Model Quantization

Yu-Chi Chu

21 Mar 2024

Post-training quantization

Post-Training Quantization Implementation Process:

1) Fusing the modules

   This step involves merging consecutive layers, such as [Conv, BN] or [Conv, BN, ReLU], or other layer combinations, into a single module. This is done to improve computational speed and accuracy. It's important to note that not all operations can be fused. Here are some common combinations suitable for fusion:

   - Convolution, Batch normalization
   - Convolution, Batch normalization, ReLU
   - Convolution, ReLU
   - Linear, ReLU
   - Batch normalization, ReLU

2) Setting qconfig

   qconfig is set for the model or its sub-modules. qconfig includes two types of observers: a weight observer and an activation observer. Simply put, an observer finds suitable scale and zero-point (zp) values for the quantization formula. Different qconfig modes should be selected depending on the backend (hardware device used). For x86 architecture, get_default_qconfig('fbgemm') should be used, while get_default_qconfig('qnnpack') should be used for ARM architecture. The following table shows the corresponding activation and weight observers for different backends.

量化的 backend	activation	weight
fbgemm	HistogramObserver (reduce_range=True)	PerChannelMin MaxObserver (default_per_channel _weight_observer)
qnnpack	HistogramObserver (reduce_range=False)	MinMaxObserver (default_weight _observer)
默认（非 fbgemm和qnnpack）	MinMaxObserver (default_observer)	MinMaxObserver (default_weight _observer)

3) Prepare model for quantization

   prepare is used to insert observers into each sub-module to collect and calibrate data. Among various observers, calculating the scale and zero-point (zp) for weights relies on four variables: min_val, max_val, qmin, and qmax. These represent the minimum and maximum values of the operation's weight data/input tensor data distribution, and the minimum and maximum values of the quantized range, respectively. Therefore, we expect the observer to obtain

these four variables after observing the input data and use these four variables to determine the scale and zp.

4) Feeding data

This step is not for training. It is to obtain the distribution characteristics of the data to better calculate the scale and zp of the activations. Here, example_inputs = (next(iter(trainloader))[0]) is used to feed the data. In my understanding, this step can also be called calibration.

5) Convert the model

The diagram below shows the schematic of the layers before and after quantization:

```
#原始的模型，所有的tensor和计算都是浮点型
previous_layer_fp32 -- linear_fp32 -- activation_fp32 -- next_layer_fp32
                      /
    linear_weight_fp32

#静态量化的模型，权重和输入都是int8
previous_layer_int8 -- linear_with_activation_int8 -- next_layer_int8
                      /
  linear_weight_int8
```

Summary of PTQ:

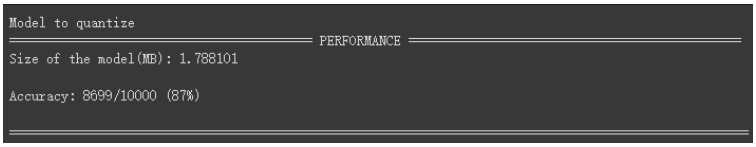
```
ResNet(
  (conv1): Sequential(
    (0): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv2): Sequential(
    (0): Conv2d(8, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(12, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv3): Sequential(
    (0): Conv2d(12, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv4): Sequential(
    (0): Conv2d(20, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res1): Sequential(
    (0): Sequential(
      (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
```

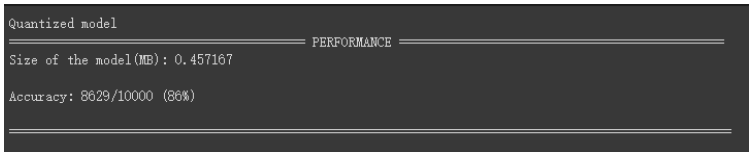
Model Before Quantization

```
GraphModule(
  (conv1): Module(
    (0): QuantizedConvReLU2d(3, 8, kernel_size=(3, 3), stride=(1, 1), scale=0.04003132879734039, zero_point=0, padding=(1, 1))
  )
  (conv2): Module(
    (0): QuantizedConvReLU2d(8, 12, kernel_size=(3, 3), stride=(1, 1), scale=0.038154106587171555, zero_point=0, padding=(1, 1))
  )
  (conv3): Module(
    (0): QuantizedConvReLU2d(12, 20, kernel_size=(3, 3), stride=(1, 1), scale=0.030395518988370895, zero_point=0, padding=(1, 1))
  )
  (conv4): Module(
    (0): QuantizedConvReLU2d(20, 32, kernel_size=(3, 3), stride=(1, 1), scale=0.030311226844787598, zero_point=0, padding=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res1): Module(
    (0): Module(
      (0): QuantizedConvReLU2d(32, 32, kernel_size=(3, 3), stride=(1, 1), scale=0.014520660974085331, zero_point=0, padding=(1, 1))
    )
    (1): Module(
      (0): QuantizedConvReLU2d(32, 32, kernel_size=(3, 3), stride=(1, 1), scale=0.02702411636710167, zero_point=0, padding=(1, 1))
    )
  )
  (conv5): Module(
    (0): QuantizedConvReLU2d(32, 48, kernel_size=(3, 3), stride=(1, 1), scale=0.037790760397911072, zero_point=0, padding=(1, 1))
  )
  (conv6): Module(
    (0): QuantizedConvReLU2d(48, 64, kernel_size=(3, 3), stride=(1, 1), scale=0.10955063253641129, zero_point=0, padding=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
```

