TORCHQUANT: A HACKABLE QUANTIZATION LIBRARY FOR RESEARCHERS, BY RESEACHERS

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

ABSTRACT

Quantization is a popular technique for accelerating and compressing neural networks by utilizing low-bit arithmetic to represent weights and activations. It remains a hot area for research, with continued work on removing the gap in accuracy between full and low precision models. We observe that researchers in this area tend to rely on custom implementations, rather than approaches built into the popular machine learning libraries, as they are not sufficiently flexible to enable research. We are open sourcing *TorchQuant*, our MIT licensed library that builds upon PyTorch by providing researchers with modular components and implementations that will accelerate their research, and provide the community with consistent baselines. Using our library, we provide an example of how to quickly evaluate a research hypothesis: the "range-precision" trade-off for quantization-aware training. our library can be found at this URL: https://github.com/camlsys/torchquant.

1 INTRODUCTION

It is traditional to use 32-bit floating point for training and inference of neural networks. In recent years, however, there has been interest in using fewer bits for these use-cases. The benefits of using fewer bits include: (1) reduced data movement, the biggest driver of energy consumption (Horowitz, 2014); (2) reduced storage requirements for weights (*model compression*) and runtime memory consumption of activations; (3) favourable characteristics for accelerator design e.g. simpler and more efficient processing elements, and (4) reduced latency in situations where the accelerator is memory-bound.

Quantization is one of the keys to enabling us to deploy deep learning to applications that were previously considered impossible. We are already witnessing these deep learning being deployed to smartphones for speech recognition and computer vision applications, aided by advances in hardware-software co-design. By reducing the resource footprint of our models further through advances in quantization, we can hope to apply deep learning in application domains such as wearable devices (Tong et al., 2020) or space satellites (Kothari et al., 2020).

For inference, the use of 8-bit integer (fixed-point) arithmetic has become commonplace, in part due to it resulting in minimal accuracy losses, and widespread hardware support. Support for quantization-aware training or post-training quantization for 8-bit arithmetic is now provided in the popular ML libraries (Abadi et al., 2015; Paszke et al., 2019), with TensorFlow adding support for flexible bitwidths in recent releases. These libraries are focused on *productionization*, rather than research: therefore, the approaches employed in these libraries are relatively primitive compared to the state-of-the-art in research. The research community continues to march towards reducing the gap in accuracy between models at very low (as far as 1-bit), and full precision, with recent approaches enabling minimal degradation on ImageNet when using 4-bit arithmetic (Esser et al., 2020). We observe that research works do not rely on in-built implementations provided by these ML libraries, and instead re-implement experiments ad-hoc for each paper, since researchers desire more flexibility than these libraries accommodate.

To support the research community, we are open-sourcing our MIT licensed library, TorchQuant, which provides reusable components and models for the wider community by building upon PyTorch, which has become the dominant library for ML research. We hope our work can provide a firm foundation for the research community by enabling greater reproducibility of published works, and faster iteration by eliminating duplicated work.

[*]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. Correspondence to: Shyam A. Tailor <sat62@cam.ac.uk>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

2 Why Do We Need a New Quantization Library?

Popular deep learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2015) have started supporting quantization operations. There are also quantization-specific libraries such as Larq (Geiger et al., 2020) and AIMET. However, these packages often aim to hide complexities of quantization and expose off-the-shelf tested techniques that allows practitioners to train a low-precision network from scratch, or compress a prerained network. In contrast, our library focuses on researches working on developing new quantization schemes. In addition to providing well-tested and maintained implementations of commonly used quantization techniques which enables rapid prototyping, our library provides lower-level hooks, and more complicated state machines that allows researchers to fearlessly test new quantization schemes.

The modular design of our library allows easy overriding of existing behaviour by inheriting or defining new quantizers, activation range observers, and fused layers. The state machine allows explicitly placing quantization of layers and tensors into training, calibration, evaluation mode.

The built-in quantization APIs in deep learning frameworks (partially) attempt to reap benefits of quantization during training and inference by actually using quantized tensors and operations. When doing research on quantization we are often interested in improving the achievable performance of the model and leave the deployment gains made possible by quantization to a later stage. For this reason all quantization operations in our library are full-precision under the hood. This means that unlike PyTorch's support for quantization that requires implementation of quantized versions of many operations, our modules run natively on the GPU like ordinary networks.

The range of support for various types of quantization is relatively limited in these libraries. While affine (fixed-point) quantization is well supported, there is little support for schemes such as binarized arithmetic, where each weight and activation is represented using a single bit, enabling implementation with only cheap XOR operations (Courbariaux et al., 2016). We need to provide researchers with the opportunity to quickly experiment with other quantization schemes in order to enable more rapid progress to be made. Ideally, these implementations should be well tested to ensure that results reported by the community are valid.

3 Our Library: TorchQuant

Our code employs professional software development practices. TorchQuant provides full type annotations, aiding user understanding, and enabling sophisticated code completion functionality; we also provide documentation and examples. Additionally, we rigorously test functionality, and use continuous integration to avoid regressions.

3.1 Overview of Supplied Functionality

Our library consists of three levels of functionality that build on each other. Users are free to opt-in at the level of functionality that suits their project: they are free to use just our quantizer implementations, or use more higher level functionality which enables entire models to be converted automatically for research purposes.

Quantizers. Following PyTorch's convention, we split our quantizer implementation into stateless *functionals*, and stateful *modules*. Functional code implements different quantization formats, and other common functionality such as gradient clipping. Modules are implemented to cover common use-cases, and transparently manage the state around these functionals. For example, we have implemented range-observing quantizers using this approach. Quantizers must implement three methods: (1) `pre_observe`, where the full-precision tensor is supplied as an argument prior to quantization; (2) `quantize`, which applies the quantization method; and (3) `post_observe` where the quantized tensor is supplied as an argument; this method can be used to implement approaches such as kurtosis-regularized quantization (Shkolnik et al., 2020), where a loss function is applied to the distribution of the quantized tensor's elements. We are careful to ensure that all our implementations have full unit-tests to ensure that the results provided by them are correct. This provides downstream users with the confidence that their results are correct, and are not an artefact of a subtly incorrect implementation. All quantizer implementations can be freely imported into other projects: users do not need to opt-in to higher-level functionality our library also provides, which we will elaborate upon next.

State Machines and Module Wrapping. As we identified in the introduction, a common source of frustration and bugs is imprecise control over quantizer state. To address this issue, we introduce an explicit state machine that enables users to specify behaviour: in a single line of code, users can disable quantization completely, enable calibration, enable quantization-aware training, or enable quantization for evaluation. This can be also applied at the level of individual modules. To opt-in to this functionality, users must use our `QModule` interface. There are strong incentives to do this: we provide functionality to "wrap" sequences of full-precision modules, while correctly implementing the state machine. Users of our library can therefore spend their time focusing on the implementation of their quantization technique, rather than spending time ensuring that they have quantized their model correctly.

Model Converters. Quantization works usually benchmark

upon the same set of models. It is typical to see models such as ResNets (He et al., 2015), MobileNets (Howard et al., 2017; Sandler et al., 2019), and EfficientNets (Tan & Le, 2020) being used to contextualize the performance of the proposed quantization method. As we have identified as a common theme, each work tends to re-implement the quantized version of each model: this results in work being duplicated, and subtle differences in implementation that may affect results. To address this, we provide a common set of model converters that enable quantized versions of these models to be built. We support the three families of aforementioned models: users can take advantage of our model converters by supplying the full precision model, along with factory functions that generate quantizers for the weights and activations in the model. Importantly, we preserve the weights of the full precision model: this greatly accelerates convergence, and improves the quantized model accuracy, especially at low bit-widths.

3.2 Future Roadmap

Our initial release is immediately useful to the community, but there are several features that we believe would be useful to add in the near future.

Expanding Supported Models. At time of writing, we support 3 major families of image classification models. In the future, we hope to support a wider variety of models, and remove our reliance on hard-coded model converters. The graph rewriting functionality recently added to PyTorch (`torch.fx`) is the ideal approach for this problem. Additionally, while the quantization literature is most focused on image models, it is also important to add support for other domains. We would like our library to seamlessly integrate with popular language and speech toolkits such as HuggingFace (Wolf et al., 2020) and SpeechBrain (Ravanelli et al., 2021). It would also be beneficial to integrate with federated learning libraries such as Flower (Beutel et al., 2020).

Methods "Zoo". It is in the community interest that proposed methods are reproducible. We hope to expand the scope of our work to hosting tested implementations of state-of-the-art methods, built on top of our library. This would enable researchers to quickly iterate, and reduce the overhead in open-sourcing their contributions.

4 Example: Evaluating the Range-Precision Trade-Off

We now provide an example of how our library can be used to quickly enable new research ideas to be evaluated. In this section, we expand upon an observation presented in Tailor et al. (2021), where it was observed that using percentile-based range-observers could be used to significantly improve quantized model accuracy. In summary, this work argues that for graph neural networks that it is better to trade *range* for *precision* when using quantization-aware training, as using traditional min/max-based observers results in distorted ranges due to infrequent outliers. This section evaluates how this observation applies to image models.

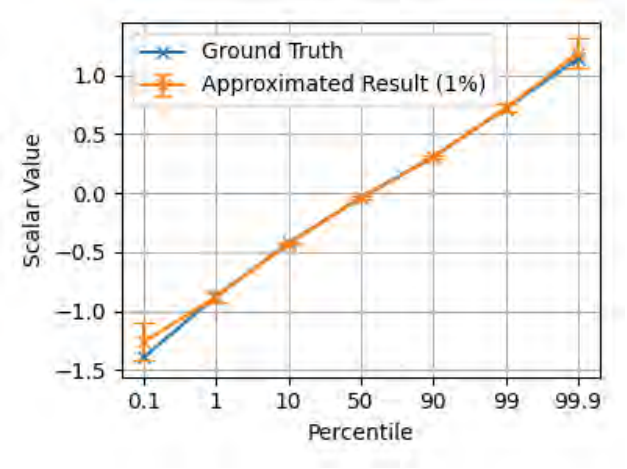

Figure 1. Accuracy of percentile approximations by sub-sampling across 10 runs. Applied to an intermediate activation tensor from a ResNet18. We observe that using 1% of the tensor to calculate the percentile is more than sufficient to arrive at a good approximation for the 1st and 99th percentile, which we are interested in. Error bars generated are standard deviation across 10 runs.

4.1 Implementation

Since TorchQuant already implements quantization using range observers, we only needed to implement a single class: a new type of range observer. We provide the abstract class `RangeObserver` that requires the implementation of a single method: `observe_batch`. Once this method is defined, we can simply pass our range observer class to our `AffineQuantizer` class, which handles the state management required for using fixed-point integer arithmetic quantizers. We can quickly evaluate this quantizer across a variety of models by using our included model converters.

We note that calculating percentiles is an expensive operation; we accelerate it by sub-sampling 1% of values from tensors with more than 100k elements. The accuracy of this approximation is justified in fig. 1. In our experiments we set the range minimum to the 1st percentile, and maximum to the 99th percentile, and used momentum to interpolate across batches.

4.2 Results

We evaluated the range-precision trade-off at very low bit-widths. Since it is well known that MobileNet and Effi-

Table 1. Test accuracies with different range observers on ImageNet. We use the abbreviation PAct to refer to percentiles being applied to activations only, and PAll to refer to percentiles also being applied to weights. The baseline uses a batch-min/max observer on weights, and momentum-min/max on activations.

Model	Bits	Range Observer	Top 1 %	Top 5 %
ResNet18	2	Baseline	-	-
ResNet18	2	PAct	12.6	30.2
ResNet18	2	PAll	35.5	60.8
MobileNetV2	4	Baseline	54.5	78.6
MobileNetV2	4	PAct	63.1	84.8
MobileNetV2	4	PAll	66.5	87.2
EfficientNet-B0	4	Baseline	27.8	52.3
EfficientNet-B0	4	PAct	54.4	78.6
EfficientNet-B0	4	PAll	66.5	87.6

cientNet suffer from wide variance in magnitudes across channels (Nagel et al., 2019), we evaluate them at 4-bits. ResNets, by comparison, are more tolerant to quantization, and we evaluate them at 2-bits to enable the benefit to be easier to observe.

We used a batch size of 256 and a step learning rate schedule that decreases by 10× every 30 epochs from an initial value of 0.01. We found it was vital to use gradient clipping with percentiles, unlike the baseline: we believe that since there are a relatively large number of values that are clipped in training, it is necessary to remove their contribution in order to preserve gradient accuracy. Our results are presented in Table 1. We find that applying percentiles is beneficial for both the weights and activations across all three models evaluated.

The benefits of using percentiles can be clearly observed in fig. 2. While the baseline model settles at a maximum range ∼15, the percentile model settles nearer to 2.5, with far less noise. Despite increasing truncation error by clipping more aggressively, the reduction in rounding error is sufficient to increase accuracy, as observed in Table 1.

The runtime overhead of using percentiles is not insurmountable. We ran each ResNet18 experiment on 2 Nvidia RTX 2080Tis, and obtained a time-per-epoch of 27 minutes for the baseline. When using percentiles for both weights and activations, the time-per-epoch was 34 minutes. While this is a noticeable overhead, it is certainly worth it in the context of better convergence properties. We also note that PyTorch's percentile function fully sorts the tensor, which is not strictly required for percentile calculation, and hence further speedups may be achievable. TorchQuant's flexibility has enabled us to quickly evaluate a research hypothesis, with the only work required being to implement a new range observer.

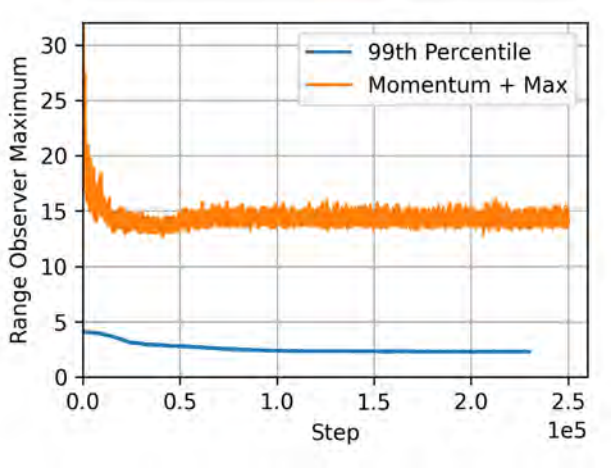

Figure 2. Range observer maximum for an intermediate activation tensor from an EfficientNet-B0 model immediately after a projection convolution. We see that the momentum + min/max run settles at a far higher value (near 15), and has greater noise, compared to the percentile run, where the value settles at 2.5. This enables our tensor to be represented in quantized form with greater precision, at the cost of truncation error.

5 Conclusion

We have described our quantization library, TorchQuant, that is explicitly designed to give researchers the freedom to implement novel quantization algorithms. Our library is highly modular, with users being able to opt-in to higher-level functionality. We have provided well tested implementations of quantization primitives that can be easily integrated into other libraries, along with higher level functionality to aid researchers in writing bug-free code. In this work we have demonstrated how we can use it to quickly evaluate research hypotheses by assessing the range-precision trade-off for quantization-aware training; we have found that trading precision for truncation error is worthwhile at low bitwidths. This observation may be useful for major library implementers due to the simplicity of proposed technique.

Acknowledgments

This work was supported by Samsung AI and by the UK's Engineering and Physical Sciences Research Council (EPSRC) with grants EP/M50659X/1 and EP/S001530/1 (the MOA project) and the European Research Council via the REDIAL project. We also thank ARM for their generous support.

REFERENCES

Abadi, M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Parcollet, T., and Lane, N. D. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. Learned step size quantization. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgO66VKDS.

Geiger, L. et al. Larq: An open-source library for training binarized neural networks. *Journal of Open Source Software*, 5(45):1746, January 2020. doi: 10.21105/joss.01746. URL https://doi.org/10.21105/joss.01746.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

Horowitz, M. Computing's energy problem (and what we can do about it). volume 57, pp. 10–14, 02 2014. ISBN 978-1-4799-0920-9. doi: 10.1109/ISSCC.2014.6757323.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

Kothari, V., Liberis, E., and Lane, N. D. The final frontier: Deep learning in space. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, pp. 45–49, 2020.

Nagel, M., van Baalen, M., Blankevoort, T., and Welling, M. Data-free quantization through weight equalization and bias correction, 2019.

Paszke, A. et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.

Ravanelli, M., Parcollet, T., Rouhe, A., Plantinga, P., Rastorgueva, E., Lugosch, L., Dawalatabad, N., Ju-Chieh, C., Heba, A., Grondin, F., Aris, W., Liao, C.-F., Cornell, S., Yeh, S.-L., Na, H., Gao, Y., Fu, S.-W., Subakan, C., De Mori, R., and Bengio, Y. Speechbrain. https://github.com/speechbrain/speechbrain, 2021.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

Shkolnik, M., Chmiel, B., Banner, R., Shomron, G., Nahshan, Y., Bronstein, A., and Weiser, U. Robust quantization: One model to rule them all, 2020.

Tailor, S. A., Fernandez-Marques, J., and Lane, N. D. Degree-quant: Quantization-aware training for graph neural networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=NSBrFgJAHg.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

Tong, C., Tailor, S. A., and Lane, N. D. Are accelerometers for activity recognition a dead-end? In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, pp. 39–44, 2020.

Wolf, T. et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.