TorchQuant: A Hackable Quantization Library for Researchers, by Researchers

Shyam A. Tailor[1,*] Milad Alizadeh[2,*] Nicholas D. Lane[1,3]

On-Device Intelligence Workshop, MLSys 2021

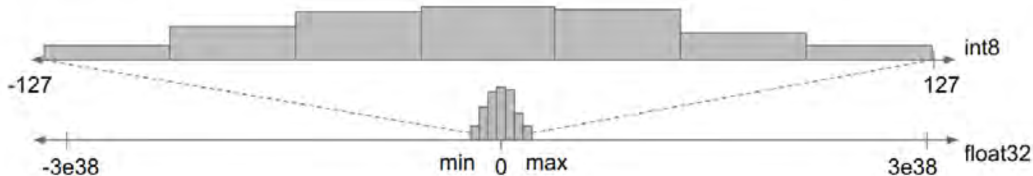
[*]Equal Contribution

[1]Department of Computer Science and Technology, University of Cambridge

[2]Department of Computer Science, University of Oxford

[3]Samsung AI Center, Cambridge (UK)

What is Quantization?



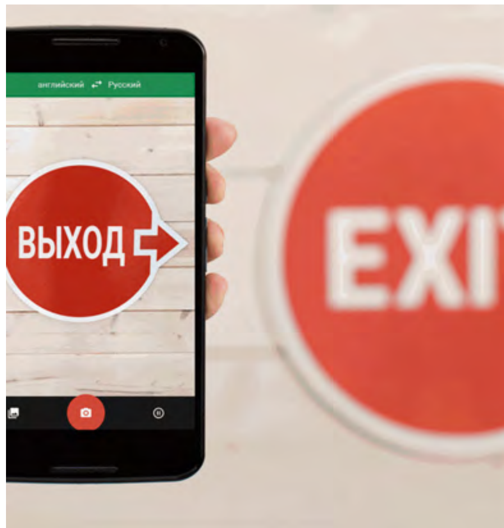
- **Main Idea:** Use fewer bits to represent our numbers when performing inference.
- Want to use the fewest number of bits possible without losing accuracy!

Image source: TensorFlow Blog

What is Quantization Useful For?

- Smaller Models
- Less Data Movement → Lower Power
- Lower Latency
- Simpler Accelerators

This allows us to put models in places that would otherwise be impossible!



How Do Researchers Implement Quantization?

We tend to simulate quantization, but beyond that **there is plenty of diversity**

- Different Techniques
- Different Models
- Quantizing during or after training?

The Problem: Researchers don't find the toolkits included in major libraries flexible enough, so they build everything themselves.

Duplicated work, and **more opportunities for bugs**!

What this Library Gives Us

1. A set of ***well tested*** core components that we can share and re-use.
2. Higher level interfaces that implement a ***state machine*** to ensure that our code is doing what we think it is doing.
3. ***Automated model conversion*** functionality for models that are commonly used as benchmarks in the literature.

Unit tested code and `mypy` type annotations.

Well Tested Low-Level Components

- Functionality like quantizer implementations and gradient clipping.
- Functional (`torch.nn.functional`) and class-based (`torch.nn.Module`) implementations.

```
class Quantizer(nn.Module, abc.ABC):
    @abc.abstractmethod
    def pre_observe(self, x: torch.Tensor):
        pass

    @abc.abstractmethod
    def quantize(self, x: torch.Tensor)
        -> torch.Tensor:
        pass

    @abc.abstractmethod
    def post_observe(self, x: torch.Tensor):
        pass
```

An Explicit State Machine To Prevent Bugs

`model.train()` and `model.eval()` is not enough!

- **We need finer control**: sometimes we only want to turn on some quantizers, or just feed some batches for calibration.
- Opt into our `QModule` construct and you can quickly change model settings:

```
with qmodule_state(module, QModuleState.CALIBRATION_WEIGHT_ONLY):
    ...
```

Automated Model Conversion

- Automatically convert full precision models to fused models–including preserving weights.
- ResNet, MobileNetV2 and EfficientNet.

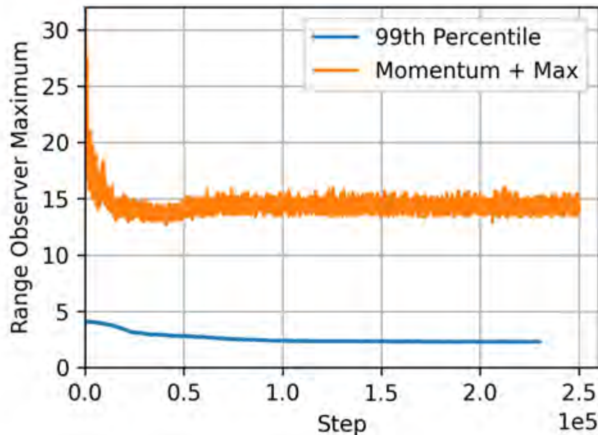
```
fused_model = FusedResNet(
    full_precision_model,
    weight_quantizer=lambda module: QuantizerForModuleWeights(module),
    acts_quantizer=lambda module: QuantizerForModuleActs(module)
)
```

What's Next?

1. More core components: BNNs, etc
2. Flexible model conversion
3. What about language or speech models?
4. Popular research methods in one place: faster progress for the community!

Checking a Hypothesis with Our Library: The "Range-Precision Tradeoff"

- At low bit-widths we have to choose between range and precision.
- Only a $\sim$**15 LOC** with our library to implement percentile range observers.

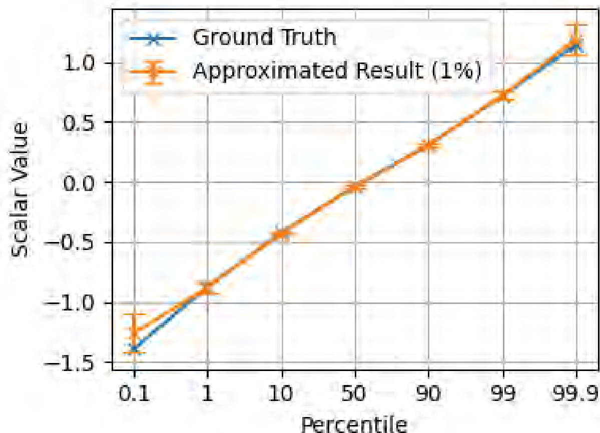


Should You Trade Range for Precision?

- ***Yes, absolutely***: 15-20%+ improvements at low bit-widths (2-4 bits).
- Beneficial on both weights and activations.
- Can easily fit into the range observer quantizers used by TensorFlow and PyTorch.

Aren't Percentiles Slow?

- Not as much as you might think!
- 27 minutes / epoch for baseline, 34 minutes for percentiles (**25% slower**).



Any Questions?

https://github.com/camlsys/torchquant