# Brevitas TVMCon 2021 tutorial #

Brevitas requires Python 3.8+ and PyTorch 1.5.0+ and can be installed from PyPI with `pip install brevitas`. This notebook was originally released for Brevitas 0.7.1. It has since been updated to run with Brevitas 0.8 and PyTorch 1.9.0 and is going to be available at `https://github.com/Xilinx/brevitas/tree/master/notebooks`.

# Fundamentals

## QuantLinear layer

We start by looking at `brevitas.nn.QuantLinear`, a quantized alternative to `torch.nn.Linear`, and an instance of a `QuantWeightBiasInputOutputLayer`, meaning that it supports quantization of its weight, bias, input and output. Other instances of `QuantWBIOL` are `QuantConv1d`, `QuantConv2d`, `QuantConvTranspose1d` and `QuantConvTranspose2d`, and they all follow the same principles.

[1]:
```
import inspect
from brevitas.nn import QuantLinear
from IPython.display import Markdown, display

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

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

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

```
def __init__(
        self,
        in_features: int,
        out_features: int,
        bias: Optional[bool] = True
```

Skip to main content

```
        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:
    Linear.__init__(self, in_features, out_features, 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)
```

By default `weight_quant=Int8WeightPerTensorFloat`, while `bias_quant`, `input_quant` and `output_quant` are set to `None`, meaning that by default weight quantization is enabled, while everything else is disabled.

# Weight quantization

## Default weight quantization

`weight_quant=Int8WeightPerTensorFloat` means that by default weights are quantized to *8-bit signed integer with a per-tensor floating-point scale factor*, while quantization of bias, input, and output are disabled.

We can instantiate a layer and inspect the original weight tensor and the quantized version to see the effect:

[2]:
```
import torch

torch.manual_seed(0)

quant_linear = QuantLinear(2, 4, bias=True)

print(f"Original float weight tensor:\n {quant_linear.weight} \n")
print(f"Quantized weight QuantTensor:\n {quant_linear.quant_weight()} \n")
```

```
Original float weight tensor:
 Parameter containing:
tensor([[-0.0053,  0.3793],
        [-0.5820, -0.5204],
```

Skip to main content

```
Quantized weight QuantTensor:
 IntQuantTensor(value=tensor([[-0.0046,  0.3803],
        [-0.5820, -0.5224],
        [-0.2704,  0.1879],
        [-0.0137,  0.5591]], grad_fn=<MulBackward0>), scale=0.004582525696605444, zero_
```

As it can be noticed, by default the scale factor is differentiable and it's computed based on the maximum absolute value found within the full precision weight tensor.

We can also retrieve the corrisponding integer representation:

[3]:
```
print(f"Quantized Weight integer tensor:\n {quant_linear.quant_weight().int()}")
```

```
Quantized Weight integer tensor:
 tensor([[  -1,   83],
        [-127, -114],
        [ -59,   41],
        [  -3,  122]], dtype=torch.int8)
```

# Mixing quantized weights and floating-point inputs

In any of the layers defined so far we enabled only weight quantization. So by passing in a float input we would get in general an float output, with the linear operation being computed between the unquantized input and the dequantized weights:

[4]:
```
torch.manual_seed(0)

float_input = torch.randn(3, 2)
quant_linear = QuantLinear(2, 4, bias=False)
float_output = quant_linear(float_input)

print(f"Float input:\n {float_input}")
print(f"Float output:\n {float_output}")
```

```
Float input:
 tensor([[ 1.5410, -0.2934],
        [-2.1788,  0.5684],
        [-1.0845, -1.3986]])
Float output:
 tensor([[-0.9036, -0.4586,  0.3096, -0.6472],
        [ 1.2058,  0.6525, -0.3723,  0.8677],
        [ 1.3873,  0.2801, -0.9009,  0.9507]], grad_fn=<MmBackward0>)

/proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.10/site
  return super(Tensor, self).rename(names)
```

In general operations involving quantized tensors are always computed through standard torch operators (here `torch.nn.functional.linear` called internally by the module) on the dequantized representation, the so-called fake-quantization approach.

## Fixed-point weight quantization

Brevitas exposes various pre-made quantizers. For example, for pure fixed point quantization (i.e. restricting the scale to a power of two) we can use `Int8WeightPerTensorFixedPoint`:

[5]:
```
torch.manual_seed(0)

import math
from brevitas.quant import Int8WeightPerTensorFixedPoint

quant_linear = QuantLinear(2, 4, weight_quant=Int8WeightPerTensorFixedPoint, bias=False

print(f"Weight QuantTensor:\n {quant_linear.quant_weight()}")
print(f"Weight fix point: {- math.log2(quant_linear.quant_weight().scale)}")
```

```
Weight QuantTensor:
 IntQuantTensor(value=tensor([[-0.0078,  0.3828],
        [-0.5781, -0.5234],
        [-0.2734,  0.1875],
        [-0.0156,  0.5625]], grad_fn=<MulBackward0>), scale=0.0078125, zero_point=0.0,
Weight fix point: 7.0
```

## Binary weight quantization

For binary quantization with a constant scale factor (defaults 0.1) we can set the `SignedBinaryWeightPerTensorConst` quantizer:

[6]:
```
torch.manual_seed(0)

from brevitas.quant import SignedBinaryWeightPerTensorConst

quant_linear = QuantLinear(2, 4, weight_quant=SignedBinaryWeightPerTensorConst, bias=Fa

print(f"Weight QuantTensor:\n {quant_linear.quant_weight()}")
```

```
Weight QuantTensor:
 IntQuantTensor(value=tensor([[-0.1000,  0.1000],
        [-0.1000, -0.1000],
```

Skip to main content

```
        [-0.1000,  0.1000],
        [-0.1000,  0.1000]], grad_fn=<MulBackward0>), scale=tensor([0.1000]), zero_poin
```

## Sharing a weight quantizer

Brevitas also allows to share **instances** (not definitions) of weight quantizers among layers. This is different from applying the same quantization strategy to multiple layers, and it forces them to have the same scale, zero-point, and bit-width. It can be handy in a variety of situations as we will see later, and it works across different quantization strategies.

It's designed to operate in an eager-mode friendly way, meaning that the number of layers sharing it doesn't have to be known in advance. The quantizer is re-initialized appropriately any time it's shared to a new layer.

For example, we mentioned that the default weight quantizer `Int8WeightPerTensorFloat` computes the scale factor based on the maximum value found within the floating-point weight tensor to quantize. Sharing it among multiple layers means that the quantizer now looks at all the weight tensors that are being quantized to determine the overall maximum value and generate a single scale factor:

[7]:
```
torch.manual_seed(0)

# Define a QuantLinear layer 1
quant_linear1 = QuantLinear(2, 4, bias=False)

# Keep a pointer to the scale factor of QuantLinear layer 1 weights before sharing
quant_linear1_scale_before_sharing = quant_linear1.quant_weight().scale

