AOC Lab2 Quantization

Advisor：CCTsai

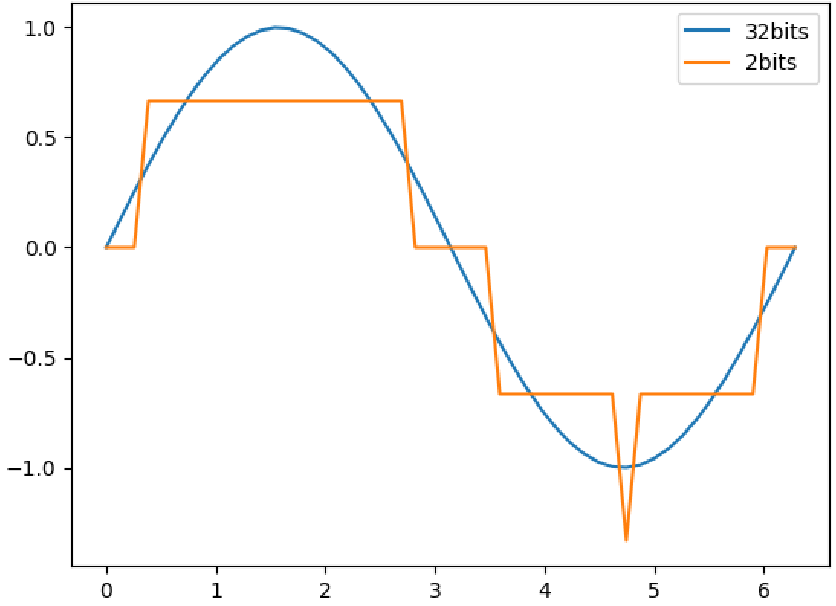
TA：林泳陞

Outline

- Introduction
  - Quantization
  - PTQ and QAT
- Task
  - Example code
  - Task

Introduction - Quantization

- Quantization is a method used to reduce the model size and computation requirements by replacing floating-point 32-bit (FP32) representations with lower bit precision, such as FP16, INT8 or even less bits. However, reducing the number of bits used to represent values can lead to a loss of accuracy. The primary objective is to strike a balance between accuracy and computational resources.



Introduction - Quantization

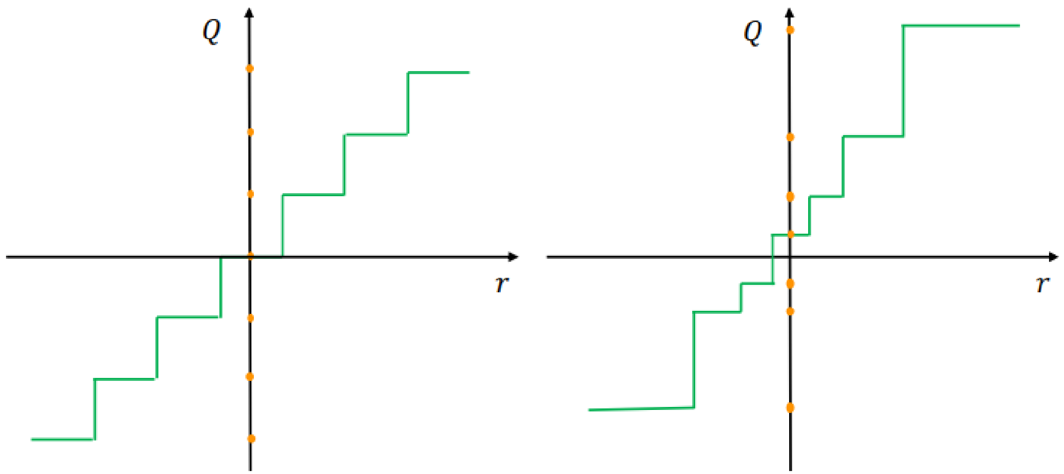


Figure 1: *Comparison between uniform quantization (left) and non-uniform quantization (right). Real values in the continuous domain r are mapped into discrete, lower precision values in the quantized domain Q, which are marked with the orange bullets. Note that the distances between the quantized values (quantization levels) are the same in uniform quantization, whereas they can vary in non-uniform quantization.*

