BOOTCAMP WEEK 05

TECHNICAL REPORT

> **Reported by: NGUYEN XUAN QUANG**

I. Baseline models

We decide to study how to apply post-training compression to the `ResNet-34` architecture. Notably, ResNet-34 has 3x3 filters in each of its modular layers with shortcut connections being 1x1. The filter width ranges from 64 to 512 as depth increases. We train the original pre-activation variant of ResNet implementation using full-precision `32 bits` on the CIFAR-100 dataset. After training in `60 epochs`, the model achieves an accuracy of `73.15%`, the top-1 and top-5 error rate are `26.85%` and `6.75%`, respectively.

II. Method

Because of time constraints, we decide to study only the Quantization technique, according to it simplicity and efficiency in reducing model size and computation complexity

1. Post-training Dynamic Quantization

This is the simplest form of quantization where the weights are quantized ahead of time but the activations are dynamically quantized during inference. However, Pytorch does not support Dynamic Quantization for its `nn.Conv1d/2d/3d` layers. So that this method may not work well with the CNNs architecture.

In this work, we dynamically quantize our baseline (fp32) to `16-bit (fp16)` and `8-bit (int8) precision` and see how it helps improve model size, inference time and memory usage.

2. Post-training Static Quantization

PyTorch supports quantizing both the weights and activations of the model beforehand, in which the activations (e.g. ReLU) are fused into preceding layers. According to Pytorch documents, this techniques works best for the CNNs, provides best performance although it may have impact on accuracy. \

A tricky detail is that since ResNet has skip connections, which uses the `torch.add` implementation, we have to replace this operation with the `FloatFunctional.add_relu()`. This change is a result of layer fusions. Without it, there will be no activation quantization for skip connection additions. Following PyTorch instrucions, the quantization workflow is as follow:

1. Switch the baseline model to CPU and `eval` mode
2. Fuse model layers, specifically `Conv2d + BatchNorm + ReLU` and `Conv2d + BatchNorm`
3. Add `torch.quantization.QuantStub()` to the inputs and `torch.quantization.DeQuantStub()` to the outputs
4. Specify quantization config, which is the default `fbgemm` config, and prepare the model based on the config using `torch.quantization.prepare()`
5. Calibrate the model on a representative dataset, which is the CIFAR-100 train set in our work.
6. Convert the calibratd floating point model to a quantized int8 model.
7. Evaluate the quantized model.

3. Quantization-aware Training

We also implement a QAT version for our baseline model. According to PyTorch documents, this method results in the highest accuracy as the model adds "fake quantized" layers during the training process. Although all the computations are done in floating points, the model still "awares" that it will ultimately be quantized.

However, we cannot fine-tune the model successfully since it requires more memory than regular training process as additional modules (fake quantized modules) are inserted during training. All of the experiments lead to `CUDA out of memory` error because of the hardware constraints.

III. Experiments

1. Settings

All of our experiements are executed on Kaggle notebook, which provides a P100 GPU with a memory of 16GB and an Intel Xeon 2.20 GHz 4 cores CPU with a memory of 32GB. As for the virtual environment, we use Python 3.10, PyTorch 2.0, and CUDA version 11.8. The batch size is set to 32 for training and 8 for testing, and the learning rate is constantly assigned to 0.1.

2. Results and Analysis

Because PyTorch does not support CUDA quantization inference currently, we carry out all of the evaluations solely on the CPU platform for the sake of fairness, albeit the baseline model may run more efficiently on the CUDA backend.

a) Baseline

- Model size: 85.5 MB
- Top-1 Accuracy: 0.7315
- Top-5 Error Rate: 0.0675
- Inference time (CPU): 53.55 ms/sample
- Memory usage:

```
-------------------------------  ------------  ------------  ------------
                           Name       CPU Mem  Self CPU Mem    # of Calls
-------------------------------  ------------  ------------  ------------
                    aten::empty      14.19 Mb      14.19 Mb           288
                   aten::conv2d       3.91 Mb           0 b            36
              aten::convolution       3.91 Mb           0 b            36
             aten::_convolution       3.91 Mb           0 b            36
               aten::batch_norm       3.91 Mb           0 b            36
   aten::_batch_norm_impl_index       3.91 Mb           0 b            36
        aten::native_batch_norm       3.91 Mb     -59.25 Kb            36
               aten::empty_like       3.91 Mb     384.00 Kb            36
       aten::mkldnn_convolution       2.75 Mb           0 b            17
                      aten::add       1.72 Mb       1.72 Mb            16
-------------------------------  ------------  ------------  ------------
Self CPU time total: 51.758ms
```