# Define a QuantLinear layer 2 where the weight quantizer is taken from layer 1
quant_linear2 = QuantLinear(2, 4, weight_quant=quant_linear1.weight_quant, bias=False)

print(f"QuantLinear 1 scale before sharing with QuantLinear 2: {quant_linear1_scale_bef
print(f"QuantLinear 2 scale: {quant_linear2.quant_weight().scale:.4f}")
print(f"QuantLinear 1 scale after sharing with QuantLinear 2: {quant_linear1.quant_weig
```

```
QuantLinear 1 scale before sharing with QuantLinear 2: 0.0046
QuantLinear 2 scale: 0.0053
QuantLinear 1 scale after sharing with QuantLinear 2: 0.0053
```

Let's say now we want to quantize the input to `QuantLinear` to generate a quantized output. We can set an appropriate input quantizer like `Int8ActPerTensorFloat`:

[8]:
```
torch.manual_seed(0)

from brevitas.quant import Int8ActPerTensorFloat

float_input = torch.randn(3, 2)
quant_linear = QuantLinear(2, 4, input_quant=Int8ActPerTensorFloat, bias=False)

quant_output = quant_linear(float_input)

print(f"Float input:\n {float_input} \n")
print(f"Quant output:\n {quant_output}")
```

```
Float input:
 tensor([[ 1.5410, -0.2934],
        [-2.1788,  0.5684],
        [-1.0845, -1.3986]])

Quant output:
 tensor([[-0.9109, -0.4609,  0.3135, -0.6523],
        [ 1.2089,  0.6524, -0.3752,  0.8697],
        [ 1.3893,  0.2816, -0.9011,  0.9521]], grad_fn=<MmBackward0>)
```

Note how by default the output tensor is returned as a standard torch tensor in dequantized format. This is designed to minimize friction with standard torch operators in the default case. To return a output `QuantTensor` we have to set `return_quant_tensor=True`:

[9]:
```
torch.manual_seed(0)

from brevitas.quant import Int8ActPerTensorFloat

float_input = torch.randn(3, 2)
quant_linear = QuantLinear(2, 4, input_quant=Int8ActPerTensorFloat, bias=False, return_

quant_output = quant_linear(float_input)

print(f"Quant output:\n {quant_output}")
```

```
Quant output:
 IntQuantTensor(value=tensor([[-0.9109, -0.4609,  0.3135, -0.6523],
        [ 1.2089,  0.6524, -0.3752,  0.8697],
        [ 1.3893,  0.2816, -0.9011,  0.9521]], grad_fn=<MmBackward0>), scale=tensor([[9
```

# QuantIdentity layer

Skip to main content

Alternatively we could have instantiated a `QuantIdentity` layer. By default `QuantIdentity` adopts the `Int8ActPerTensorFloat` quantizer:

[10]:
```
torch.manual_seed(0)

from brevitas.nn import QuantIdentity

float_input = torch.randn(3, 2)
quant_identity = QuantIdentity(return_quant_tensor=True)
quant_linear = QuantLinear(2, 4, bias=False, return_quant_tensor=True)

quant_input = quant_identity(float_input)
quant_output = quant_linear(quant_input)

print(f"Float input:\n {float_input} \n")
print(f"Quant input:\n {quant_input} \n")
print(f"Quant output:\n {quant_output}")
```

```
Float input:
 tensor([[ 1.5410, -0.2934],
        [-2.1788,  0.5684],
        [-1.0845, -1.3986]])

Quant input:
 IntQuantTensor(value=tensor([[ 1.5490, -0.2894],
        [-2.1788,  0.5617],
        [-1.0894, -1.3958]], grad_fn=<MulBackward0>), scale=0.017021792009472847, ze

Quant output:
 IntQuantTensor(value=tensor([[-0.9109, -0.4609,  0.3135, -0.6523],
        [ 1.2089,  0.6524, -0.3752,  0.8697],
        [ 1.3893,  0.2816, -0.9011,  0.9521]], grad_fn=<MmBackward0>), scale=tensor([[9
```

Note that Having the input/output quantizer as part of a layer like `QuantLinear` or as a standalone `QuantIdentity` can make a difference at export time with targets like e.g. the standard ONNX opset, where `QLinearMatMul` or `QLinearConv` assume the output quantizer is part of the layer.

# QuantReLU layer

For QuantReLU the default quantizer is `Uint8ActPerTensorFloat`. Quantized activations layers like `QuantReLU` by defaults quantize the output, meaning that a relu is computed first, and then the output is quantized:

[11]:
```
torch.manual_seed(0)

from brevitas.nn import QuantReLU
```

```
quant_relu = QuantReLU(return_quant_tensor=True)

quant_output = quant_relu(float_input)

print(f"Float input:\n {float_input} \n")
print(f"Quant output:\n {quant_output}")
```

```
Float input:
 tensor([[ 1.5410, -0.2934],
        [-2.1788,  0.5684],
        [-1.0845, -1.3986]])

Quant output:
 IntQuantTensor(value=tensor([[1.5410, 0.0000],
        [0.0000, 0.5681],
        [0.0000, 0.0000]], grad_fn=<MulBackward0>), scale=0.006043121684342623, zero_po
```

# Requantizing a tensor

As you might expect, we can pass a `QuantTensor` to a quantized activation, and the tensor will be requantized:

[12]:
```
torch.manual_seed(0)

float_input = torch.randn(3, 2)
quant_identity = QuantIdentity(return_quant_tensor=True)
quant_relu = QuantReLU(return_quant_tensor=True)

signed_quant_output = quant_identity(float_input)
unsigned_quant_output = quant_relu(signed_quant_output)

print(f"Float input:\n {float_input} \n")
print(f"Quant output after QuantIdentity:\n {signed_quant_output}")
print(f"Quant output after QuantReLU:\n {unsigned_quant_output}")
```

```
Float input:
 tensor([[ 1.5410, -0.2934],
        [-2.1788,  0.5684],
        [-1.0845, -1.3986]])

