ActorQ: Quantization for Actor-Learner Distributed Reinforcement Learning

Maximilian Lam*, Sharad Chitlangia*, Srivatsan Krishnan*, Zishen Wan,
Gabriel Barth-Maron, Aleksandra Faust, Vijay Janapa Reddi

Harvard University

Reinforcement Learning Training

Training reinforcement learning models is fundamentally resource intensive due to

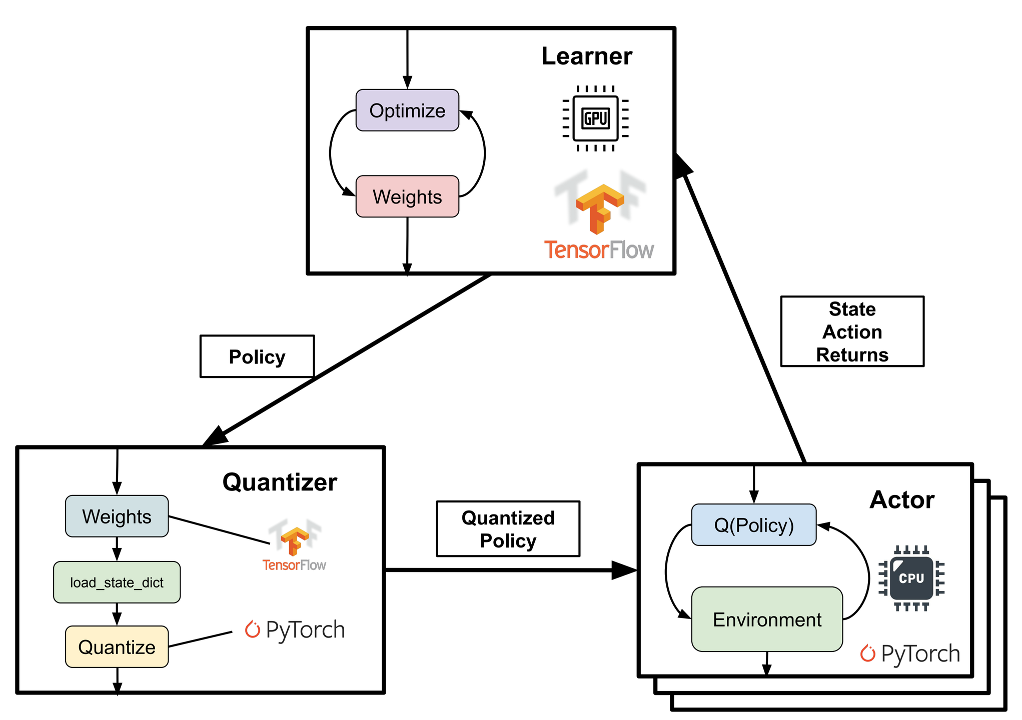
1. The **computationally expensive** nature of deep neural networks
2. The **sample inefficiency** of reinforcement learning algorithms

Applying quantization to reinforcement learning is nontrivial and different from traditional neural network.

1. In the context of policy inference, due to the sequential decision making nature of reinforcement learning, errors made at one state might propagate to subsequent states.
2. In the context of reinforcement learning training, quantization seems difficult to apply due to the myriad of different algorithms (A2C, DDPG, DQN, etc) and the 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 for exploration, making them robust. On the latter point, we leverage the fact that reinforcement learning procedures may be framed through the actor-learner training paradigm, and rather than quantizing learner optimization, we may achieve speedups while maintaining convergence by quantizing just the actors' experience generation.