Model After Quantization

Based on the above two figures, it can be observed that the model has been correctly quantized, and the original [Conv, BN, ReLU] modules have been compressed into smaller modules, resulting in a significant reduction in model size.





The number of parameters in the quantized model is one-fourth of that before quantization (because 32-bit floating point is converted to INT8-bit, hence one-fourth). As expected from theory, the accuracy of the quantized model is slightly reduced.

Quantization aware training

Quantization aware training 實作流程分成五步驟:

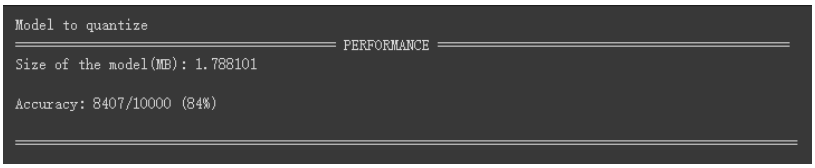
1) Setting qconfig
   Before setting the qconfig, the model needs to be set to train mode. The qconfig here is different from that in PTQ. In QAT's qconfig, the activation and weight observers are replaced with FakeQuantize. This inserts extra fake quantization operators into the operators in the model, keeping the calculations in floating-point during training, but using clamping and rounding techniques to approximate the INT8 operation mode. The observer included in FakeQuantize is MovingAverageMinMaxObserver, which is different from PTQ.
2) Fuse modules, as described above (in PTQ).
3) Prepare qat
   This step installs the qconfig onto every operation, which means inserting fake observers.
4) Feeding data
   Input data is fed to the model for training. The training process uses floating-point calculations, but the calculations approximate INT8 behavior. Therefore, although the entire training process is floating-point computation, it is called quantization aware training because the training process itself is aware of simulating integer operations.
5) Convert, as in PTQ.

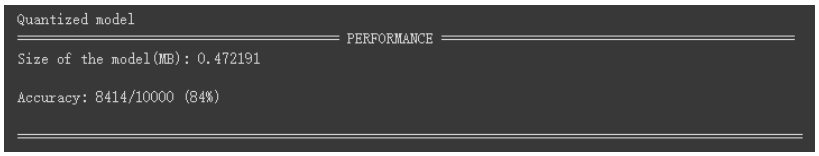
```
# original model
# all tensors and computations are in floating point
previous_layer_fp32 -- linear_fp32 -- activation_fp32 -- next_layer_fp32
                       /
    linear_weight_fp32

# model with fake_quants for modeling quantization numerics during training
previous_layer_fp32 -- fq -- linear_fp32 -- activation_fp32 -- fq -- next_layer_fp32
                            /
   linear_weight_fp32 -- fq

# quantized model
# weights and activations are in int8
previous_layer_int8 -- linear_with_activation_int8 -- next_layer_int8
                       /
   linear_weight_int8
```

Overall Flow Diagram

Summary of QAT:

During the training stage (for QAT), setting the original training model's EPOCH to 1 is sufficient. I tried setting the EPOCH to 5, but the training effect clearly saturated at EPOCH 1.





It can be seen that the accuracy of the pre-quantized model is consistent with the accuracy of the quantized model. This phenomenon demonstrates the advantage of QAT: in addition to reducing the number of model parameters by a quarter, the accuracy is also preserved.

Analyzing two types of quantization results

In the PTQ experiments, it can be seen that the accuracy of the quantized model is slightly lower than that of the pre-quantized model. However, the accuracy of QAT remains unchanged, thus demonstrating that QAT is a better quantization tool. The biggest difference between PTQ and QAT is that the "P" in PTQ stands for "post," meaning that the model is quantized *after* it has been trained. In contrast, QAT enables quantization *during* the training process, and this training process does not require as much time as training a pre-trained model.

Reference:

1. https://github.com/Sanjana7395/static_quantization/blob/master/quantization%20pytorch.ipynb
2. https://blog.csdn.net/c9Yv2cf9I06K2A9E/article/details/113488003?ops_request_misc=&request_id=&biz_id=102&utm_term=post%20training%20static%20quantizat&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-4113488003.142^v74^wechat,201^v4^add_ask,239^v2^insert_chatgpt&spm=1018.2226.3001.4187
3. https://pytorch.org/docs/stable/quantization.html