Introduction - Quantization

- The simplest way to implement quantization is to map the original values to an integer range of -128 to 127, and then map these 256 integers back to the original value range to minimize the error.
- For example:

```
Original output:
 tensor([ 1.3742, -2.7129,  2.9312, -1.1382,  3.8776, -1.1048,  4.1834, -6.2159,
          4.9783, -6.4220], grad_fn=<SelectBackward0>)
Quantized output:
 tensor([ -4., -45.,  11., -30.,  21., -29.,  24., -80.,  32., -82.])
Quantized output transform to float:
 tensor([ 1.4014, -2.7027,  2.9029, -1.2012,  3.9039, -1.1011,  4.2042, -6.2062,
          5.0050, -6.4064])
```

- Therefore, we need to find a method to implement quantization using scale factor and zero point.

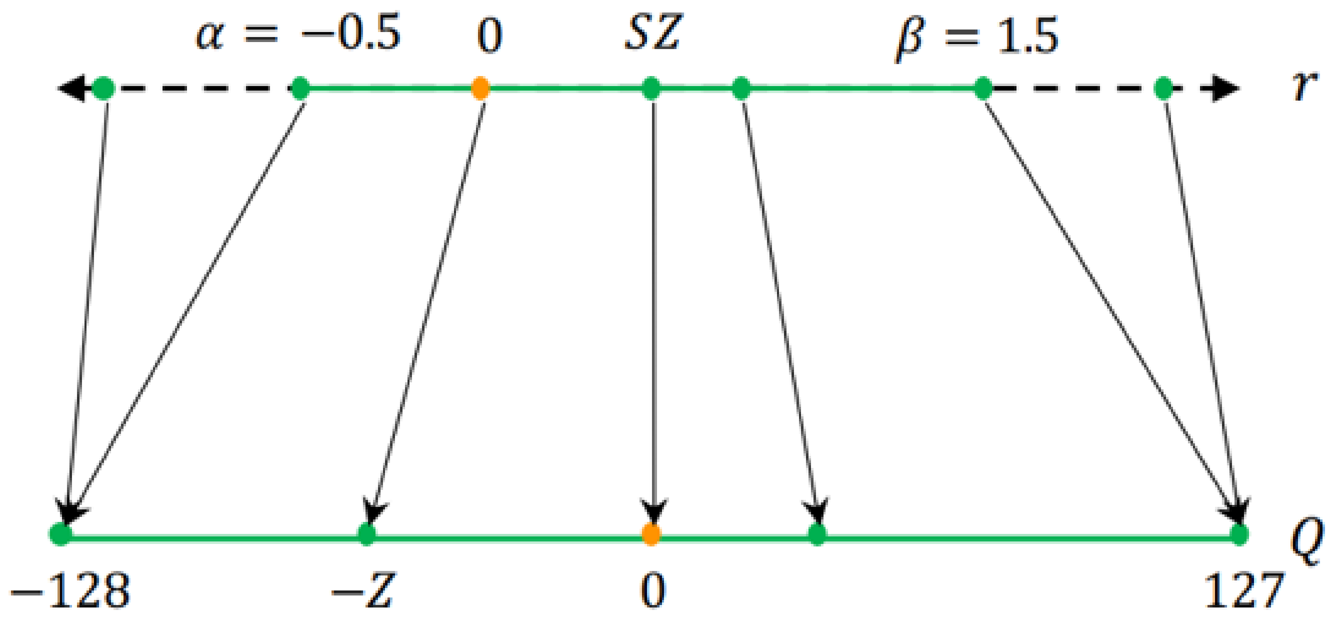
Introduction - Quantization

- q = round($\frac{r}{s} + Z$), where $s = \frac{\beta - \alpha}{\beta_q - \alpha_q}$ and $Z = round(\alpha_q - \frac{\alpha}{s})$
- For an arbitrary tensor, [α, β] is the tensor range, and [α_q, β_q] is usually [-128, 127] for int8 quantization.
- r_q = (q – Z) * s

