (fused_activation_quant_proxy): FusedActivationQuantProxy(
      (activation_impl): ReLU()
      (tensor_quant): RescalingIntQuant(
        (int_quant): IntQuant(
          (float_to_int_impl): RoundSte()
          (tensor_clamp_impl): TensorClamp()
          (delay_wrapper): DelayWrapper(
            (delay_impl): _NoDelay()
          )
          (input_view_impl): Identity()
        )
        (scaling_impl): ParameterFromRuntimeStatsScaling(
          (stats_input_view_shape_impl): OverOutputChannelView(
            (permute_impl): PermuteDims()
          )
          (stats): _Stats(
            (stats_impl): AbsPercentile()
          )
          (restrict_scaling): _RestrictValue(
            (restrict_value_impl): FloatRestrictValue()
          )
          (restrict_threshold): _RestrictValue(
            (restrict_value_impl): FloatRestrictValue()
          )
          (clamp_scaling): _ClampValue(
            (clamp_min_ste): ScalarClampMinSte()
          )
          (restrict_inplace_preprocess): Identity()
          (restrict_scaling_pre): Identity()
          (restrict_threshold_pre): Identity()
        )
        (int_scaling_impl): IntScaling()
        (zero_point_impl): ZeroZeroPoint(
          (zero_point): StatelessBuffer()
        )
        (msb_clamp_bit_width_impl): BitWidthConst(
          (bit_width): StatelessBuffer()
        )
      )
    )
  )
)
```

We can also observe the effect on the quantization parameters:

Skip to main content

```
out_channel = per_chan_quant_relu(inp3)
out_channel.scale * ((2**8) -1)
```

```
[24]: tensor([[[[2.9998]],

              [[1.0000]],

              [[1.0000]]]], grad_fn=<MulBackward0>)
```

We can see that the number of elements in the quantization scale of the outputted tensor is now 3, matching those of the 3-channel tensor! Furthermore, we see that each channel has an 8-bit quantization range that matches its data distribution, which is much more ideal in terms of reducing quantization mismatch. However, it's important to note that some hardware providers don't efficiently support per-channel quantization in production, so it's best to check if your targetted hardware will allow per-channel quantization.

Finally, a reminder that mixing things up is perfectly legal and encouraged in Brevitas. For example, a `QuantIdentity` with `act_quant=Int8ActPerTensorFloatMinMaxInit` is equivalent to a default `QuantHardTanh`, or conversely a `QuantHardTanh` with `act_quant=Int8ActPerTensorFloat` is equivalent to a default `QuantIdentity`. This is allowed by the fact that - as it will be explained in the next tutorial - the same layer can accept different keyword arguments when different quantizers are set. So a QuantIdentity with `act_quant=Int8ActPerTensorFloatMinMaxInit` is going to expect