Quant output after QuantIdentity:
 IntQuantTensor(value=tensor([[ 1.5490, -0.2894],
        [-2.1788,  0.5617],
        [-1.0894, -1.3958]], grad_fn=<MulBackward0>), scale=0.017021792009472847, zero_
Quant output after QuantReLU:
 IntQuantTensor(value=tensor([[1.5490, 0.0000],
        [0.0000, 0.5588],
        [0.0000, 0.0000]], grad_fn=<MulBackward0>), scale=0.006074443459510803, zero_po
```

Skip to main content

## How is the activation scale determined by default?

To minimize user interaction, default quantizers like `Uint8ActPerTensorFloat` and `Int8ActPerTensorFloat` initializes activations' scale by collecting statistics for a number of training steps (by default the 99.999 percentile of the absolute value for 300 steps).

This can be seen as an initial calibration step, although by default it happens with quantization already enabled, while calibration typically collects floating-point statistics first, and the enables quantization. These statistics are accumulated in an exponential moving average that at end of the collection phase is used to internally initialize a learned `nn.Parameter`. It's also possible to delay enabling quantization and perform proper calibration, but we will cover that later.

During the collection phase the quantizer behaves differently between `train()` and `eval()` mode, similarly to what happens for batch normalization. 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.

# Bias Quantization

In many inference toolchains, bias is assumed to be quantized with scale factor equal to `input_scale * weight_scale`, which means that we need a quantized input somehow. A predefined bias quantizer that reflects that assumption is `brevitas.quant.scaled_int.Int16Bias`. If we simply tried to set it to a `QuantLinear` without any sort of input quantization, we would get an error:

[13]:
```
torch.manual_seed(0)

from brevitas.quant.scaled_int import Int16Bias

float_input = torch.randn(3, 2)
quant_linear = QuantLinear(2, 4, bias=True, bias_quant=Int16Bias, return_quant_tensor=T

quant_output = quant_linear(float_input)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[13], line 8
      5 float_input = torch.randn(3, 2)
      6 quant_linear = QuantLinear(2, 4, bias=True, bias_quant=Int16Bias, return_quant_
----> 8 quant_output = quant_linear(float_input)

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

Skip to main content

```
   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
     65 def forward(self, input: Union[Tensor, QuantTensor]) -> Union[Tensor, QuantTens
---> 66     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
```

We can solve the issue by passing in a `QuantTensor` coming from a different layer as input, or by setting an input quantizer:

[14]:
```
torch.manual_seed(0)

float_input = torch.randn(3, 2)
quant_linear = QuantLinear(
    2, 4, bias=True, input_quant=Int8ActPerTensorFloat, bias_quant=Int16Bias, return_qu

quant_linear(float_input)
```

[14]:
```
IntQuantTensor(value=tensor([[-0.6541,  0.1263,  0.1680, -0.1231],
        [ 1.4658,  1.2395, -0.5207,  1.3989],
        [ 1.6461,  0.8687, -1.0466,  1.4813]], grad_fn=<AddmmBackward0>), scale=tensor(
```

# Operations on QuantTensor

## Element-wise Arithmetic

Some basic arithmetic operations on QuantTensor are supported. Depending on the operation, some constraints can apply.

### Element-wise adds

For element-wise addition, in line with traditional *fixed-point arithmetic*, the requirement is that scale of the operands is the same.

In order to make it work with the way quantized activation operates by default during the statistics collection phase, this restriction is enforced in `eval()` mode but not in `train()` mode (during which the operands' scales are averaged).

Calling the same QuantIdentity on tensors that need to be alligned is an easy way to do so:

[15]:
```
torch.manual_seed(0)

float_inp1 = torch.randn(3, 2)
float_inp2 = torch.randn(3, 2)
quant_identity = QuantIdentity(return_quant_tensor=True)

#Training mode, statistics are being collected, scaling factors are different but it do
train_quant_inp1 = quant_identity(float_inp1)
train_quant_inp2 = quant_identity(float_inp2)
train_mode_add = train_quant_inp1 + train_quant_inp2

#Inference mode, the EMA buffer is being used, scaling factors are the same
quant_identity.eval()
eval_quant_inp1 = quant_identity(float_inp1)
eval_quant_inp2 = quant_identity(float_inp2)
eval_mode_add = eval_quant_inp1 + eval_quant_inp2

print(f"Eval mode add quant inputs:\n {eval_quant_inp1} \n {eval_quant_inp2} \n")
print(f"Eval mode add quant output:\n {eval_mode_add}")
```

```
Eval mode add quant inputs:
 IntQuantTensor(value=tensor([[ 1.5335, -0.2875],
        [-2.0447,  0.5751],
        [-1.0863, -1.4057]]), scale=0.015974320471286774, zero_point=0.0, bit_width=8.0
 IntQuantTensor(value=tensor([[ 0.3994,  0.8307],
        [-0.7188, -0.3994],
        [-0.5910,  0.1757]]), scale=0.015974320471286774, zero_point=0.0, bit_width=8.0

Eval mode add quant output:
 IntQuantTensor(value=tensor([[ 1.9329,  0.5431],
        [-2.7636,  0.1757],
        [-1.6773, -1.2300]]), scale=0.015974320471286774, zero_point=0.0, bit_width=9.0
```

# Calling torch functions

Through the `__torch_function__` interface (supported on PyTorch >= 1.5.0), standard torch

Skip to main content

affine quantization a `QuantTensor` is returned, otherwise the output decays to a floating-point `torch.Tensor`.

In this context for a function `func` to be invariant to quantization means that the output of the function applied to the dequantized value should still be a dequantized value with the same scale, zero-point, and bit-width, i.e. given an input dequantized value where `input_dequant_value / scale + zero_point == input_integer_value` that can be represented within `bit_width` bits, `func(input_dequant_value) / scale + zero_point == output_integer_value` that can also be represented within `bit_width` bits.

# max_pool on QuantTensor

For example, `torch.nn.functional.max_pool1d` is invariant to quantization:

[16]:
```
torch.manual_seed(0)

float_inp = torch.randn(3, 2, 4)
quant_identity = QuantIdentity(return_quant_tensor=True)

quant_input = quant_identity(float_inp)
quant_output = torch.nn.functional.max_pool1d(quant_input, kernel_size=2, stride=2)

print(f"Quant input:\n {quant_input} \n")
print(f"Quant output:\n {quant_output}")
```

```
Quant input:
 IntQuantTensor(value=tensor([[[-1.1218, -1.1580, -0.2533, -0.4343],
         [ 0.8504,  0.6876, -0.3076, -2.1170]],

        [[ 0.4704, -0.1628,  1.4475,  0.2714],
         [ 0.1628,  0.8685, -0.1448, -0.1086]],

        [[ 0.9228,  1.2666,  2.0084,  0.0543],
         [ 0.6152, -0.4162, -0.8323, -2.3160]]], grad_fn=<MulBackward0>), scale=0.01809

Quant output:
 IntQuantTensor(value=tensor([[[-1.1218, -0.2533],
         [ 0.8504, -0.3076]],

        [[ 0.4704,  1.4475],
         [ 0.8685, -0.1086]],

        [[ 1.2666,  2.0084],
         [ 0.6152, -0.8323]]], grad_fn=<SqueezeBackward1>), scale=0.018094077706336975,
```

# tanh on QuantTensor

While for example `torch.tanh` is not:

