ONNX Export

Requirements

Brevitas requires Python 3.8+ and PyTorch 1.9.1+ and can be installed from PyPI with `pip install brevitas`.

For this notebook, you will also need to install `onnx`, `onnxruntime`, `onnxoptimizer` and `netron` (for visualization of ONNX models). For this tutorial, PyTorch 1.8.1+ is required.

```
[1]: %pip install netron
```

```
Requirement already satisfied: netron in /proj/xlabs/users/nfraser/opt/miniforge3/envs/
Note: you may need to restart the kernel to use updated packages.
```

Introduction

The main goal of this notebook is to show how to use Brevitas to export your models in the two standards currently supported by ONNX for quantized models: QCDQ and QOps (i.e., `QLinearConv`, `QLinearMatMul`). Once exported, these models can be run using onnxruntime.

This notebook doesn't cover QONNX, a custom extension over ONNX with more features for quantization representation that Brevitas can generate as export, which requires the `qonnx` library.

QuantizeLinear-Clip-DeQuantizeLinear (QCDQ)

QCDQ is a style of representation introduced by Brevitas that extends the standard QDQ representation for quantization in ONNX. In Q(C)DQ export, before each operation, two (or three, in case of clipping) extra ONNX nodes are added: - `QuantizeLinear`: Takes as input a FP tensor, and

Takes as input an INT8 tensor, and, given ntenger min/max values, restricts its range. - `DeQuantizeLinear`: Takes as input an INT8 tensor, and converts it to its FP equivalent with a given zero-point and scale factor.

There are several implications associated with this set of operations: - It is not possible to quantize with a bit-width higher than 8. Although `DequantizeLinear` supports both (U)Int8 and Int32 as input, currently `QuantizeLinear` can only output (U)Int8. - Using only `QuantizeLinear` and `DeDuantizeLinear`, it is possible only to quantize to 8 bit (signed or unsigned). - The addition of the `Clip` function between `QuantizeLinear` and `DeQuantizeLinear`, allows to quantize a tensor to bit-width < 8. This is done by Clipping the Int8 tensor coming out of the `QuantizeLinear` node with the min/max values of the desired bit-width (e.g., for unsigned 3 bit, `min_val = 0` and `max_val = 7`). - It is possible to perform both per-tensor and per-channel quantization (requires ONNX Opset >=13).

We will go through all these cases with some examples.

Basic Example

First, we will look at `brevitas.nn.QuantLinear`, a quantized alternative to `torch.nn.Linear`. Simi considerations can also be used for `QuantConv1d`, `QuantConv2d`, `QuantConvTranspose1d` and `QuantConvTranspose2d`.

Brevitas offers several API to export Pytorch modules into several different formats, all sharing the same interface. The three required arguments are: - The PyTorch model to export - A representative input tensor (or a tuple of input args) - The path where to save the exported model

[2]:
```
import netron
import time
from IPython.display import IFrame

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

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)
```

[3]:
```
import brevitas.nn as qnn
```

Skip to main content

```
IN_CH = 3
OUT_CH = 128
BATCH_SIZE = 1

# set seed
torch.manual_seed(0)

linear = qnn.QuantLinear(IN_CH, OUT_CH, bias=True)
inp = torch.randn(BATCH_SIZE, IN_CH)
path = 'quant_linear_qcdq.onnx'

exported_model = export_onnx_qcdq(linear, args=inp, export_path=path, opset_version=13)
```

[4]:
```
show_netron(path, 8082)
```

```
Serving 'quant_linear_qcdq.onnx' at http://localhost:8082
```

[4]:

As it can be seen from the exported ONNX, by default in `QuantLinear` only the weights are quantized, and they go through a Quantize/DequantizeLinear before being used for the `Gemm` operation. Moreover, there is a clipping operation that sets the min/max values for the tensor to ±127. This is because in Brevitas the default weight quantizer (but not the activation one) has the option `narrow_range=True`. This option, in case of signed quantization, makes sure that the quantization interval is perfectly symmetric (otherwise, the minimum integer would be -128), so that it can absorb sign changes (e.g. from batch norm fusion).

Skip to main content

The input and bias remains in floating point. In QCDQ export this is not a problem since the weights, that are quantized at 8 bit, are dequantized to floating-point before passed as input to the `Gemm` node.

Complete Model

A similar approach can be used with entire Pytorch models, rather than single layer.

[4]:
```
class QuantModel(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.linear = qnn.QuantLinear(IN_CH, OUT_CH, bias=True, weight_scaling_per_outp
        self.act = qnn.QuantReLU()

    def forward(self, inp):
        inp = self.linear(inp)
        inp = self.act(inp)
        return inp

model = QuantModel()
inp = torch.randn(BATCH_SIZE, IN_CH)
path = 'quant_model_qcdq.onnx'

exported_model = export_onnx_qcdq(model, args=inp, export_path=path, opset_version=1
```

[6]:
```
show_netron(path, 8083)
```

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

[6]:

We did not specify the argument `output_quant` in our `QuantLinear` layer, thus the output of the layer will be passed directly to the ReLU function without any intermediate re-quantization step.

Furthermore, we have defined a per-channel quantization, so the scale factor will be a Tensor rather than a scalar (ONNX opset >= 13 is required for this).

Finally, since we are using a `QuantReLU` with default initialization, the output is re-quantized as an UInt8 Tensor.

The C in QCDQ (Bitwidth <= 8)

As mentioned, Brevitas export expands on the basic QDQ format by adding the `Clip` operation.

This operations is inserted between the `QuantizeLinear` and `DeQuantizeLinear` node, and thus operates on integers.

Normally, using only the QDQ format, it would be impossible to export models quantize with less than 8 bit.

