Getting started

Brevitas serves various types of users and end goals. With respect to **defining** quantized models, Brevitas supports two types of user flows:

- *By hand*, writing a quantized model using `brevitas.nn` quantized layers, possibly by modifying an original PyTorch floating-point model definition.
- *Programmatically*, by taking a floating-point model as input and automatically deriving a quantized model definition from it according to some user-defined criteria.

Once a quantize model is defined in either way, it can then be used as a starting point for either:

- PTQ (Post-Training Quantization), starting from a pretrained floating-point model.
- QAT (Quantization Aware Training), either training from scratch or finetuning a pretrained floating-point model.
- PTQ followed by QAT finetuning, to combine the best of both approaches.

PTQ over hand or programmatically defined quantized models

Checkout how it's done for ImageNet classification over torchvision or other hand-defined models with our example scripts here.

Defining a quantized model with brevitas.nn layers

We consider quantization of a classic neural network, LeNet-5.

Skip to main content

Let's say we are interested in assessing how well the model does at *4 bit weights* for CIFAR10 classification. For the purpose of this tutorial we will skip any detail around how to perform training, as training a neural network with Brevitas is no different than training any other neural network in PyTorch.

`brevitas.nn` provides quantized layers that can be used **in place of** and/or **mixed with** traditional `torch.nn` layers. In this case then we import `brevitas.nn.QuantConv2d` and `brevitas.nn.QuantLinear` in place of their PyTorch variants, and we specify `weight_bit_width=4`. For relu and max-pool, we leverage the usual `torch.nn.ReLU` and `torch.nn.functional.max_pool2d`.

The result is the following:

```
from torch import nn
from torch.nn import Module
import torch.nn.functional as F

import brevitas.nn as qnn


class QuantWeightLeNet(Module):
    def __init__(self):
        super(QuantWeightLeNet, self).__init__()
        self.conv1 = qnn.QuantConv2d(3, 6, 5, bias=True, weight_bit_width=4)
        self.relu1 = nn.ReLU()
        self.conv2 = qnn.QuantConv2d(6, 16, 5, bias=True, weight_bit_width=4)
        self.relu2 = nn.ReLU()
        self.fc1   = qnn.QuantLinear(16*5*5, 120, bias=True, weight_bit_width=4)
        self.relu3 = nn.ReLU()
        self.fc2   = qnn.QuantLinear(120, 84, bias=True, weight_bit_width=4)
        self.relu4 = nn.ReLU()
        self.fc3   = qnn.QuantLinear(84, 10, bias=True, weight_bit_width=4)

    def forward(self, x):
        out = self.relu1(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = self.relu2(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.reshape(out.reshape[0], -1)
        out = self.relu3(self.fc1(out))
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out

quant_weight_lenet = QuantWeightLeNet()

# ... training ...
```

Skip to main content

A neural network with 4 bits weights and floating-point activations can provide an advantage in terms of model storage, but it doesn't provide any advantage in terms of compute, as the weights need to be converted to float at runtime first. In order to make more practical, we want to quantize activations too.

Weights and activations quantization, float biases

We now quantize both weights and activations to 4 bits, while keeping biases in floating-point. In order to do so, we replace `torch.nn.ReLU` with `brevitas.nn.QuantReLU`, specifying `bit_width=4`. Additionally, in order to quantize the very first input, we introduce a `brevitas.nn.QuantIdentity` at the beginning of the network. The end result is the following:

```
from torch.nn import Module
import torch.nn.functional as F

import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant


class QuantWeightActLeNet(Module):
    def __init__(self):
        super(QuantWeightActLeNet, self).__init__()
        self.quant_inp = qnn.QuantIdentity(bit_width=4)
        self.conv1 = qnn.QuantConv2d(3, 6, 5, bias=True, weight_bit_width=4)
        self.relu1 = qnn.QuantReLU(bit_width=4)
        self.conv2 = qnn.QuantConv2d(6, 16, 5, bias=True, weight_bit_width=4)
        self.relu2 = qnn.QuantReLU(bit_width=3)
        self.fc1   = qnn.QuantLinear(16*5*5, 120, bias=True, weight_bit_width=4)
        self.relu3 = qnn.QuantReLU(bit_width=4)
        self.fc2   = qnn.QuantLinear(120, 84, bias=True, weight_bit_width=4)
        self.relu4 = qnn.QuantReLU(bit_width=4)
        self.fc3   = qnn.QuantLinear(84, 10, bias=True)

    def forward(self, x):
        out = self.quant_inp(x)
        out = self.relu1(self.conv1(out))
        out = F.max_pool2d(out, 2)
        out = self.relu2(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.reshape(out.shape[0], -1)
        out = self.relu3(self.fc1(out))
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out

quant_weight_act_lenet = QuantWeightActLeNet()
```

```
# ... training ...
```