[17]:
```
torch.manual_seed(0)

float_inp = torch.randn(3, 2, 4)
quant_identity = QuantIdentity(return_quant_tensor=True)

quant_input = quant_identity(float_inp)
quant_output = torch.tanh(quant_input)

print(f"Quant input:\n {quant_input} \n")
print(f"Quant output:\n {quant_output}")
```

```
Quant input:
 IntQuantTensor(value=tensor([[[-1.1218, -1.1580, -0.2533, -0.4343],
         [ 0.8504,  0.6876, -0.3076, -2.1170]],

        [[ 0.4704, -0.1628,  1.4475,  0.2714],
         [ 0.1628,  0.8685, -0.1448, -0.1086]],

        [[ 0.9228,  1.2666,  2.0084,  0.0543],
         [ 0.6152, -0.4162, -0.8323, -2.3160]]], grad_fn=<MulBackward0>), scale=0.01809

Quant output:
 tensor([[[-0.8082, -0.8204, -0.2480, -0.4089],
         [ 0.6913,  0.5964, -0.2983, -0.9714]],

        [[ 0.4386, -0.1614,  0.8952,  0.2649],
         [ 0.1614,  0.7006, -0.1438, -0.1081]],

        [[ 0.7272,  0.8529,  0.9646,  0.0542],
         [ 0.5478, -0.3937, -0.6817, -0.9807]]], grad_fn=<TanhBackward0>)
```

# QuantTensor concatenation

Concatenation, i.e. `torch.cat`, can also be applied to quantized tensors whenever they all have the same sign, scale, zero-point and bit-width. Simiarly to element-wise adds, scale and zero-point are allowed to be different in training mode, but have to be the same in inference mode. Example:

[18]:
```
torch.manual_seed(0)

float_inp1 = torch.randn(3, 2)
float_inp2 = torch.randn(3, 2)
quant_identity = QuantIdentity(return_quant_tensor=True)
```

```
#Inference mode, the EMA buffer is being used, scaling factors are the same
quant_identity.eval()
eval_quant_inp1 = quant_identity(float_inp1)
eval_quant_inp2 = quant_identity(float_inp2)
eval_mode_cat = torch.cat([eval_quant_inp1, eval_quant_inp2], dim=1)

print(f"Eval mode concat quant inputs:\n {eval_quant_inp1} {eval_quant_inp2} \n")
print(f"Eval mode concat quant output:\n {eval_mode_cat}")
```

```
Eval mode concat quant inputs:
 IntQuantTensor(value=tensor([[ 1.5335, -0.2875],
        [-2.0447,  0.5751],
        [-1.0863, -1.4057]]), scale=0.015974320471286774, zero_point=0.0, bit_width=8.0
        [-0.7188, -0.3994],
        [-0.5910,  0.1757]]), scale=0.015974320471286774, zero_point=0.0, bit_width=8.0

Eval mode concat quant output:
 IntQuantTensor(value=tensor([[ 1.5335, -0.2875,  0.3994,  0.8307],
        [-2.0447,  0.5751, -0.7188, -0.3994],
        [-1.0863, -1.4057, -0.5910,  0.1757]]), scale=0.015974320471286774, zero_point=
```

# Customizing Quantizers

## Common keyword arguments

The easiest way to customize a quantizer is by passing appropriate keyword arguments. What happens any time we are setting a keyword argument is that we are overriding an hyperparameter in the underlying quantizer. Different quantizers expose different hyperparameters, which is why quantized layers like QuantLinear accept arbitrary `**kwargs`:

## Weight bit-width

Let's look at customizing the default quantizers first. To override the `bit_width` of weights in the default `weight_quant` quantizer we set `weight_bit_width`:

[19]:
```
torch.manual_seed(0)

quant_linear = QuantLinear(2, 4, weight_bit_width=5, bias=True)

print(f"Weight QuantTensor:\n {quant_linear.quant_weight()}")
```

```
Weight QuantTensor:
```

Skip to main content

```
        [-0.2716,  0.1940],
        [-0.0000,  0.5432]], grad_fn=<MulBackward0>), scale=0.03879871591925621, zero_p
```

## Per-channel weight quantization

We can enable channel-wise quantization by setting `weight_scaling_per_output_channel=True`:

[20]:
```
torch.manual_seed(0)

quant_linear = QuantLinear(2, 4, weight_bit_width=5, weight_scaling_per_output_channel=

print(f"Weight QuantTensor:\n {quant_linear.quant_weight()}")
```

```
Weight QuantTensor:
 IntQuantTensor(value=tensor([[-0.0000,  0.3793],
        [-0.5820, -0.5044],
        [-0.2723,  0.1816],
        [-0.0000,  0.5607]], grad_fn=<MulBackward0>), scale=tensor([[0.0253],
        [0.0388],
        [0.0182],
        [0.0374]], grad_fn=<DivBackward0>), zero_point=0.0, bit_width=5.0, signed_t=Tru
```

## Activation bit-width

Similarly we can set the `bit_width` for activations:

[21]:
```
torch.manual_seed(0)

float_input = torch.randn(3, 2)
quant_identity = QuantIdentity(bit_width=3, return_quant_tensor=True)
print(f"QuantTensor:\n {quant_identity(float_input)}")
```

```
QuantTensor:
 IntQuantTensor(value=tensor([[ 1.6341, -0.5447],
        [-2.1788,  0.5447],
        [-1.0894, -1.6341]], grad_fn=<MulBackward0>), scale=0.5446973443031311, zero_po
```

## Activation quantization with max_val init

Certain quantizers require the user to pass in additional keyword arguments. This is the case for

user-defined maximum value rather than from statistics:

[22]:
```
torch.manual_seed(0)

from brevitas.quant import Uint8ActPerTensorFloatMaxInit

float_inp1 = torch.randn(3, 2)
quant_relu = QuantReLU(max_val=6.0, act_quant=Uint8ActPerTensorFloatMaxInit, return_qua
quant_relu(float_inp1)
```

[22]:
```
IntQuantTensor(value=tensor([[1.5294, 0.0000],
        [0.0000, 0.5647],
        [0.0000, 0.0000]], grad_fn=<MulBackward0>), scale=tensor(0.0235, grad_fn=<DivBa
```

# Per-channel activation quantization

And its also possible to perform *per-channel activation quantization* if the number of channels is statically defined by providing some extra information. This makes sense for example as input to a depthwise separable layer, like a depthwise-separable convolution. Example:

[23]:
```
torch.manual_seed(0)

from brevitas.nn import QuantConv1d

BATCHES = 1
CHANNELS = 2
FEATURES = 5
KERNEL = 3

float_input = torch.randn(BATCHES, CHANNELS, FEATURES)
per_channel_depthwise_quant_conv = QuantConv1d(
    CHANNELS, CHANNELS, KERNEL, groups=CHANNELS, bias=True,
    # set the quantizers
    input_quant=Int8ActPerTensorFloat,
    bias_quant=Int16Bias,
    # customize the input quantizer with extra keyword args
    input_bit_width=3,
    input_scaling_per_output_channel=True,
    input_scaling_stats_permute_dims=(1, 0, 2), # permute dims to put channels first du
    input_per_channel_broadcastable_shape=(1, CHANNELS, 1), # shape of the learned para
    return_quant_tensor=True)

quant_output = per_channel_depthwise_quant_conv(float_input)

print(f"Float input:\n {float_input} \n")
print(f"Per-channel quant output:\n {quant_output}")
```