b) Dynamic Quantized Float16 Model

- Model size: 85.5 MB
- Top-1 Accuracy: 0.7315
- Top-5 Error Rate: 0.0675
- Inference time (CPU): 47.17 ms/sample
- Memory usage:

```
---------------------------------  ------------  ------------  ----------
-
                             Name       CPU Mem  Self CPU Mem    # of
Calls
---------------------------------  ------------  ------------  ----------
-
                      aten::empty      14.18 Mb      14.18 Mb
289
                     aten::conv2d       3.91 Mb           0 b
36
                aten::convolution       3.91 Mb           0 b
36
               aten::_convolution       3.91 Mb           0 b
36
                 aten::batch_norm       3.91 Mb      32.00 Kb
36
     aten::_batch_norm_impl_index       3.91 Mb           0 b
36
          aten::native_batch_norm       3.91 Mb     -64.00 Kb
36
                 aten::empty_like       3.91 Mb     256.00 Kb
36
         aten::mkldnn_convolution       2.75 Mb           0 b
17
                        aten::add       1.72 Mb       1.72 Mb
16
---------------------------------  ------------  ------------  ----------
-
Self CPU time total: 43.497ms
```

c) Dynamic Quantized Int8 Model

- Model size: 85.3 MB
- Top-1 Accuracy: 0.7303
- Top-5 Error Rate: 0.0673
- Inference time (CPU): 49.10 ms/sample
- Memory usage:

```
-------------------------------  ------------  ------------  ------------
                           Name       CPU Mem  Self CPU Mem    # of Calls
-------------------------------  ------------  ------------  ------------
```

```
                   aten::empty      14.07 Mb      14.07 Mb           290
              aten::empty_like       3.91 Mb     512.00 Kb            37
                  aten::conv2d       3.91 Mb     128.00 Kb            36
             aten::convolution       3.91 Mb           0 b            36
            aten::_convolution       3.91 Mb           0 b            36
              aten::batch_norm       3.91 Mb           0 b            36
  aten::_batch_norm_impl_index       3.91 Mb           0 b            36
       aten::native_batch_norm       3.91 Mb     -61.75 Kb            36
      aten::mkldnn_convolution       2.75 Mb           0 b            17
                     aten::add       1.72 Mb       1.72 Mb            16
-------------------------------  ------------  ------------  ------------
Self CPU time total: 49.732ms
```

d) Static Quantized Int8 Model

Note that we have serialized this model into TorchScript using the JIT compiler. Thus, the original PyTorch model may have slight differences in performance compared to this below statistic.

- Model size: 21.6 MB
- Top-1 Accuracy: 0.7309
- Top-5 Error Rate: 0.0677
- Inference time (CPU): 20.55 ms/sample
- Memory usage:

```
---------------------------------  ------------  ------------  ------------
                             Name       CPU Mem  Self CPU Mem    # of Calls
---------------------------------  ------------  ------------  ------------
                      aten::empty       3.84 Mb       3.84 Mb            38
   aten::_empty_affine_quantized       1.41 Mb       1.41 Mb            55
          quantized::conv2d_relu     501.00 Kb      -1.97 Mb            17
               quantized::conv2d      24.00 Kb      -2.34 Mb            19
       aten::quantize_per_tensor       3.00 Kb       3.00 Kb             1
                 aten::contiguous       3.00 Kb           0 b             1
                      aten::clone       3.00 Kb           0 b             1
       aten::adaptive_avg_pool2d         512 b           0 b             1
      aten::_adaptive_avg_pool2d         512 b           0 b             1
                          forward         400 b    -112.50 Kb             1
---------------------------------  ------------  ------------  ------------
Self CPU time total: 19.874ms
```

Analysis

Model	Model Size (MB)	Accuracy (%)	Top-5 Error (%)	CPU Inference Time (ms)	Memory (MB)
Baseline (ResNet-34)	85.5	**73.15**	6.75	53.55	14.19
Dynamic Float16	85.5	**73.15**	6.75	47.17	14.18

Model	Model Size (MB)	Accuracy (%)	Top-5 Error (%)	CPU Inference Time (ms)	Memory (MB)
Dynamic Int8	85.3	73.03	**6.73**	49.10	14.07
Static Int8 (JIT)	**21.6**	73.09	6.77	**20.55**	**5.75**

Based on the data presented in the table, two main conclusions can be drawn:

- **Post-training Dynamic Quantization** works but is limited to the only *Linear* layers used in ResNet. Therefore, the resulting improvements in model size and inference latency are just a few percent. The overall computation complexity is approximately the same as the baseline model.
- **Post-training Static Quantization**, on the other hand, demonstrates the most significant improvements accross all aspects, including model size, memory usage, and inference time. Both weights and activations are converted to 8-bit ints, which helps reduce the model size by `74.74%`, i.e. `4 times compression`. Furthermore, the inference on CPU is `2.5 times faster` while maintaining the prediction `accuracy within 0.1% of the original model`.

IV. Conclusion

In this study, we investigate multiple quantization techniques available in PyTorch and their application to compress our baseline ResNet-34 model. Through our analysis, we have found that Static Quantization emerged as the most promising technique, delivering significantly faster inference performance on the CPU, with a little loss in accuracy.

Notably, all of our measurements are conducted solely on CPU since PyTorch only supports quantization inference for this platform, which is a crucial limitation. In the future, we are going to investigate quantization on CUDA, leveraging the capabilities of the `TensorRT` engine.