ActorQ

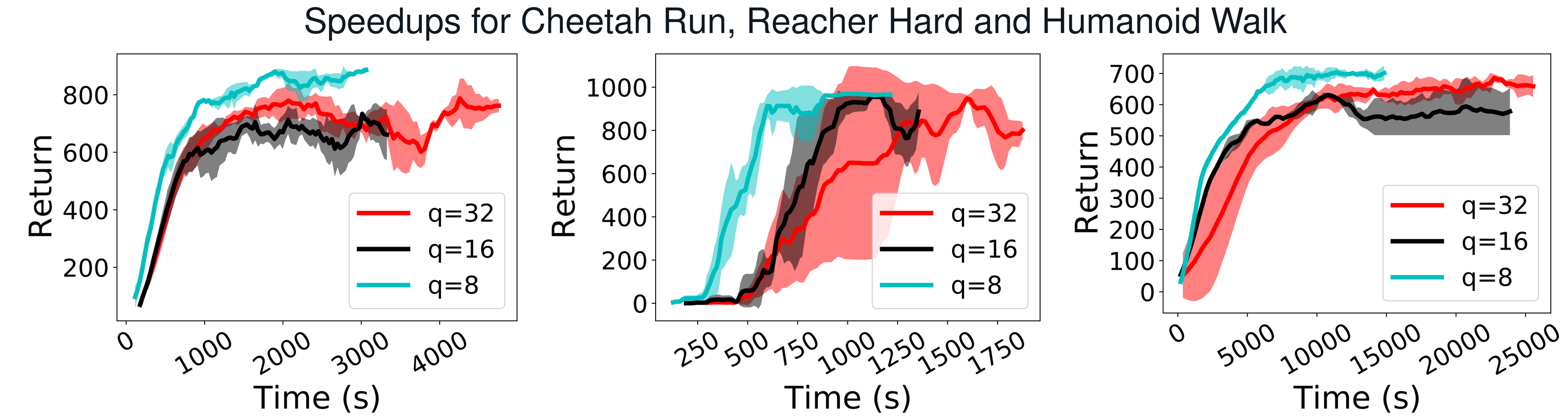


1. **Learner on GPUs; Actors on CPUs**: Learners perform batched optimization on GPUs; Multiple parallel actors perform inference to generate data
2. **Tensorflow On Learner; Pytorch on Actors**: Pytorch's Quantized Inference allows the actors to speedup data generation.
3. **Quantize Compute v/s Quantize Communication**: Actors can be quantized to perform 8-bit or 16-bit and generate data faster. Or Communication can be quantized to any number of bits.
4. **Separate Parameter Quantizer Process**: Aids in not burdening the learning with the conversion processes.
5. **Asynchronous Model Pushes on Learner Side; Synchronous Model Pulls on Actor Side**: Asynchronous Pushes maximizes learner resource usage. Synchronous Model Pulls on Actors to avoid stale models