Skip to main content

```
Float input:
 tensor([[[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845],
         [-1.3986,  0.4033,  0.8380, -0.7193, -0.4033]]])

Per-channel quant output:
 IntQuantTensor(value=tensor([[[ 0.8616, -0.7012,  0.4503],
         [-1.1285, -0.4937, -0.1901]]], grad_fn=<ConvolutionBackward0>), scale=tensor([
         [0.0013]]], grad_fn=<MulBackward0>), zero_point=tensor([[[-254.0000],
         [ 406.0000]]], grad_fn=<DivBackward0>), bit_width=17.0, signed_t=True, trainin
```

# Inheriting from a quantizer

What happens whenever we are passing a keyword argument (with prefix `weight_`, `input_`, `output_`, `bias_` for a layer like `QuantLinear`, no prefix for an activation layer like `QuantReLU`) is that we are overriding attributes of the underlying quantizer. To make things more compact and re-usable, we can also simply define a new quantizer by inheriting from the one we are customizing. The previous example then would look like this:

[24]:
```
torch.manual_seed(0)

from brevitas.inject.enum import ScalingPerOutputType
from brevitas.nn import QuantConv1d

BATCHES = 1
CHANNELS = 2
FEATURES = 5
KERNEL = 3

class PerChannel3bActQuant(Int8ActPerTensorFloat):
    bit_width = 3
    scaling_per_output_channel=True
    scaling_stats_permute_dims=(1, 0, 2)

float_input = torch.randn(BATCHES, CHANNELS, FEATURES)
per_channel_depthwise_quant_conv = QuantConv1d(
    CHANNELS, CHANNELS, KERNEL, groups=CHANNELS, bias=True,
    # set the quantizers
    input_quant=PerChannel3bActQuant,
    bias_quant=Int16Bias,
    # layer-specific kwarg
    input_per_channel_broadcastable_shape=(1, CHANNELS, 1),
    return_quant_tensor=True)

quant_output = per_channel_depthwise_quant_conv(float_input)

print(f"Float input:\n {float_input} \n")
```

```
Float input:
 tensor([[[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845],
         [-1.3986,  0.4033,  0.8380, -0.7193, -0.4033]]])

Per-channel quant output:
 IntQuantTensor(value=tensor([[[ 0.8616, -0.7012,  0.4503],
         [-1.1285, -0.4937, -0.1901]]], grad_fn=<ConvolutionBackward0>), scale=tensor([
         [0.0013]]], grad_fn=<MulBackward0>), zero_point=tensor([[[-254.0000],
         [ 406.0000]]], grad_fn=<DivBackward0>), bit_width=17.0, signed_t=True, trainin
```

# Defining a quantizer from scratch with enums

So far we have seen some options to customize the behaviour of existing quantizers. Let's look now at how exactly they are defined.

Below is how two of the quantizers we have seen so far are declared within Brevitas using the enum-driven API (except for zero-point). As we will see in a second, it's not the only way to define a quantizer, but it's a way for users to easily experiment with different built-in options.

The quantizer defines various directives and hyperparameters driving the underlying algorithms as class level attributes.

## Weight quantizer

For weights, What we are doing by inheriting from `WeightQuantSolver`, meaning those directive should be translated to an implementation that makes sense for weights.

[25]:
```python
from brevitas.inject.enum import *
from brevitas.core.zero_point import ZeroZeroPoint
from brevitas.quant.solver import WeightQuantSolver, ActQuantSolver

class Int8WeightPerTensorFloat(WeightQuantSolver):
    quant_type = QuantType.INT # integer quantization
    bit_width_impl_type = BitWidthImplType.CONST # constant bit width
    float_to_int_impl_type = FloatToIntImplType.ROUND # round to nearest
    scaling_impl_type = ScalingImplType.STATS # scale based on statistics
    scaling_stats_op = StatsOp.MAX # scale statistics is the absmax value
    restrict_scaling_type = RestrictValueType.FP # scale factor is a floating point val
    scaling_per_output_channel = False # scale is per tensor
    bit_width = 8 # bit width is 8
    signed = True # quantization range is signed
```

Skip to main content

```
    narrow_range = True # quantization range is [-127,127] rather than [-128, 127]
    zero_point_impl = ZeroZeroPoint # zero point is 0.
```

# Activation quantizer

Similarly for activations we can inherit from `ActQuantSolver`. The generated implementation is gonna be a bit different, since we are quantizing tensors defined at runtime rather than a parameter, by it's driven by the same enums:

[26]:
```
class Int8ActPerTensorFloat(ActQuantSolver):
    quant_type = QuantType.INT # integer quantization
    bit_width_impl_type = BitWidthImplType.CONST # constant bit width
    float_to_int_impl_type = FloatToIntImplType.ROUND # round to nearest
    scaling_impl_type = ScalingImplType.PARAMETER_FROM_STATS # scale is a parameter ini
    scaling_stats_op = StatsOp.PERCENTILE # scale statistics is a percentile of the abs
    high_percentile_q = 99.999 # percentile is 99.999
    collect_stats_steps = 300  # statistics are collected for 300 forward steps before
    restrict_scaling_type = RestrictValueType.FP # scale is a floating-point value
    scaling_per_output_channel = False  # scale is per tensor
    bit_width = 8  # bit width is 8
    signed = True # quantization range is signed
    narrow_range = False # quantization range is [-128, 127] rather than [-127, 127]
    zero_point_impl = ZeroZeroPoint # zero point is 0.
```

Any of attributes in the quantizers above can be passed in or overriden as a keyword argument to the layer the quantizer is passed to (together with its appropriate prefix), with keyword arguments always having the priority over the value defined in the quantizer.

# Learned scale and bit-width quantizer

Let's now look at how we can build something more advanced by tweaking a few options. We want a weight quantizer with: - per-channel scale factors learned in log domain as a parameter initialized from absmax statistics. - bit-width initialized to 8b and learned as a parameter from there.