Note a couple of things:

- By default `QuantReLU` is *stateful*, so there is a difference between instantiating one `QuantReLU` that is called multiple times, and instantiating multiple `QuantReLU`` that are each called once.
- `QuantReLU` first computes a relu function, and then quantizes its output. To take advantage of the fact that the output of relu is `>= 0` then, by default `QuantReLU` performs *unsigned* quantization, meaning in this case its output is `int4` data in `[0, 15]`.
- Quantized data in Brevitas is always represented in **dequantized** format, meaning that is represented within a float tensor. The output of *QuantReLU* then looks like a standard float torch Tensor, but it's restricted to *16 different values* (with 4 bits quantization). In order to get a more informative representation of quantized data, we need to set `return_quant_tensor=True`.

Weights, activations, biases quantization

```
from torch.nn import Module
import torch.nn.functional as F

import brevitas.nn as qnn
from brevitas.quant import Int32Bias


class QuantWeightActBiasLeNet(Module):
    def __init__(self):
        super(LowPrecisionLeNet, self).__init__()
        self.quant_inp = qnn.QuantIdentity(bit_width=4, return_quant_tensor=True)
        self.conv1 = qnn.QuantConv2d(3, 6, 5, bias=True, weight_bit_width=4, bias_quant=In
        self.relu1 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.conv2 = qnn.QuantConv2d(6, 16, 5, bias=True, weight_bit_width=4, bias_quant=I
        self.relu2 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.fc1   = qnn.QuantLinear(16*5*5, 120, bias=True, weight_bit_width=4, bias_quan
        self.relu3 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.fc2   = qnn.QuantLinear(120, 84, bias=True, weight_bit_width=4, bias_quant=In
        self.relu4 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.fc3   = qnn.QuantLinear(84, 10, bias=True, weight_bit_width=4, bias_quant=Int

    def forward(self, x):
        out = self.quant_inp(x)
        out = self.relu1(self.conv1(out))
        out = F.max_pool2d(out, 2)
        out = self.relu2(self.conv2(out))
        out = F.max_pool2d(out, 2)
```

```
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out

quant_weight_act_bias_lenet = QuantWeightActBiasLeNet()

# ... training ...
```

Compared to the previous scenario:

- We now set `return_quant_tensor=True` in every quantized activations to propagate a `QuantTensor` to the next layer. This informs each `QuantLinear` or `QuantConv2d` of how the input passed in has been quantized.
- A `QuantTensor` is just a tensor-like data structure providing metadata about how a tensor has been quantized, similar to a *torch.qint* dtype, but training friendly. Setting `return_quant_tensor=True` does not affect the way quantization is performed, it only changes the way the output is represented.
- We enable bias quantization by setting the *Int32Bias* quantizer. What it does is to perform bias quantization with `` `bias_scale = input_scale * weight_scale `` , as it commonly done across inference toolchains. This is why we have to set `return_quant_tensor=True` : each layer with `Int32Bias` can read the input scale from the `QuantTensor` passed in and use for bias quantization.
- `torch` operations that are algorithmically invariant to quantization, such as *F.max_pool2d*, can propagate QuantTensor through them without extra changes.

Export to ONNX

Brevitas does not perform any low-precision acceleration on its own. For that to happen, the model need to be exported first to an inference toolchain through some intermediate representation like ONNX. One popular way to represent 8-bit quantization within ONNX is through the QDQ format. Brevitas extends *QDQ* to **QCDQ**, inserting a *Clip* node to represent quantization to *<= 8 bits*. We can then export all previous defined model to *QCDQ*. The interface of the export function matches the *torch.onnx.export* function, and accepts all its kwargs:

```
from brevitas.export import export_onnx_qcdq
import torch
```

```
# Weight-activation model
export_onnx_qcdq(quant_weight_act_lenet, torch.randn(1, 3, 32, 32), export_path='4b_weight

# Weight-activation-bias model
export_onnx_qcdq(quant_weight_act_bias_lenet, torch.randn(1, 3, 32, 32), export_path='4b_w
```

Where to go from here

Check out the Tutorials section for more information on things like ONNX export, quantized recurrent layers, quantizers, or a more detailed overview of the library in the TVMCon tutorial.