Computer Vision for Embedded Systems

Yung-Hsiang Lu
Purdue University
yunglu@purdue.edu

PURDUE UNIVERSITY

Why Quantization

from 32-bit floating point to 8-bit integer:

- 75% reduction in memory requirements (32 ⇨ 8)
- 50% - 75% reduction in memory bandwidth
- 50% - 75% reduction in execution time
- usually at lower accuracy
- PyTorch quantized models are traceable and scriptable
- can mix quantized and floating point operations in a model

https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

Quantization in PyTorch

Three types of quantization

- Dynamic: FP values are stored in memory. convert to int before computation. `torch.quantization.quantize_dynamic`
- Static: Observers, Operator fusion, Per-channel quantization. `torch.quantization.fuse_modules`, `torch.quantization.prepare`, `torch.quantization.convert`
- Quantization-Aware Training (QAT): use FP in training, forward pass round to int. `torch.quantization.prepare_qat`, `torch.quantization.convert`
- quantized operators are supported only for CPU inference

Select Quantization

factors: speed / accuracy requirements, supported operators

Model Type	Preferred scheme	Why
LSTM/RNN	Dynamic Quantization	Throughput dominated by compute/memory bandwidth for weights
BERT/Transformer	Dynamic Quantization	Throughput dominated by compute/memory bandwidth for weights
CNN	Static Quantization	Throughput limited by memory bandwidth for activations
CNN	Quantization Aware Training	In the case where accuracy can't be achieved with static quantization

LSTM: Long short-term memory, RNN: Recurrent neural network, BERT: Bidirectional Encoder Representations from Transformers

Performance (Time)

Model	Float Latency (ms)	Quantized Latency (ms)	Inference Performance Gain	Device	Notes
BERT	581	313	1.8x	Xeon-D2191 (1.6GHz)	Batch size = 1, Maximum sequence length= 128, Single thread, x86-64, Dynamic quantization
Resnet-50	214	103	2x	Xeon-D2191 (1.6GHz)	Single thread, x86-64, Static quantization
Mobilenet-v2	97	17	5.7x	Samsung S9	Static quantization, Floating point numbers are based on Caffe2 run-time and are not optimized

Model	Top-1 Accuracy (Float)	Top-1 Accuracy (Quantized)	Quantization scheme
Googlenet	69.8	69.7	Static post training quantization
Inception-v3	77.5	77.1	Static post training quantization
ResNet-18	69.8	69.4	Static post training quantization
Resnet-50	76.1	75.9	Static post training quantization
ResNext-101 32x8d	79.3	79	Static post training quantization
Mobilenet-v2	**71.9**	**71.6**	Quantization Aware Training
Shufflenet-v2	69.4	68.4	Static post training quantization
BERT	0.902	0.895	Dynamic quantization

Eager vs. FX Quantization

	Eager Mode Quantization	FX Graph Mode Quantization
Release Status	beta	prototype
Operator Fusion	Manual	Automatic
Quant/DeQuant Placement	Manual	Automatic
Quantizing Modules	Supported	Supported
Quantizing Functionals/Torch Ops	Manual	Automatic
Support for Customization	Limited Support	Fully Supported

https://pytorch.org/docs/stable/quantization.html

	Eager Mode Quantization	FX Graph Mode Quantization
Quantization Mode Support	Post Training Quantization: Static, Dynamic, Weight Only Quantization Aware Training: Static	Post Training Quantization: Static, Dynamic, Weight Only Quantization Aware Training: Static
Input/Output Model Type	torch.nn.Module	torch.nn.Module (May need some refactors to make the model compatible with FX Graph Mode Quantization)
When to use	when execution time is dominated by loading weights from memory rather than matrix multiplications	

Eager Mode Quantization

Dynamic Quantization

```
import torch

# define a floating point model
class M(torch.nn.Module):
    def __init__(self):
        super(M, self).__init__()
        self.fc = torch.nn.Linear(4, 4)

    def forward(self, x):
        x = self.fc(x)
        return x

# create a model instance
model_fp32 = M()
# create a quantized model instance
model_int8 = torch.quantization.quantize_dynamic(
    model_fp32,  # the original model
    {torch.nn.Linear},  # a set of layers to dynamically quantize
    dtype=torch.qint8)  # the target dtype for quantized weights

# run the model
input_fp32 = torch.randn(4, 4, 4, 4)
res = model_int8(input_fp32)
```

```python
import torch

# define a floating point model where some layers could benefit from
QAT
class M(torch.nn.Module):
    def __init__(self):
        super(M, self).__init__()
        # QuantStub converts tensors from floating point to quantized
        self.quant = torch.quantization.QuantStub()
        self.conv = torch.nn.Conv2d(1, 1, 1)
        self.bn = torch.nn.BatchNorm2d(1)
        self.relu = torch.nn.ReLU()
        # DeQuantStub converts tensors from quantized to floating
point
        self.dequant = torch.quantization.DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.dequant(x)
        return x
```

Eager Mode Quantization

Quantization Aware Training

```
# create a model instance
model_fp32 = M()

# model must be set to train mode for QAT logic to work
model_fp32.train()

# attach a global qconfig, which contains information about what kind
# of observers to attach. Use 'fbgemm' for server inference and
# 'qnnpack' for mobile inference. Other quantization configurations
such
# as selecting symmetric or assymetric quantization and MinMax or
L2Norm
# calibration techniques can be specified here.
model_fp32.qconfig =
torch.quantization.get_default_qat_qconfig('fbgemm')

# fuse the activations to preceding layers, where applicable
# this needs to be done manually depending on the model architecture
model_fp32_fused = torch.quantization.fuse_modules(model_fp32,
    [['conv', 'bn', 'relu']])
```

```
# Prepare the model for QAT. This inserts observers and fake_quants
in
# the model that will observe weight and activation tensors during
calibration.
model_fp32_prepared =
torch.quantization.prepare_qat(model_fp32_fused)

# run the training loop (not shown)
training_loop(model_fp32_prepared)

# Convert the observed model to a quantized model. This does several
things:
# quantizes the weights, computes and stores the scale and bias value
to be
# used with each activation tensor, fuses modules where appropriate,
# and replaces key operators with quantized implementations.
model_fp32_prepared.eval()
model_int8 = torch.quantization.convert(model_fp32_prepared)

# run the model, relevant calculations will happen in int8
res = model_int8(input_fp32)
```

Common Errors

When calling `torch.load` on a quantized model, if you see an error like:

```
AttributeError: 'LinearPackedParams' object has no attribute
'_modules'
```

This is because directly saving and loading a quantized model using `torch.save` and `torch.load` is not supported. To save/load quantized models, the following ways can be used:

Passing a non-quantized Tensor into a quantized kernel

If you see an error similar to:

```
RuntimeError: Could not run 'quantized::some_operator' with arguments
from the 'CPU' backend...
```

Passing a quantized Tensor into a non-quantized kernel

If you see an error similar to:

```
RuntimeError: Could not run 'aten::thnn_conv2d_forward' with
arguments from the 'QuantizedCPU' backend.
```