[27]:
```
torch.manual_seed(0)

from brevitas.quant import Int8WeightPerChannelFloat

class LearnedIntWeightPerChannelFloat(Int8WeightPerChannelFloat):
    scaling_impl_type = ScalingImplType.PARAMETER_FROM_STATS
    restrict_scaling_type = RestrictValueType.LOG_FP
```

```
quant_linear = QuantLinear(2, 4, weight_quant=LearnedIntWeightPerChannelFloat, bias=Fal

print(f"Weight QuantTensor:\n {quant_linear.quant_weight()}")
```

```
Weight QuantTensor:
 IntQuantTensor(value=tensor([[-0.0060,  0.3793],
        [-0.5820, -0.5224],
        [-0.2723,  0.1887],
        [-0.0132,  0.5607]], grad_fn=<MulBackward0>), scale=tensor([[0.0030],
        [0.0046],
        [0.0021],
        [0.0044]], grad_fn=<DivBackward0>), zero_point=0.0, bit_width=8.0, signed_t=Tru
```

Notice how the quantized weights' `bit_width` is now a value we can backpropagate through, as it exposes a `grad_fn` function. This way we can use it as part of a loss regularization function to model a particular hardware cost function, e.g. pushing larger layers to have a smaller bit width. The same principle applies to activation, and we can combine them:

[28]:
```
torch.manual_seed(0)

class LearnedIntActPerTensorFloat(Int8ActPerTensorFloat):
    bit_width_impl_type = BitWidthImplType.PARAMETER
    restrict_scaling_type = RestrictValueType.LOG_FP

float_inp = torch.randn(3, 2)
quant_linear = QuantLinear(
    2, 4,
    input_quant=LearnedIntActPerTensorFloat,
    weight_quant=LearnedIntWeightPerChannelFloat,
    return_quant_tensor=True, bias=False)

quant_linear(float_inp)
```

[28]:
```
IntQuantTensor(value=tensor([[-0.9109, -0.4588,  0.3119, -0.6530],
        [ 1.2089,  0.6493, -0.3731,  0.8706],
        [ 1.3893,  0.2823, -0.8979,  0.9543]], grad_fn=<MmBackward0>), scale=tensor([[9
       grad_fn=<MulBackward0>), zero_point=tensor([0.]), bit_width=tensor(17., grad_fn=
```

As we can see, we can backpropagate through the output `bit_width`. By including it in a loss function, we could then try to control the size of the output accumulator, and in turn the `bit_width` of both weights and input.

# Retraining from floating-point

In many scenarios it's convenient to perform quantization-aware training starting from a floating-point model. Say for example we want to load a pretrained-floating point state on top of the layer

with learned bit-width and scale we just saw. We simulate it with a separate floating-point `nn.Linear`. If we didn't do anything else we would get an error:

[29]:
```
torch.manual_seed(0)

from torch import nn

float_linear = nn.Linear(2, 4, bias=False)
quant_linear = QuantLinear(
    2, 4,
    input_quant=LearnedIntActPerTensorFloat,
    weight_quant=LearnedIntWeightPerChannelFloat,
    return_quant_tensor=True, bias=False)

quant_linear.load_state_dict(float_linear.state_dict())
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[29], line 12
      5 float_linear = nn.Linear(2, 4, bias=False)
      6 quant_linear = QuantLinear(
      7     2, 4,
      8     input_quant=LearnedIntActPerTensorFloat,
      9     weight_quant=LearnedIntWeightPerChannelFloat,
     10     return_quant_tensor=True, bias=False)
---> 12 quant_linear.load_state_dict(float_linear.state_dict())

File /proj/xlabs/users/nfraser/opt/miniforge3/envs/20231115_brv_pt1.13.1/lib/python3.
   1666         error_msgs.insert(
   1667             0, 'Missing key(s) in state_dict: {}. '.format(
   1668                 ', '.join('"{}"'.format(k) for k in missing_keys)))
   1670 if len(error_msgs) > 0:
-> 1671     raise RuntimeError('Error(s) in loading state_dict for {}:\n\t{}'.format(
   1672                        self.__class__.__name__, "\n\t".join(error_msgs)))
   1673 return _IncompatibleKeys(missing_keys, unexpected_keys)

RuntimeError: Error(s) in loading state_dict for QuantLinear:
        Missing key(s) in state_dict: "input_quant.fused_activation_quant_proxy.tensor_
```

That's because the quantizers for both weights and input are introducing new learned parameters that are not present in the original floating-point layer.

We can supress the error to accomodate the re-training scenario by either setting the env variable `BREVITAS_IGNORE_MISSING_KEYS=1`, or by enabling the corresponding config flag:

[30]:
```
torch.manual_seed(0)

from torch import nn
from brevitas import config
```

Skip to main content

```
float_linear = nn.Linear(2, 4, bias=False)
quant_linear = QuantLinear(
    2, 4,
    input_quant=LearnedIntActPerTensorFloat,
    weight_quant=LearnedIntWeightPerChannelFloat,
    return_quant_tensor=True, bias=False)

quant_linear.load_state_dict(float_linear.state_dict())
```

[30]: <All keys matched successfully>

However, in this scenario we are saying we want the learned component of the scale factor for weights to be initialized to the maximum value of the weight tensor on a per channel basis. So even though it's not part of the floating-point state dict, the value of the weight scale factor should be updated. Thanks to how the weight quantizer works this happens *automatically*:

[31]:
```
torch.manual_seed(0)

from torch import nn
from brevitas import config

config.IGNORE_MISSING_KEYS = True

float_linear = nn.Linear(2, 4, bias=False)
quant_linear = QuantLinear(
    2, 4,
    input_quant=LearnedIntActPerTensorFloat,
    weight_quant=LearnedIntWeightPerChannelFloat,
    return_quant_tensor=True, bias=False)

print(f'Weight scale before load_state_dict: {quant_linear.quant_weight().scale}')

quant_linear.load_state_dict(float_linear.state_dict())

print(f'Weight scale after load_state_dict: {quant_linear.quant_weight().scale}')
```

```
Weight scale before load_state_dict: tensor([[0.0015],
        [0.0017],
        [0.0053],
        [0.0023]], grad_fn=<DivBackward0>)
Weight scale after load_state_dict: tensor([[0.0030],
        [0.0046],
        [0.0021],
        [0.0044]], grad_fn=<DivBackward0>)
```

This works also in the scenario where the weight quantizer is shared among multiple layers.

# Defining a quantizer from scratch with dependency-injection

So far we have seen how users can pick and adopt various quantization algorithms that can be expressed through components already built into Brevitas. The more interesting question is how to expressed completely novel algorithms based on new quantization components.

# Activation quantization from scratch

To do so we need to move away from the enum-driven API, and understand how quantizers work underneath. If we look at the structure of a layer like `QuantReLU`, we can immediately notice that it's formed by a variety of nested modules:

[32]:
```
QuantReLU()
```

[32]:
```
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_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(
```

Skip to main content

```
            (msb_clamp_bit_width_impl): BitWidthConst(
              (bit_width): StatelessBuffer()
            )
          )
        )
      )
    )
```

Where the implementation of the quantization component is expressed by the `tensor_quant` module.