In Brevitas however, if a quantized layer with bit-width <= 8 is exported, the Clip node will be automatically inserted, with the min/max values computed based on the particular type of quantized

Even though the Tensor data type will still be a Int8 or UInt8, its values are restricted to the desired bit-width.

[5]:
```
class Model(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.linear = qnn.QuantLinear(IN_CH, OUT_CH, bias=True, weight_bit_width=3)
        self.act = qnn.QuantReLU(bit_width=4)

    def forward(self, inp):
        inp = self.linear(inp)
        inp = self.act(inp)
        return inp

model = Model()
model.eval()

inp = torch.randn(BATCH_SIZE, IN_CH)
path = 'quant_model_3b_4b_qcdq.onnx'

exported_model = export_onnx_qcdq(model, args=inp, export_path=path, opset_version=13)
```

[8]:
```
show_netron(path, 8084)
```

```
Serving 'quant_model_3b_4b_qcdq.onnx' at http://localhost:8084
```

[8]:

As can be seen from the generated ONNX, the weights of the `QuantLinear` layer are clipped

Skip to main content

`narrow_range=True`.

Similarly, the output of the QuantReLU is clipped between 0 and 15, since in this case we are doing an unsigned 4 bit quantization.

Even when using `QLinearConv` and `QLinearMatMul`, it is still possible to represent bit-width < 8 through the use of clipping.

However, in this case the `Clip` operation over the weights won't be captured in the exported ONNX graph. Instead, it will be performed at export-time, and the clipped tensor will be exported in the ONNX graph.

Examining the last exported model, it is possible to see that the weight tensor, even though it has Int8 has type, has a min/max values equal to `[-7, 7]`, given that it is quantized at 4 bit with narrow_range set to True.

ONNX Runtime

QCDQ

Since for QCDQ we are only using standard ONNX operation, it is possible to run the exported model using ONNX Runtime.

[6]:
```
import onnxruntime as ort

class Model(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.linear = qnn.QuantLinear(IN_CH, OUT_CH, bias=True, weight_bit_width=3)
        self.act = qnn.QuantReLU(bit_width=4)

    def forward(self, inp):
        inp = self.linear(inp)
        inp = self.act(inp)
        return inp

model = Model()
```

Skip to main content

```
path = 'quant_model_3b_4b_qcdq.onnx'

exported_model = export_onnx_qcdq(model, args=inp, export_path=path, opset_version=13)

sess_opt = ort.SessionOptions()
sess = ort.InferenceSession(path, sess_opt)
input_name = sess.get_inputs()[0].name
pred_onx = sess.run(None, {input_name: inp.numpy()})[0]


out_brevitas = model(inp)
out_ort = torch.tensor(pred_onx)

assert_with_message(torch.allclose(out_brevitas, out_ort))
```

```
True
2025-05-09 15:50:06.990808285 [W:onnxruntime:, graph.cc:1283 Graph] Initializer linear.
```

QGEMM vs GEMM

QCDQ allows to execute low precision fake-quantization in ONNX Runtime, meaning operations actually happen among floating-point values. ONNX Runtime is also capable of optimizing and accelerating a QCDQ model leveraging a int8 based QGEMM kernels in some scenarios.

This seems to happen only when using a `QuantLinear` layer, with the following requirements: - Input, Weight, Bias, and Output tensors must be quantized; - Bias tensor must be present, and quantized with bitwidth > 8. - The output of the QuantLinear must be re-quantized. - The output bit-width must be equal to 8. - The input bit-width must be equal to 8. - The weights bit-width can be <= 8. - The weights can be quantized per-tensor or per-channel.

We did not observe a similar behavior for other operations such as `QuantConvNd`.

An example of a layer that will match this definition is the following:

[7]:
```
from brevitas.quant.scaled_int import Int32Bias
from brevitas.quant.scaled_int import Int8ActPerTensorFloat

qgemm_ort = qnn.QuantLinear(
    IN_CH, OUT_CH,
    weight_bit_width=5,
    input_quant=Int8ActPerTensorFloat,
    output_quant=Int8ActPerTensorFloat,
    bias=True, bias_quant=Int32Bias)
```

Unfortunately ONNX Runtime does not provide a built-in way to log whether execution goes through unoptimized floating-point GEMM, or int8 QGEMM.

Export Dynamically Quantized Models to ONNX

You can also export dynamically quantized models to ONNX, but there are some limitations. The ONNX DynamicQuantizeLinear requires the following settings: - Asymmetric quantization (and therefore *unsigned*) - Min-max scaling - Rounding to nearest - Per tensor scaling - Bit width set to 8

This is shown in the following example:

[8]:
```
from brevitas_examples.common.generative.quantizers import ShiftedUint8DynamicActPerTer

IN_CH = 3
IMG_SIZE = 128
OUT_CH = 128
BATCH_SIZE = 1

class Model(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.linear = qnn.QuantLinear(IN_CH, OUT_CH, bias=True, weight_bit_width=8,
        self.act = qnn.QuantReLU(input_quant=ShiftedUint8DynamicActPerTensorFloat)

    def forward(self, inp):
        inp = self.linear(inp)
        inp = self.act(inp)
        return inp

inp = torch.randn(BATCH_SIZE, IN_CH)
model = Model()
model.eval()
path = 'dynamic_quant_model_qcdq.onnx'

exported_model = export_onnx_qcdq(model, args=inp, export_path=path, opset_version=13)
```

[9]:
```
show_netron("dynamic_quant_model_qcdq.onnx", 8086)
```

```
Serving 'dynamic_quant_model_qcdq.onnx' at http://localhost:8086
```

[9]: