ActorQ: Quantization for Actor-Learner Distributed Reinforcement Learning

Maximilian Lam[*]
Harvard University

Sharad Chitlangia[*†]
BITS Pilani Goa

Srivatsan Krishnan[*]
Harvard University

Zishen Wan[‡]
Harvard University

Gabriel Barth-Maron
Deepmind

Aleksandra Faust
Robotics at Google

Vijay Janapa Reddi
Harvard University

Abstract

In this paper, we introduce a novel Reinforcement Learning (RL) training paradigm, *ActorQ*, for speeding up actor-learner distributed RL training. *ActorQ* leverages full precision optimization on the learner, and distributed data collection through lower-precision quantized actors. The quantized, 8-bit (or 16 bit) inference on actors, speeds up data collection without affecting the convergence. The quantized distributed RL training system, *ActorQ*, demonstrates end to end speedups of $> 1.5 \times$ - $2.5 \times$, and faster convergence over full precision training on a range of tasks (Deepmind Control Suite) and different RL algorithms (D4PG, DQN). Finally, we break down the various runtime costs of distributed RL training (such as communication time, inference time, model load time, etc) and evaluate the effects of quantization on these system attributes.

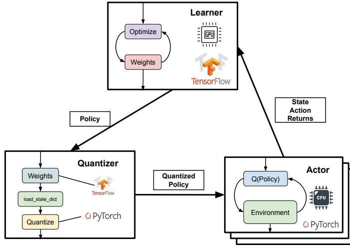
1 Introduction

Deep reinforcement learning has attained significant achievements in various fields and has demonstrated considerable potential in areas spanning from robotics Chiang et al. (2019); OpenAI et al. (2019) to game playing Bellemare et al. (2012); Silver et al. (2016); OpenAI (2018). Despite its promise, the computational burdens of training and deploying reinforcement learning policies remain a significant issue. Training reinforcement learning models is fundamentally resource intensive due to the computationally expensive nature of deep neural networks and the sample inefficiency of reinforcement learning training Buckman et al. (2018).

Neural network quantization and improving the system performance of reinforcement learning have both been the subject of much research; however, research cutting across these two domains has been largely absent. Neural network quantization has been successfully applied to various supervised learning applications such as image recognition Hubara et al. (2016); Tambe et al. (2020); Rusci et al. (2020), natural language processing Shen et al. (2020); Zafrir et al. (2019) and speech recognition Shangguan et al. (2019); Tambe et al. (2019) but has yet to be applied in the context of reinforcement learning. Reinforcement learning has made efforts to develop more efficient learning algorithms and also more computationally efficient systems to speed training and inference, but these efforts have mainly focused either on sample efficiency Buckman et al. (2018); Mnih et al. (2016a) or on parallelization/hardware acceleration Babaeizadeh et al. (2016); Espeholt et al. (2018); Petrenko et al. (2020). To the best of our knowledge, the impact of quantization on various aspects of reinforcement learning (e.g: training, deployment, etc) remains unexplored.

Applying quantization to reinforcement learning is nontrivial and different from traditional neural network. In the context of policy inference, it may seem that, due to the sequential decision making nature of reinforcement learning, errors made at one state might propagate to subsequent states, suggesting that policies might be more challenging to quantize than traditional neural network applications. In the context of reinforcement learning training, quantization seems difficult to apply due to the myriad of different algorithms Mnih et al. (2016b); Barth-Maron et al. (2018) and the

[*]Equal contribution.
[†]This work was done while Sharad was a visiting student at Harvard.
[‡]Now at Georgia Tech.



(a) *ActorQ* system setup. We leverage Tensorflow on the learner process and PyTorch on the actor processes to facilitate full precision GPU inference for optimization and quantized inference for experience generation. We introduce a parameter quantizer to bridge the gap between the Tensorflow model from the learner and the quantized PyTorch models on the actors.

Characteristic	Description
Learners on GPUs Actors on CPUs	Learners perform batched optimization on GPUs Multiple parallel actors perform inference to generate data
Tensorflow on Learner Pytorch on Actors	Pytorch's Quantized Inference allows to speedup actor inference on hence data generation
Separate Parameter Quantizer Process	A separate parameter quantizer process helps in not burdening the learner with the conversion processes On a high level, the learner sends the full precision weights to the parameter quantizer, converts it to a pytorch model for efficient inference and broadcasts it to all actors
Quantize Compute v/s Quantize Communication	Actors can be quantized to perform 8-bit or 16-bit and generate data faster Communication can be quantized to any number of bits.
Asynchronous Data Pushes on Learner Side Synchronous Data Pulls on Actor Side	Asynchronous Pushes maximizes learner resource usage Synchronous Pulls on Actors to avoid stale models in replay buffer and to avoid thrashing
Push Model Dict instead of serialized object	Send Serialized Pytorch Model Dict as it is the most compact representation

complexity of these optimization procedures. On the former point, our insight is that reinforcement learning policies are resilient to quantization error as policies are often trained with noise Igl et al. (2019) for exploration, making them robust. And on the latter point, we leverage the fact that reinforcement learning procedures may be framed through the actor-learner training paradigm Horgan et al. (2018), and rather than quantizing learner optimization, we may achieve speedups while maintaining convergence by quantizing just the actors' experience generation.