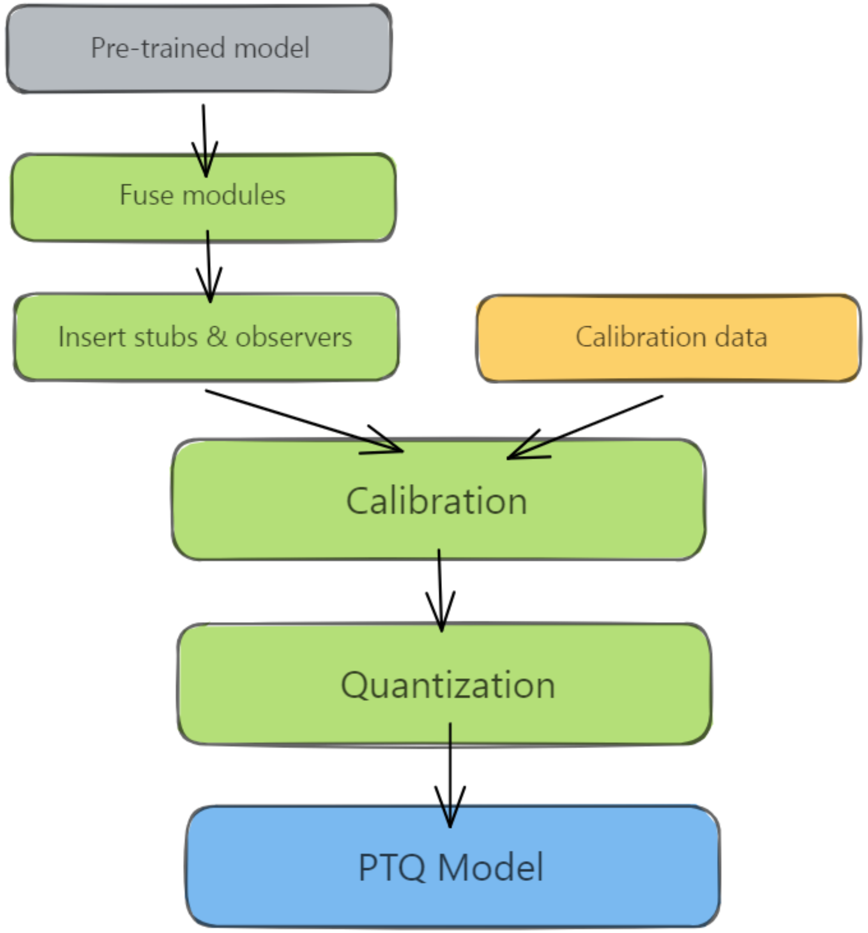
```
Original output:
 tensor([ 1.3742, -2.7129,  2.9312, -1.1382,  3.8776, -1.1048,  4.1834, -6.2159,
          4.9783, -6.4220], grad_fn=<SelectBackward0>)
Quantized output:
 tensor([ -4., -45.,  11., -30.,  21., -29.,  24., -80.,  32., -82.])
Quantized output transform to float:
 tensor([ 1.4014, -2.7027,  2.9029, -1.2012,  3.9039, -1.1011,  4.2042, -6.2062,
          5.0050, -6.4064])
```