The idea is that we express a quantization algorithm as various `nn.Module` combined in a depedency-injection fashion, meaning that they combine from the outside-in through standardized interfaces.

So we have different modules to express a particular type of scale factor, a particular type of rounding, a particular integer quantization procedure, and so on. The overall quantization algorithm is then expressed as a nested assembly of all its different pieces. If we were to do it manually for the layer above, it would look like this:

[33]:
```
from brevitas.core.function_wrapper import TensorClamp, OverTensorView, RoundSte
from brevitas.core.scaling import ParameterFromRuntimeStatsScaling, IntScaling
from brevitas.core.stats import AbsPercentile
from brevitas.core.restrict_val import FloatRestrictValue
from brevitas.core.bit_width import BitWidthConst
from brevitas.core.quant import IntQuant, RescalingIntQuant
from brevitas.core.zero_point import ZeroZeroPoint
from brevitas.core.function_wrapper.misc import Identity

tensor_quant = RescalingIntQuant(
    int_quant=IntQuant(
        float_to_int_impl=RoundSte(),
        tensor_clamp_impl=TensorClamp(),
        input_view_impl=Identity,
        signed=False,
        narrow_range=False),
    zero_point_impl=ZeroZeroPoint(),
    bit_width_impl=BitWidthConst(bit_width=8),
    int_scaling_impl=IntScaling(signed=False, narrow_range=False),
    scaling_impl=ParameterFromRuntimeStatsScaling(
        scaling_stats_input_view_shape_impl=OverTensorView(),
        scaling_stats_impl=AbsPercentile(
            high_percentile_q=99.999, stats_reduce_dim=None),
        collect_stats_steps=300)
)
```

With this approach we can express arbitrary quantization algorithms in a very modular fashion. However, we have two extra constraints we care about:

Skip to main content

- This degree of modularity comes at a cost, i.e. the overhead of doing a lot of small operations scattered across different modules in Python shows. We want to avoid it by taking advantage of PyTorch's TorchScript JIT compiler, meaning the `tensor_quant` module above should be fused and end-to-end compiled.
- We want a way to define our quantizer such that it can be re-instantiated and re-compiled on-demand in certain scenarios, like whenever it's shared among multiple layers or when a pretrained `state_dict` is load on top of it.

To solve the first issue, we restrict ourselves to using only `ScriptModule`, a variant of a `nn.Module` that can be JIT-compiled. This is also why we favour composition over inheritance, as TorchScript has almost no support for inheritance. So all the modules we are instantiating above derive from an alias to a ScriptModule called `brevitas.jit.ScriptModule`, whose compilation can be enabled by setting the env variable `BREVITAS_JIT=1`.

To solve the second issue, we adopt and extend an auto-wiring dependency injection library called dependencies that performs the assembly and instantiation process of `tensor_quant` automatically. Let's look at an example to make things more clear:

[34]:
```
from brevitas.inject import ExtendedInjector
from brevitas.proxy import ActQuantProxyFromInjector

class Int8ActPerTensorFloatFromScratch(ExtendedInjector):
    proxy_class = ActQuantProxyFromInjector # Python wrapper, returns QuantTensor
    tensor_quant = RescalingIntQuant # TorchScript implementation, returns (dequant_val
    int_quant = IntQuant
    float_to_int_impl = RoundSte
    zero_point_impl = ZeroZeroPoint
    bit_width_impl = BitWidthConst
    tensor_clamp_impl = TensorClamp
    int_scaling_impl = IntScaling
    scaling_impl = ParameterFromRuntimeStatsScaling
    scaling_stats_input_view_shape_impl = OverTensorView
    scaling_stats_impl = AbsPercentile
    restrict_scaling_impl = FloatRestrictValue
    high_percentile_q = 99.999
    collect_stats_steps = 300
    scaling_shape = ()
    stats_reduce_dim = None
    bit_width = 8
    narrow_range = True
    signed = True
```

`Int8ActPerTensorFloatFromScratch` is equivalent to `Int8ActPerTensorFloat`, but defined directly in terms of the modules that define the implementation, rather than with enums. As such, we inherit

Skip to main content

two attributes: - `proxy_class` represents the Python wrapper that interfaces a quantized layer like QuantIdentity with the quantizer itself, and for convinience is defined as part of the quantizer itself. It has to return a `QuantTensor`. - `tensor_quant` is the actual ScriptModule implementating our quantization algorithm. It has to return a 4-tuple containing tensors (dequant_value, scale, zero_point, bit_width).

`tensor_quant` is automatically instantiated by the dependency-injection engine by matching its keyword argument to attributes of the `ÈxtendedInjector`. If one of the attributes is a class that requires to be instantiated the process is repeated in a recursive manner. If it's already an instance, it's taken as is.

Note that the enum-driven API only adds an extra layer of abstraction on top of this, but it maps to the same components with the same names.

This way, indipendently of how it's originally defined, a user can override any component of an existing quantizer in a fine grained manner, or define a new quantizer from scratch with entirely custom components.

# Weight quantization with learned scale from scratch

Similarly, we can define a weight quantizer from scratch with a learned scale factor implemented by `ParameterScaling` module. To initialize it, we define `scaling_init` as a function of the full precision weights to quantize, by taking the absmax of weight tensor, as with did in previous examples. To do so, we define a `@value` function, a function which is evaluated at *dependency-injection time*:

[35]:
```
from brevitas.inject import value
from brevitas.proxy import WeightQuantProxyFromInjector
from brevitas.core.scaling import ParameterScaling
from brevitas.core.function_wrapper.misc import Identity

class Int8ActPerTensorFloatParameterFromScratch(ExtendedInjector):

    @value
    def scaling_init(module):
        return module.weight.abs().max().detach()

    proxy_class = WeightQuantProxyFromInjector # Python wrapper, returns QuantTensor
    tensor_quant = RescalingIntQuant
    int_quant = IntQuant
```

Skip to main content

```
    bit_width_impl = BitWidthConst
    tensor_clamp_impl = TensorClamp
    int_scaling_impl = IntScaling
    scaling_impl = ParameterScaling
    restrict_scaling_impl = FloatRestrictValue
    input_view_impl = Identity
    scaling_shape = ()
    bit_width = 8
    narrow_range = True
    signed = True

quant_linear = QuantLinear(2, 4, weight_quant=Int8ActPerTensorFloatParameterFromScratch
```

This is how `scaling_impl_type = ScalingImplType.PARAMETER_FROM_STATS` for weight quantization works underneath.

the argument `module` of `scaling_init` represents the layer (or tuple of layers) the quantizer is passed to, and it allows the quantizer to depend on properties of the layer to quantize.

By declaring `scaling_init` in this way we can easily recompute it any time the layer changes. As we saw previously, for changes to the `state_dict` this happens automatically. For in-place changes like weight initialization, which cannot easily be intercepted automatically, we can invoke it manually by calling `quant_linear.weight_quant.init_tensor_quant()`.

# Sharing learned bit-width among layers

We saw in a previous section how to share entire quantizers among layers. We are now interested in sharing only certain components, and we leverage the dependency-injection mechanism we just saw to do so.

For example, we want the `QuantIdentity` and the `QuantLinear` with learned bit-width of our previous example to share the same learned bit-width. That can be useful whenever a particular hardware target supports only operations among tensors of the same datatype . With some verbosity, doing so is possible:

[36]:
```
torch.manual_seed(0)

float_inp = torch.randn(3, 2)
quant_linear = QuantLinear(
    2, 4, weight_quant=LearnedIntWeightPerChannelFloat, return_quant_tensor=True, bias=
quant_identity = QuantIdentity(
    bit_width_impl=quant_linear.weight_quant.tensor_quant.msb_clamp_bit_width_impl,
```

```
quant_identity_bit_width = quant_identity.act_quant.fused_activation_quant_proxy.tensor
quant_linear_bit_width = quant_linear.weight_quant.tensor_quant.msb_clamp_bit_width_imp

assert_with_message(quant_identity_bit_width is quant_linear_bit_width)
```

```
True
```

As we can see the two bit-width implementations are the same instance. What we are doing is sharing the instance of `msb_clamp_bit_width_impl` from one quantizer to another.

# Export

Brevitas doesn't perform any kind of acceleration on its own. To do so, a quantized model needs to be exported to a backend first. In a perfect world there would be a single way to represent quantized models. In practice however that's not the case, and different toolchains make different assumptions on what kind of quantization is supported and how it should be represented.

Brevitas then supports different flavours of export at different levels of abstractions by taking advantage of PyTorch's support for custom symbolic representations, and in particular ONNX.

Existing export flows assume static quantization, i.e. scales, zero-point and bit-width need to be indipedentent from the input. All export flows abstract away from the specifics of how scales, zero-point and bit-width have been determined. However, different export flows provide supports only for certain combinations of scales, zero-point, precision, or structure of the model.

We install onnx and onnxoptimizer as they are required for export, and we use Netron to visualize the exported models:

[37]:
```
!pip install netron onnx onnxoptimizer
```

```
Requirement already satisfied: netron in /proj/xlabs/users/nfraser/opt/miniforge3/envs/
Requirement already satisfied: onnx in /proj/xlabs/users/nfraser/opt/miniforge3/envs/20
Requirement already satisfied: onnxoptimizer in /proj/xlabs/users/nfraser/opt/miniforge
Requirement already satisfied: numpy in /proj/xlabs/users/nfraser/opt/miniforge3/envs/2
Requirement already satisfied: protobuf>=3.20.2 in /proj/xlabs/users/nfraser/opt/minifo
```

[38]:
```
import netron
import time
from IPython.display import IFrame

def show_netron(model_path, port):
    time.sleep(3.)
    netron.start(model_path, address=("localhost", port), browse=False)
    return IFrame(src=f"http://localhost:{port}/", width="100%", height=400)
```

Skip to main content

# Export to custom Quantized ONNX

As an alternative, we can export it to QONNX, a custom ONNX dialect that Brevitas defines with support for custom quantization operators that can capture those informations:

[39]:
```
torch.manual_seed(0)

from brevitas.export import export_qonnx
from brevitas.quant import Int8ActPerTensorFloat, Int16Bias

float_inp = torch.randn(1, 2, 5)

quant_conv_4b8b = QuantConv1d(
    2, 4, 3, bias=True, weight_bit_width=4,
    input_quant=Int8ActPerTensorFloat,
    output_quant=Int8ActPerTensorFloat,
    bias_quant=Int16Bias)

output_path = 'brevitas_onnx_conv4b8b.onnx'
exported_model = export_qonnx(quant_conv_4b8b, input_t=float_inp, export_path=output_pa
```

```
[W NNPACK.cpp:53] Could not initialize NNPACK! Reason: Unsupported hardware.
```

[40]:
```
show_netron(output_path, 8083)
```

```
Serving 'brevitas_onnx_conv4b8b.onnx' at http://localhost:8083
```

[40]:

Skip to main content

In the `Quant` nodes above, arbitrary scale, zero-point and bit-widths are supported. This way we can support exporting scenarios where, for example, we are quantizing only weights to 4b, which a QLinearConv wouldn't be able to capture:

[40]:
```
torch.manual_seed(0)

quant_conv_4b_weights = QuantConv1d(2, 4, 3, bias=True, weight_bit_width=4)

output_path = 'brevitas_onnx_conv_4b_weights.onnx'
exported_model = export_qonnx(quant_conv_4b_weights, input_t=float_inp, export_path=out
```

[42]:
```
show_netron(output_path, 8084)
```

```
Serving 'brevitas_onnx_conv_4b_weights.onnx' at http://localhost:8084
```

[42]:

The custom format shown above can integrated into ONNX-based toolchains, e.g. it's supported by our own FINN toolchain for low-precision dataflow style custom FPGAs implementations, and would be a starting point for direct integration with TVM.

# Brevitas and FX

In this tutorial we have covered how to incorporate quantized layers into the design of a neural networks. However, in some scenarios it might convenient to start from an existing definition of a neural networks in floating-point, and programmatically replace floating-point layers with quantized

Skip to main content

PyTorch's recently introduced FX graph subsystem enables this kind of transformations. While covering FX goes beyond the scope of this tutorial, it's worth mentioning that Brevitas has embraced FX, and it actually implements a backport of PyTorch's 1.8 FX (the current LTS) under `brevitas.fx`, together with a custom input-driven tracer (`brevitas.fx.value_trace`) which similarly to `torch.jit.trace` allows to trace through conditionals and unpacking operations, as well as various graph transformations under `brevitas.graph`.

# Calibration-based post-training quantization

While Brevitas was originally designed for quantization-aware training, performing calibration-based post-training quantization is possible, and it can be useful on its own or an intermediate step between floating-point training and quantization-aware retraining.

The idea of calibration-based quantization is to performs forward passes only with a small set of data and collect statistics to determine scale factors and zero-points.

Any Brevitas quantizer that is based on statistics can be used for this purpose, with the caveat that don't want quantization to be enabled while statistics are being collected, or the data wouldn't be representative of what the floating-point model does, so we temporarely disabling quantization while doing so. Afterwards, we re-enable calibration and apply bias correction.

Assuming `quant_model` is a quantized model, with pretrained floating-point weights loaded on top, we can do as follows:

Previous
Tutorials
Next
An overview of QuantTensor and QuantConv2d

[41]:
```
from brevitas.graph.calibrate import bias_correction_mode
from brevitas.graph.calibrate import calibration_mode

def calibrate_model(calibration_loader, quant_model, args):
    with torch.no_grad():
```