In summary, our fundamental contributions are as follows:

- We introduce a simple but effective reinforcement learning training quantization technique: *ActorQ* to speed up distributed reinforcement learning training. *ActorQ* operates by having the learner perform full precision optimization and the actors perform quantized inference. *ActorQ* achieves between $1.5\times$ and $2.5\times$ speedup on a variety of tasks from the Deepmind control suite Tassa et al. (2018)
- We develop a system around *ActorQ* that leverages both Tensorflow and PyTorch to perform quantized distributed reinforcement learning and demonstrate significant speedups on a range of tasks. We furthermore discuss the design of the distributed system and identify computation and communication as key runtime overheads in distributed reinforcement learning training

2 ActorQ

Our distributed reinforcement learning system follows the standard actor-learner approach: a single learner optimizes the policy while multiple actors perform rollouts in parallel. As the learner performs computationally intensive operations (batched updates to both actor and critic), it is assigned faster compute (in our case a GPU). Actors, on the other hand, perform individual rollouts which involves executing inference one example at a time, which suffers from limited parallelizability; hence they are assigned individual CPU cores and run independently of each other. The learner holds a master copy of the policy and periodically broadcasts the model to all actors. The actors pull the model and use it to perform rollouts, submitting examples to the replay buffer which the learner samples to optimize the policy and critic networks. A diagram showing the actor-learner setup is shown in Figure 1a.

We introduce *ActorQ* for quantized actor-learner training. *ActorQ* involves maintaining all learner computation in full precision while using quantized execution on the actors. When the learner broadcasts its model, post training quantization is performed on the model and the actors utilize the quantized model in their rollouts. In experiments, we measure the quality of the full precision policy from the learner. Several motivations for *ActorQ* include:

- The learner is faster than all the actors combined due to hardware and batching; hence overall training speed is limited by how fast actors can perform rollouts.

- Post training quantization is effective in producing a quantized reinforcement learning policy with little loss in reward. This indicates that quantization (down to 8 bits) has limited impact on the output of a policy and hence can be used to speed up actor rollouts.
- Actors perform only inference (no optimization) so all computation on the actor's side may be quantized significantly without harming optimization; conversely, the learner performs complex optimization procedures on both actor and critic networks and hence quantization on the learner would likely degrade convergence.

While simple, *ActorQ* distinguishes from traditional quantized neural network training as the inference-only role of actors enables the use of very low precision ($\leq$ 8 bit) operators to speed up training. This is unlike traditional quantized neural network training which must utilize more complex algorithms like loss scaling Das et al. (2018), specialized numerical representations Sun et al. (2019); Wang et al. (2018), stochastic rounding Wang et al. (2018) to attain convergence. This adds extra complexity and may also limit speedup and, in many cases, are still limited to half precision operations due to convergence issues.

The benefits of *ActorQ* are twofold: not only is the computation on the actors sped up, but communication between learner and actors are also significantly reduced. Additionally, post training quantization in this process can be seen as injecting noise into actor rollouts, which is known to improve exploration Louizos et al. (2018); Bishop (1995); Hirose et al. (2018), and we show that, in some cases, this may even benefit convergence. Finally, *ActorQ* applies to many different reinforcement learning algorithms as the actor-learner paradigm is general across various algorithms.

3 RESULTS

We apply PTQ in the context of distributed reinforcement learning training through *ActorQ* and demonstrate significant end to end training speedups without harming convergence. We evaluate the impact of quantizing communication versus computation in distributed reinforcement learning training and break down the runtime costs in training to understand how quantization affects these systems components.

Task	Reward Achieved	FP32 Time to Reward (s)	Fp16 Time to Reward (s)	Int8 Time to Reward (s)	Fp16 Speedup	Int8 Speedup
Cartpole Balance	941.22	870.91	284.65	279.00	3.06	3.12
Walker Stand	947.74	871.32	625.00	534.37	1.39	1.63
Hopper Stand	836.41	2660.41	2406.14	1699.17	1.11	1.57
Reacher Hard	948.12	1597.00	1122.24	875.34	1.42	1.82
Cheetah Run	732.31	2517.30	3012.00	891.84	0.84	2.82
Finger Spin	810.32	3256.56	1527.76	1065.52	2.13	3.06
Humanoid Stand	884.89	13964.92	14371.84	9302.82	0.97	1.51
Humanoid Walk	649.91	17990.66	19106.88	6223.35	0.94	2.89
Cartpole (Gym)	198.22	963.67	535.09	260.10	1.80	3.70
Mountain Car (Gym)	-120.62	2861.80	2159.26	1284.32	1.32	2.22
Acrobot (Gym)	-107.45	912.24	1148.90	168.44	0.79	5.41

Table 2: *ActorQ* time and speedups to 95% reward on select tasks from Deepmind Control Suite and Gym. 8 and 16 bit inference yield $> 1.5\times - 2.5\times$ speedup over full precision training. We use D4PG on DeepMind Control Suite environments (non-gym), DQN on gym environments.

We evaluate the *ActorQ* algorithm for speeding up distributed quantized reinforcement learning across various environments. Overall, we show that: 1) we see significant speedup (>1.5 $\times$-2.5 $\times$) in training reinforcement learning policies using *ActorQ* and 2) convergence is maintained even when actors perform down to 8 bit quantized execution. Finally, we break down the relative costs of the components of training to understand where the computational bottlenecks are. Note in *ActorQ* while actors perform quantized execution, the learner's models are full precision, hence we evaluate the learner's full precision model quality.

We evaluate *ActorQ* on a range of environments from the Deepmind Control Suite Tassa et al. (2018). We choose the environments to cover a wide range of difficulties to determine the effects of quantization on both easy and difficult tasks. Difficulty of the Deepmind Control Suite tasks are determined by Hoffman et al. (2020). Table 3 lists the environments we tested on with their corresponding difficulty and number of steps trained. Each episode has a maximum length of 1000 steps, so the maximum reward for each task is 1000 (though this may not always be attainable). We train on the features of the task rather than the pixels.

Policy architectures are fully connected networks with 3 hidden layers of size 2048. We apply a gaussian noise layer to the output of the policy network on the actor to encourage exploration; sigma is uniformly assigned between 0 and .2 according to the actor being executed. On the learner side, the critic network is a 3 layer hidden network with hidden size 512. We train policies using D4PG Barth-Maron et al. (2018) on continuous control environments and DQN Mnih et al. (2013) on discrete control environments. We chose D4PG as it was the best learning algorithm in Tassa et al. (2018); Hoffman et al. (2020), and DQN is a widely used and standard reinforcement learning algorithm. An example submitted by an actor is sampled 16 times before being removed from the replay buffer (spi=16) (lower spi is typically better as it minimizes model staleness Fedus et al. (2020)).

All experiments are run on a single machine setup (but distributed across the GPU and the multiple CPUs of the machine). A V100 GPU is used on the learner, while we use 4 actors (1 core for each actor) each assigned a Intel Xeon 2.20GHz CPU for distributed training. We run each experiment and average over at least 3 runs and compute the running mean (window=10) of the aggregated runs.

End to End Speedups

We show end to end training speedups with *ActorQ* in Figure 1 and table 2. Across nearly all tasks we see significant speedups with both 8 bit and 16 bit quantized inference. Additionally, to improve readability, we estimate the 95% percentile of the maximum attained score by fp32 and measure time to this reward level for fp32, fp16 and int8 and compute corresponding speedups. This is shown in Table 2. Note that Table 2 does not take into account cases where fp16 or int8 achieve a higher score than fp32.

On Humanoid, Stand and Humanoid, Walk convergence was significantly slower with a slower model pull frequency (1000) and so we used more frequent pulls (100) in their training. The frequent pulls slowed down 16 bit inference to the point it was as slow as full precision training. In the next sections we will identify what caused this.

Convergence

We show the episode reward versus total actor steps convergence plots using *ActorQ* in Figure 2. Data shows that broadly, convergence is maintained even with both 8 bit and 16 bit inference on actors across both easy and difficult tasks. On Cheetah, Run and Reacher, Hard, 8 bit *ActorQ* achieves even slightly faster convergence and we believe this may have happened as quantization introduces noise which could be seen as exploration.

Communication vs Computation

The frequency of model pulls on actors may have impacts on convergence as it affects the staleness of policies being used to populate the replay buffer; this has been witnessed in both prior research Fedus et al. (2020) and an example is also shown in Figure 3. Thus, we explore the effects of quantizating communication versus computation in both communication and computation heavy setups. To quantize communication, we quantize policy weights to 8 bits and compress by packing them into a matrix, reducing the memory of model broadcasts by $4\times$. Naturally, quantizating communication would be more beneficial in the communication heavy scenario and quantizing compute would yield relatively more gains in the computation heavy scenario. Figure 4 shows an ablation plot of the gains of quantization on both communication and computation in a communication heavy scenario (frequency=30) versus a computation heavy scenario (frequency=1000). Figures show that in a communication heavy scenario quantizing communication may yield up to 30% speedup; conversely, in a computation heavy scenario quantizing communication has little impact as the overhead is dominated by computation. Note that as our experiments were run on multiple cores of a single node (with 4 actors), communication is less of a bottleneck. We assume that communication would incur larger costs on a networked cluster with more actors.

Runtime Breakdown

We further break down the various components contributing to runtime on a single actor. Runtime components are broken down into: step time, pull time, deserialize time and load_state_dict time. Step time is the time spent performing neural network inference; pull time is the time between querying the Reverb queue for a model and receiving the serialized models weights; deserialize time is the time spent to deserialize the serialized model dictionary; load_state_dict time is the time to call PyTorch load_state_dict.

Figure 4c shows the relative breakdown of the component runtimes with 32, 16 and 8 bit quantized inference in the computation heavy scenario. As shown, step time is the main bottleneck and quantization significantly speeds this up. Figure 4d shows the cost breakdown in the communication heavy scenario. While speeding up computation, pull time and deserialize time are also significantly sped up by quantization due to reduction in memory.

In 8 bit and 16 bit quantized training, the cost of PyTorch load_state_dict is significantly higher. An investigation shows that the cost of loading a quantized PyTorch model is spent repacking the weights from Python object into C data. 8 bit weight repacking is noticeably faster than 16 bit weight repacking due to fewer memory accesses. The cost of model loading suggests that additional speed gains can be achieved by serializing the packed C data structure and reducing the cost of weight packing.

4 Conclusion

We evaluate quantization to speed up reinforcement learning training and inference. We show standard quantization methods can quantize policies down to ≤ 8 bits with little loss in quality. We develop *ActorQ*, and attain significant speedups over full precision training. Our results demonstrate that quantization has considerable potential in speeding up both reinforcement learning inference and training. Future work includes extending the results to networked clusters to evaluate further the impacts of communication and applying quantization to reinforcement learning to different application scenarios such as the edge.

References

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*, 2016.

Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2018.

Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. 2018.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012. URL `http://arxiv.org/abs/1207.4708`.

C. M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, Jan 1995. doi: 10.1162/neco.1995.7.1.108.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL `http://arxiv.org/abs/1606.01540`.

Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. In *Advances in Neural Information Processing Systems*, pp. 8224–8234, 2018.

Hao-Tien Lewis Chiang, Aleksandra Faust, Marek Fiser, and Anthony Francis. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4(2):2007–2014, April 2019. ISSN 2377-3766. doi: 10.1109/LRA.2019.2899918.

Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. 2018.

Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. Mixed precision training of convolutional neural networks using integer operations. *International Conference on Learning Representations (ICLR)*, 2018.

Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 293–302, 2019.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning (ICML)*, 2018.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. 2019.

William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning (ICML)*, 2020.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Kazutoshi Hirose, Ryota Uematsu, Kota Ando, Kodai Ueyoshi, Masayuki Ikebe, Tetsuya Asai, Masato Motomura, and Shinya Takamaeda-Yamazaki. Quantization error-based regularization for hardware-aware neural network training. *Nonlinear Theory and Its Applications, IEICE*, 9(4):453–465, 2018. doi: 10.1587/nolta.9.453.

Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2018.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016. URL `http://arxiv.org/abs/1609.07061`.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18 (187):1–30, 2018. URL `http://jmlr.org/papers/v18/16-456.html`.

Maximilian Igl, Kamil Ciosek, Yingzhen Li, Sebastian Tschiatschek, Cheng Zhang, Sam Devlin, and Katja Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. In *Advances in Neural Information Processing Systems*, pp. 13978–13990, 2019.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2018.

Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

Srivatsan Krishnan, Behzad Boroujerdian, William Fu, Aleksandra Faust, and Vijay Janapa Reddi. Air learning: An AI research platform for algorithm-hardware benchmarking of autonomous aerial robots. *CoRR*, abs/1906.00421, 2019. URL `http://arxiv.org/abs/1906.00421`.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies, 2015.

Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. *International Conference on Learning Representations (ICLR)*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016a.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016b. URL http://arxiv.org/abs/1602.01783.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 561–577, 2018.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2020.

Manuele Rusci, Alessandro Capotondi, and Luca Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. In I. Dhillon, D. Papailiopoulos, and V. Sze (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 326–335. 2020. URL https://proceedings.mlsys.org/paper/2020/file/9b8619251a19057cff70779273e95aa6-Paper.pdf.

Yuan Shangguan, Jian Li, Liang Qiao, Raziel Alvarez, and Ian McGraw. Optimizing speech recognition for the edge. *arXiv preprint arXiv:1909.12408*, 2019.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *AAAI*, pp. 8815–8821, 2020.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *Advances in Neural Information Processing Systems 32*. 2019.

Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Adaptivfloat: A floating-point based data type for resilient deep learning inference. *arXiv preprint arXiv:1909.13271*, 2019.

Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2020.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, 2018.

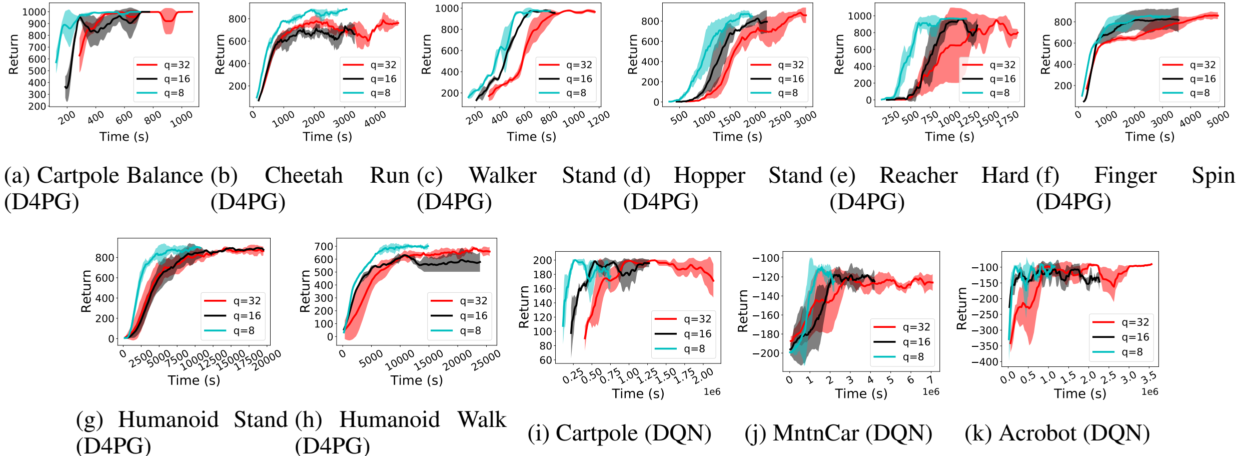
Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.

Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving Neural Network Quantization without Retraining using Outlier Channel Splitting. *International Conference on Machine Learning (ICML)*, pp. 7543–7552, June 2019.

APPENDIX

A ACTOR Q

A.1 ACTORQ SPEEDUPS

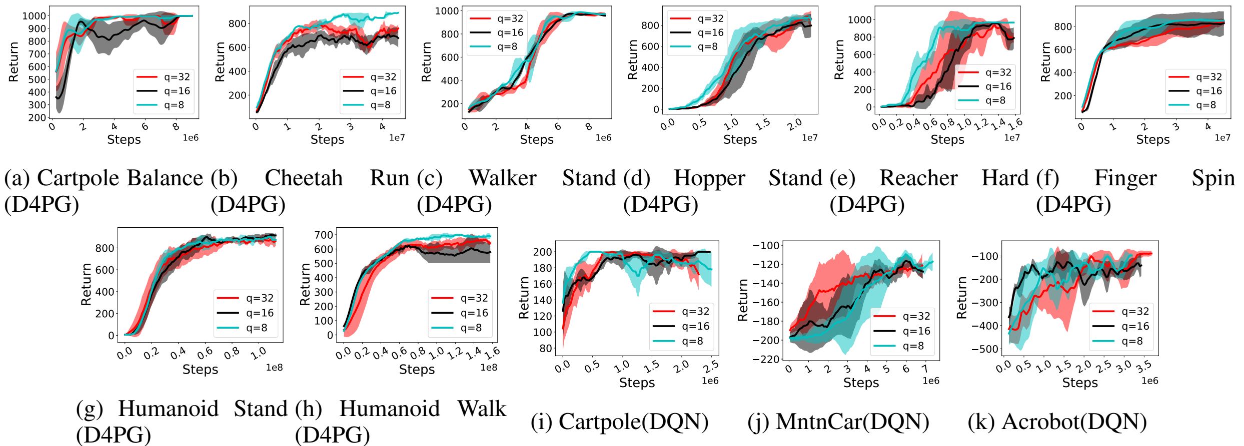


(a) Cartpole Balance (D4PG) (b) Cheetah Run (D4PG) (c) Walker Stand (D4PG) (d) Hopper Stand (D4PG) (e) Reacher Hard (D4PG) (f) Finger Spin (D4PG)

(g) Humanoid Stand (D4PG) (h) Humanoid Walk (D4PG) (i) Cartpole (DQN) (j) MntnCar (DQN) (k) Acrobot (DQN)

Figure 1: End to end speedups of *ActorQ* across various Deepmind Control Suite tasks using 8 bit, 16 bit and 32 bit inference on actors (full precision optimization on learner). 8 and 16 bit training yield significant end to end training speedups over the full precision baseline.

A.2 CONVERGENCE OF ACTORQ



(a) Cartpole Balance (D4PG) (b) Cheetah Run (D4PG) (c) Walker Stand (D4PG) (d) Hopper Stand (D4PG) (e) Reacher Hard (D4PG) (f) Finger Spin (D4PG)

(g) Humanoid Stand (D4PG) (h) Humanoid Walk (D4PG) (i) Cartpole(DQN) (j) MntnCar(DQN) (k) Acrobot(DQN)

Figure 2: Convergence of *ActorQ* across various Deepmind Control Suite tasks using 8 bit, 16 bit and 32 bit inference on actors (full precision optimization on learner). 8 bit and 16 bit quantized training attains the same or better convergence than full precision training.

A.3 Environment Details

Task	Algorithm	Difficulty	Steps Trained	Model Pull Freq (steps)
Cartpole Balance	D4PG	Trivial	40000	1000
Walker Stand	D4PG	Trivial	40000	1000
Hopper Stand	D4PG	Easy	100000	1000
Reacher Hard	D4PG	Easy	70000	1000
Cheetah Run	D4PG	Medium	200000	1000
Finger Spin	D4PG	Medium	200000	1000
Humanoid Stand	D4PG	Hard	500000	100
Humanoid Walk	D4PG	Hard	700000	100
Cartpole	DQN	N/A	60000	1000
Acrobot	DQN	N/A	100000	1000
MountainCar	DQN	N/A	200000	1000

Table 3: Tasks evaluated using *ActorQ* range from easy to difficult, along with the steps trained for corresponding tasks, with how frequently the model is pulled on the actor side.

A.4 ActorQ Runtime Breakdown

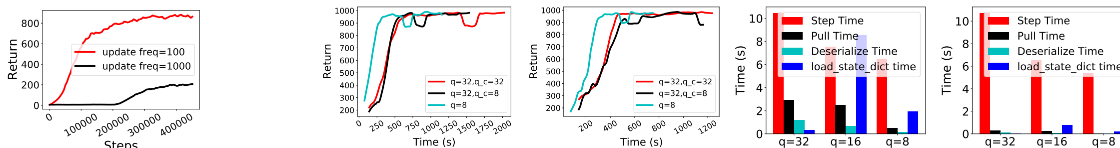


Figure 3: Humanoid stand: Model pull frequency affects staleness of actor policies and may have an effect on training.

(a) Communication Heavy (Update Freq=30) (b) Computation Heavy (Update Freq=300) (c) Communication Heavy (Update Freq=30) (d) Computation Heavy (Update Freq=300)

Figure 4: Effects of quantizing communication versus computation in compute heavy and communication heavy training scenarios. q is the precision of inference; q_c is the precision of communication. Note q=8 implicitly quantizes communication to 8 bits.

B Standard Quantization Results

B.1 Post Training Quantization

Here, we tabulate the post training quantization results

Algorithm →	**A2C**			**DQN**			**PPO**			**DDPG**		
Datatype →	**fp32**	**fp16**	**int8**	**fp32**	**fp16**	**int8**	**fp32**	**fp16**	**int8**	**fp32**	**fp16**	**int8**
Environment ↓	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**	**Rwd**
Breakout	379	371	350	214	217	78	400	400	368			
SpaceInvaders	717	667	634	586	625	509	698	662	684			
BeamRider	3087	3060	2793	925	823	721	1655	1820	1697			
MsPacman	1915	1915	2045	1433	1429	2024	1735	1735	1845			
Qbert	5002	5002	5611	641	641	616	15010	15010	14425			
Seaquest	782	756	753	1709	1885	1582	1782	1784	1795			
CartPole	500	500	500	500	500	500	500	500	500			
Pong	20	20	19	21	21	21	20	20	20			
Walker2D	399	422	442				2274	2273	2268	1890	1929	1866
HalfCheetah	2199	2215	2208				3026	3062	3080	2553	2551	2473
BipedalWalker	230	240	226				304	280	291	98	90	83
MountainCar	94	94	94				92	92	92	92	92	92

Table 4: Post training quantization error for DQN, DDPG, PPO, and A2C algorithm on Atari and Gym. Quantization down to 8 bits yields similar rewards to full precision baseline.

For all Atari games in the results section, we use a standard 3 Layer Conv (128) + 128 FC. Hyperparameters are listed in Table 5. We use stable-baselines (Hill et al., 2018) for all the reinforcement learning experiments. We use Tensorflow version 1.14 as the machine learning backend.

Hyperparameter	Value
n_timesteps	1 Million Steps
buffer_size	10000
learning_rate	0.0001
warm_up	10000
quant_delay	500000
target_network_update_frequency	1000
exploration_final_eps	0.01
exploration_fraction	0.1
prioritized_replay_alpha	0.6
prioritized_replay	True

Table 5: Hyper parameters used for mixed precision training for training DQN algorithm in all the Atari environments.

B.2 Quantization Aware Training

We present rewards for policies quantized via quantization aware training on multiple environments and training algorithms in Figure 5. Generally, the performance relative to the full precision baseline is maintained until 5/6-bit quantization, after which there is a drop in performance.

Broadly, at 8-bits, we see no degradation in performance. From the data, we see that quantization aware training achieves higher rewards than post-training quantization and also sometimes outperforms the full precision baseline.

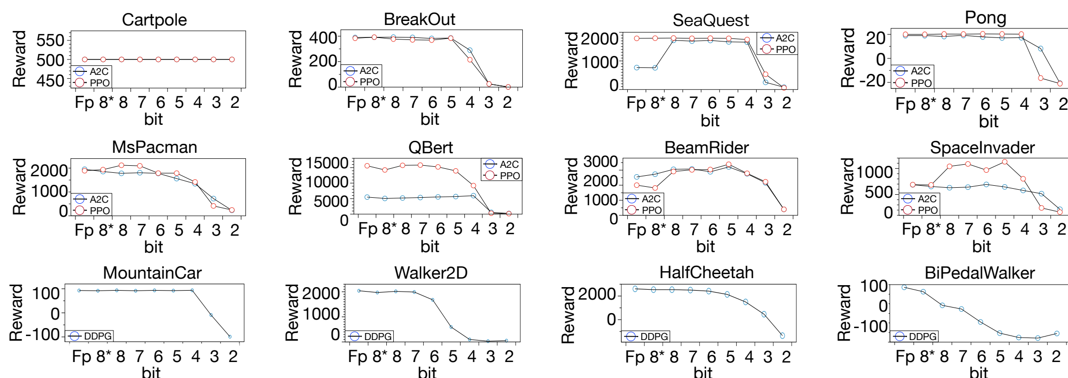


Figure 5: Quantization aware training of PPO, A2C, and DDPG algorithms on OpenAI gym, Atari, and PyBullet. FP is achieved by fp32 and 8* is achieved by 8-bit post-training quantization.

B.3 Enabling Deployment with Quantized Policy on Resource-Constrained Edge Device

To demonstrate the significant benefits of quantization in deploying reinforcement learning policies, we evaluate quantized policy benefits for a robotics use case. We train multiple point-to-point navigation models (Policy I, II, and III) for aerial robots using Air Learning (Krishnan et al., 2019) and deploy them onto a RasPi-3b, a cost effective, general-purpose embedded processor. RasPi-3b is used as proxy for the compute platform for the aerial robot. Other platforms on aerial robots have similar characteristics. For each of these policies, we report the accuracies and inference speedups attained by the int8 and fp32 policies.

Policy Name	Network Parameters	fp32 Time (ms)	fp32 Success Rate (%)	int8 Time (ms)	int8 Success Rate (%)	Speed up
Policy I	3L, MLP, 64 Nodes	0.147	60%	0.124	45%	1.18 ×
Policy II	3L, MLP, 256 Nodes	133.49	74%	9.53	60%	14 ×
Policy III	3L, MLP (4096, 512, 1024)	208.115	86%	11.036	75%	18.85 ×

Table 6: Inference speed on Ras-Pi3b+ for Air Learning policies.

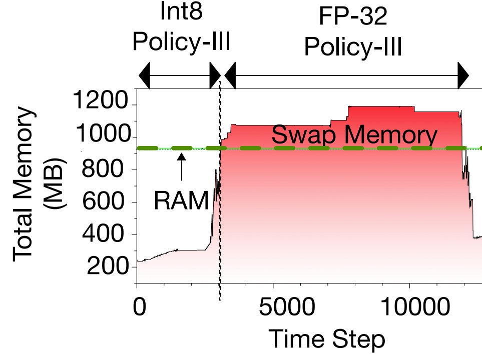


Figure 6: Figure shows the memory consumption for Policy III's fp-32 and int8 policies. Without quantization, memory accesses overflow to swap, causing large inference slowdowns.

We see that quantizing smaller policies (Policy I) yield moderate inference speedups (1.18× for Policy I), while quantizing larger models (Policies II, III) can speed up inference by up to 18×. Our results also demonstrate that a larger quantized model has significantly better runtime speed (Policy III, int8 at 11 ms) than the next comparable model (Policy II, int32 at 133 ms) at similar accuracies (Policy II, int32 at 74% success; Policy III, int8 at 75 % success).

A further analysis of the speedups shows that Policies II and III take more memory than the total RAM capacity of the RasPi-3b, causing numerous accesses to swap memory during inference. Quantizing these policies allows them to fit into the RasPi's RAM, eliminating accesses to swap and boosting performance by an order of magnitude.

Here, we describe the methodology used to train a point to point navigation policy in Air Learning and deploy it on an embedded compute platform such as Ras-Pi 3b+. Air Learning is an AI research platform that provides infrastructure components and tools to train a fully functional reinforcement learning policies for aerial robots. In simple environments like OpenAI gym, Atari the training and inference happens in the same environment without any randomization. In contrast to these environments, Air Learning allows us to randomize various environmental parameters such as such as arena size, number of obstacles, goal position etc.

In this study, we fix the arena size to 25 m × 25 m × 20 m. The maximum number of obstacles at anytime would be anywhere between one to five and is chosen randmonly on episode to episode basis. The position of these obstacles and end point (goal) are also changed every episode. We train the aerial robot to reach the end point using DQN algorithm. The input to the policy is sensor mounted on the drone along with IMU measurements. The output of the policy is one among the 25 actions with different velocity and yaw rates. The reward function we use in this study is defined based on the following equation:

$$r = 1000 * \alpha - 100 * \beta - D_g - D_c * \delta - 1 \quad (1)$$

Here, α is a binary variable whose value is '1' if the agent reaches the goal else its value is '0'. β is a binary variable which is set to '1' if the aerial robot collides with any obstacle or runs out of the maximum allocated steps for an episode.[1] Otherwise, β is '0', effectively penalizing the agent for hitting an obstacle or not reaching the end point in time. D_g is the distance to the end point from the agent's current location, motivating the agent to move closer to the

[1]We set the maximum allowed steps in an episode as 750. This is to make sure the agent finds the end-point (goal) within some finite amount of steps.

goal.D_c is the distance correction which is applied to penalize the agent if it chooses actions which speed up the agent away from the goal. The distance correction term is defined as follows:

$$D_c = (V_{max} - V_{now}) * t_{max} \tag{2}$$

V_{max} is the maximum velocity possible for the agent which for DQN is fixed at 2.5 m/s. V_{now} is the current velocity of the agent and t_{max} is the duration of the actuation.

We train three policies namely Policy I, Policy II, and Policy III. Each policy is learned through curriculum learning where we make the end goal farther away as the training progresses. We terminate the training once the agent has finished 1 Million steps. We evaluate the all the three policies in fp32 and quantized int8 data types for 100 evaluations in airlearning and report the success rate.

We also take these policies and characterize the system performance on a Ras-pi 3b platform. Ras-Pi 3b is a proxy for the compute platform available on the aerial robot. The hardware specification for Ras-Pi 3b is shown in Table 7.

Embedded System	**Ras-Pi 3b**
CPU Cores	4 Cores (ARM A53)
CPU Frequency	1.2 GHz
GPU	None
Power	$<$1W
Cost	$35

Table 7: Specification of Ras-Pi 3b embedded computing platform. Ras-Pi 3b is a proxy for the on-board compute platform available in the aerial robot.

We allocate a region of storage space as swap memory. It is the region of memory allocated in disk that is used when system memory is utilized fully by a process. In Ras-Pi 3b, the swap memory is allocated in Flash storage.

B.4 Post-Training Quantization Sweet-Spot

Here we demonstrate that post training quantization can regularize the policy. To that end, we take a pre-trained policy for three different Atari environments and quantize it from 32-bits (fp32) to 2-bit using uniform affine quantization. Figures 7 shows that there is a sweet-spot for post-training quantization. Sometimes, quantizing to fewer bits outperforms higher precision quantization. Each plot was generated by applying post-training quantization to the full precision baselines and evaluating over 10 runs.

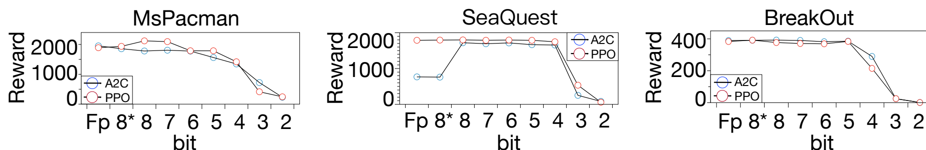


Figure 7: Post training quantization sweet spot for DQN MsPacman, DQN SeaQuest, DQN Breakout. We see that post-training quantization sweet spot depends on the specific task at hand. Note that 16-bit in this plot is 16-bit affine quantization, not fp16.

C Related Work and Motivation

Both quantization and reinforcement learning in isolation have been the subject of much research in recent years. Below we provide an overview of related works in both quantization and reinforcement learning and discuss their contributions and significance in relation to our paper.

C.1 QUANTIZATION

Quantizing a neural network reduces the precision of neural network weights, reducing memory transfer times and enabling the use of fast low-precision compute operations. Traditionally, there are two main methods for obtaining a quantized model: post training quantization and quantization aware training. Post training quantization involves taking a pretrained model and quantizing its weights while quantization aware training modifies the neural network training process so that the resulting learned weights are quantized. Innovations in both post-training quantization Krishnamoorthi (2018); Banner et al. (2018); Zhao et al. (2019) and quantization aware training Dong et al. (2019); Hubara et al. (2018); Choi et al. (2018) have demonstrated that neural networks may be quantized to very low precision without accuracy loss, suggesting that quantization has immense potential for producing efficient deployable models. In the context of speeding up training, research has also shown that quantization can yield significant performance boosts. For example, prior work on half or mixed precision training Sun et al. (2019); Das et al. (2018) demonstrates that using half-precision operators may significantly reduce compute and memory requirements while still achieving adequate convergence.

While much research has been conducted on quantization and machine learning, the primary targets of quantization are applications in the image classification and natural language processing domains. Quantization as applied to reinforcement learning has been relatively absent in the literature. In our work we comprehensively evaluate these standard quantization ideas on a wide range of reinforcement learning tasks to verify the efficacy of quantization in reinforcement learning. Additionally, unlike standard neural network training, training reinforcement learning models relies on the actor-learner paradigm, and we show that we can leverage 8-bit quantization to speed up distributed reinforcement learning significantly. This is unlike standard quantized neural network training, which is primarily limited to half precision (16-bit) training due to convergence issues.

C.2 REINFORCEMENT LEARNING & DISTRIBUTED REINFORCEMENT LEARNING TRAINING

Reinforcement learning is traditionally defined as a learning paradigm where an agent interacts with its environment to maximize expected discounted cumulative reward. Various significant work on reinforcement learning range from training algorithms Mnih et al. (2013); Levine et al. (2015) to environments Brockman et al. (2016); Bellemare et al. (2013); Tassa et al. (2018) to systems improvements Petrenko et al. (2020); Hoffman et al. (2020). From a systems angle, reinforcement learning poses a unique opportunity (as opposed to standard machine learning methods) as training a policy involves executing policies on environments (experience generation) and optimization (learning). Experience generation is trivially parallelizable and various recent research in distributed and parallel reinforcement learning training Kapturowski et al. (2018); Moritz et al. (2018); Nair et al. (2015) leverage this to accelerate training. One significant work is the Deepmind Acme reinforcement learning framework Hoffman et al. (2020), which enables scalable training to many processors or nodes on a single machine. Notably, their paper shows how industry caliber reinforcement learning training is scaled: they use a single fast GPU-based learner which performs optimization, and many slower CPU-based actors to gather experience. As the GPU based learner is much faster than the CPU-based actors (leveraging large batch sizes to improve data consumption speeds), many CPU-based actors are required in order to keep up with the GPU-based learner. Other work on scalable reinforcement learning training Espeholt et al. (2019); Petrenko et al. (2020) also mirror this actor-learner setup, with high performance GPU-based learners and lower performance CPU-based actors.

While prior work has demonstrated the immense systems component of designing an efficient and scalable reinforcement learning training system, research cutting across scalable reinforcement learning and quantization has largely been absent. In our work, we build off of the Deepmind Acme framework to develop a quantized reinforcement learning training system and demonstrate that, by speeding up actors in the actor-learner training loop, quantization has significant potential for scaling reinforcement learning training. Additionally, we discuss important design decisions and trade-offs when applying quantization to distributed reinforcement learning (such as impacts of quantization on convergence, compute/communication time, etc) and give several insights to success.