Introduction - Quantization

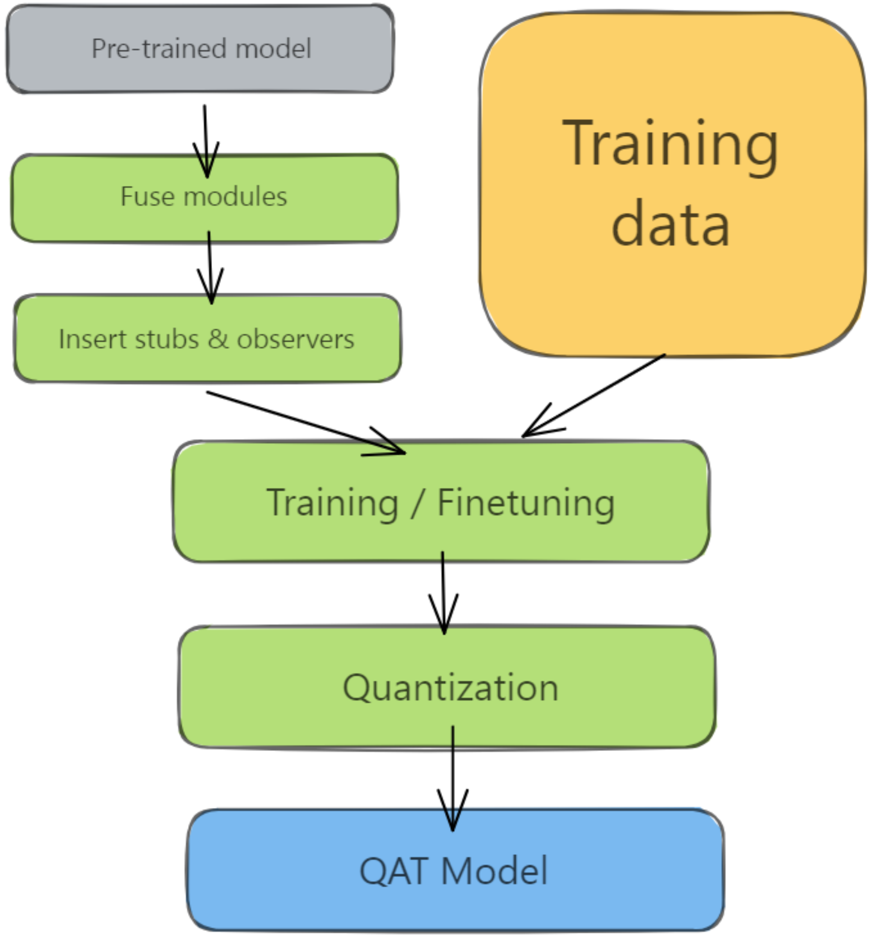
- Some details about scale factor and zero point
- When r is not symmetric, Z is not 0.



Introduction - PTQ



Introduction - QAT



Task

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2), \qquad (4)$$

where the *multiplier* M is defined as

$$M := \frac{S_1 S_2}{S_3}. \qquad (5)$$

$$Q_{output} = \mathrm{Linear}[q_{input}-Z_{input}, q_{weight}-Z_{weight}] \cdot (S_{input}S_{weight}/S_{output}) + Z_{output}$$

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \text{ and } Z = round(\alpha_q - \frac{\alpha}{s})$$

M在硬體實現的補充說明

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2),$$

where the *multiplier* M is defined as

$$M := \frac{S_1 S_2}{S_3}.$$

$$M = 2^{-n} M_0$$

M_0 = [0.5, 1)
n is a non-negative
integer

In Equation (4), the only non-integer is the multiplier M. As a constant depending only on the quantization scales S_1, S_2, S_3, it can be computed offline. We empirically find it to always be in the interval $(0, 1)$, and can therefore express it in the normalized form

$$M = 2^{-n} M_0 \tag{6}$$

where M_0 is in the interval $[0.5, 1)$ and n is a non-negative integer. The normalized multiplier M_0 now lends itself well to being expressed as a fixed-point multiplier (e.g. int16 or int32 depending on hardware capability). For example, if int32 is used, the integer representing M_0 is the int32 value nearest to $2^{31} M_0$. Since $M_0 \geqslant 0.5$, this value is always at least 2^{30} and will therefore always have at least 30 bits of relative accuracy. Multiplication by M_0 can thus be implemented as a fixed-point multiplication[4]. Meanwhile, multiplication by 2^{-n} can be implemented with an efficient bit-shift, albeit one that needs to have correct round-to-nearest behavior, an issue that we return to in Appendix B.

Example code

截圖放在Report

Question 1.

Use

$$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$$

$$Z = q_{\min} - r_{\min}/S$$

to calculate scale factor and zero point of a tensor

```python
def get_scale_and_zero_point(fp32_tensor, bitwidth=8):
    q_min, q_max = -2**(bitwidth-1), 2**(bitwidth-1) - 1
    fp_min = fp32_tensor.min().item()
    fp_max = fp32_tensor.max().item()

    ############################################################

    scale = ( __ - __ ) / ( __ - __ )
    zero_point = __ - __ / __

    ############################################################

    zero_point = round(zero_point)                 #round
    zero_point = max(q_min, min(zero_point, q_max)) #clip

    return scale, int(zero_point)
```

Example code

截圖放在Report

Question 2.

Use $q = r/S + Z$ to quantize a tensor

```
[ ] def linear_quantize(fp32_tensor, bitwidth=8):
        q_min, q_max = -2**(bitwidth-1), 2**(bitwidth-1) - 1

        scale, zero_point = get_scale_and_zero_point(fp32_tensor)

        ###################################################

        q_tensor = torch.round( __ / __ ) + __

        ###################################################

        q_tensor = torch.clamp(q_tensor, q_min, q_max)
        return q_tensor, scale, zero_point
```

Example code

Question 3.

Use

$$q_{\text{output}} = M * \text{Linear}[q_{\text{input}}, q_{\text{weight}}] + Z_{\text{output}}$$

$$M = S_{\text{input}} * S_{\text{weight}} / S_{\text{output}}$$

to compute quantized linear operation

截圖放在Report

```
def quantized_linear(input, weights, input_scale, weight_scale, output_scale, input_zero_point, weight_zero_point, output_zero_point, device, bitwidth=8, activation_bitwidth=8):
    input, weights = input.to(device), weights.to(device)
    ############################################################

    M = __ * __ / __
    output = torch.nn.functional.linear((input - __ ), (weights - __ ))
    output *= M
    output += output_zero_point

    ############################################################
    output = output.round().clamp(-2**(activation_bitwidth-1), 2**(activation_bitwidth-1)-1)

    return output
```

Example code

```
test_loop(test_loader, FP32_model, loss_fn)
```

```
Test Error:
 Accuracy: 83.9%, Avg loss: 0.000875
```

```
[ ] test_loop(test_loader, quantized_model, loss_fn)
```

```
Test Error:
 Accuracy: 83.9%, Avg loss: 0.004596
```

Task

- Practice to implement quantization function 50% (每少一張截圖-5%)
  - Question 1. (15%)
  - Question 2. (15%)
  - Question 3. (20%)
- Problem 50%
  - What is the size of the model after int8 quantization if its original size is 50MB? Please write down your calculation process. (Assume the original resolution is 32 bits) (15%)
  - If M = 0.2, determine values for M0 and n such that the equation on page 11 is true. (10%)

Task

- 閱讀“Quantization and Training of Neural Networks for Efficient Integer Arithmetic-Only Inference”. 並根據這篇論文的理論闡述，說明在軟硬體實作上，要怎麼將其理論做實際的應用?(僅說明理論不會有分數，理論說明請用自己的話闡述). 例如: M 在硬體上如何近似處理及如何和其他post-processing的步驟搭配，Batch normalization 在軟體上可以怎麼實現folding，軟體上怎麼實現fuse layer 等等 .....其他不同的面向 (20%)

- Share your thoughts on this lab, any advice or improvement on codes, tutorials, or other ideas about quantization. (5%) 有認真表達心得一律滿分

File Format

- Lab2_StudentID_Name.zip / .tar
  - Lab2_StudentID_Name.pdf — Report
  - Lab2_StudentID_Name.ipynb — code

Reference

- Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference : https://arxiv.org/abs/1712.05877

- A Survey of Quantization Methods for Efficient Neural Network Inference : https://arxiv.org/abs/2103.13630

影片連結

- PPT: PPT講解

- Code: Code講解