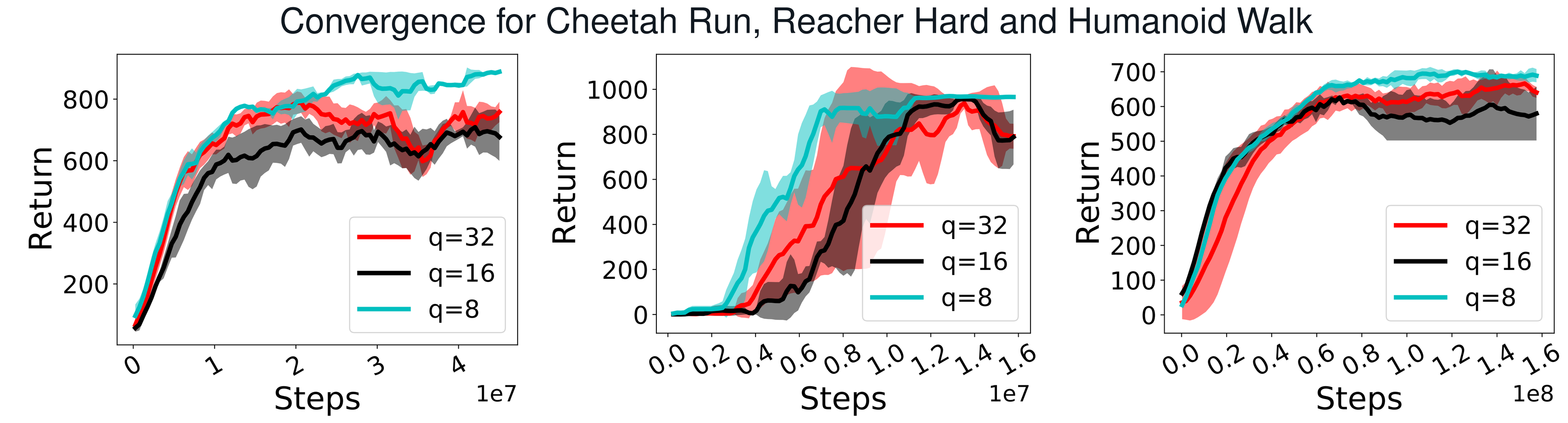
Results

We evaluate the *ActorQ* algorithm for speeding up quantized distributed reinforcement learning across various environments. Overall, we show that: 1) we see significant speedup (>1.5 ×-2.5 ×) in training reinforcement learning policies using *ActorQ* and 2) convergence is maintained even when actors perform down to 8 bit quantized execution. 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. We choose the environments to cover a wide range of difficulties to determine the effects of quantization on both easy and difficult tasks. 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).

Speedups for Cheetah Run, Reacher Hard and Humanoid Walk



Convergence for Cheetah Run, Reacher Hard and Humanoid Walk

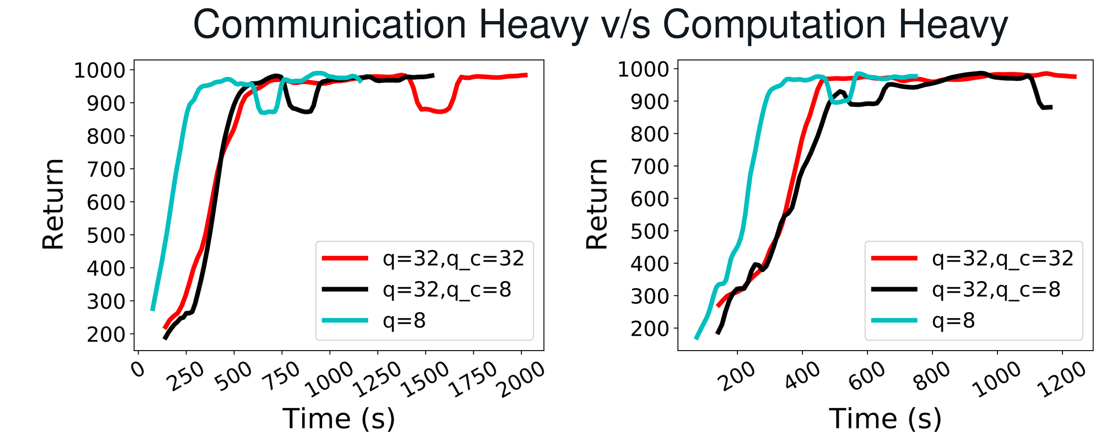


Policy architectures are fully connected networks with 3 hidden layers of size 2048. We apply a gaussian noise layer to the output of the policy network on the actor to encourage additional exploration; sigma is uniformly assigned between 0.0 and 0.2 according to the actor being executed. On the learner side, the critic network is a 3 layer hidden network with size 512. We train policies using D4PG on continuous control environments and DQN on discrete control environments. 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.

Task	Reward Achieved	FP32 Time to Reward (s)	Int8 Time to Reward (s)	Int8 Speedup
Cartpole Balance	941.22	870.91	279.00	3.12
Walker Stand	947.74	871.32	534.37	1.63
Hopper Stand	836.41	2660.41	1699.17	1.57
Reacher Hard	948.12	1597.00	875.34	1.82
Cheetah Run	732.31	2517.30	891.84	2.82
Finger Spin	810.32	3256.56	1065.52	3.06
Humanoid Stand	884.89	13964.92	9302.82	1.51
Humanoid Walk	649.91	17990.66	6223.35	2.89
Cartpole (Gym)	198.22	963.67	260.10	3.70
Mountain Car (Gym)	-120.62	2861.80	1284.32	2.22
Acrobot (Gym)	-107.45	912.24	168.44	5.41

Comparison

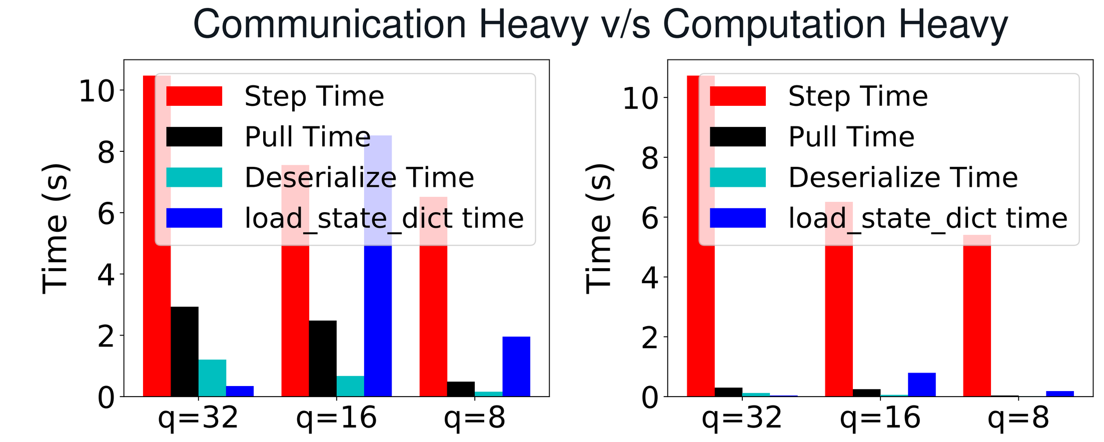
Communication Heavy v/s Computation Heavy



We further break down the various components contributing to runtime on a single actor. Runtime components are broken down into:

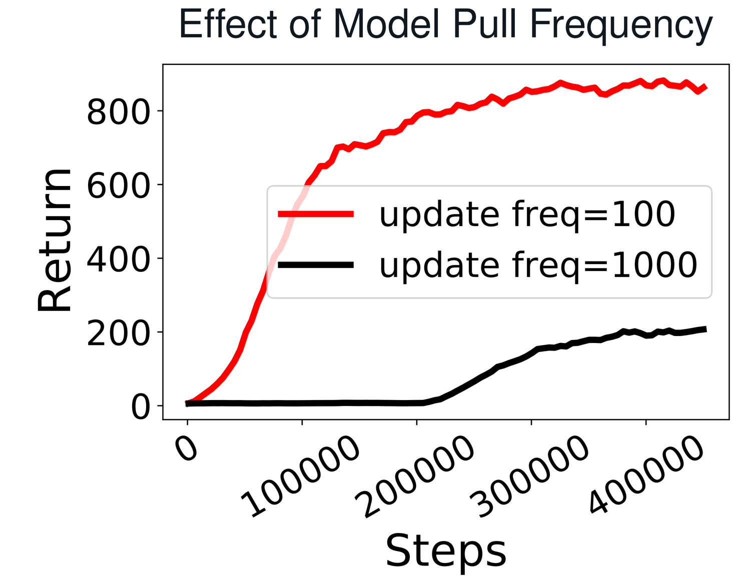
1. **Step** time is the time spent performing neural network inference
2. **Pull** time is the time between querying the Replay Buffer for a model and receiving the serialized models weights
3. **deserialize** time is the time spent to deserialize the serialized model dictionary
4. **load_state_dict** time is the time to call PyTorch load_state_dict.

Communication Heavy v/s Computation Heavy



As evident, the step time (neural network inference time) is the biggest bottleneck during training. This can be optimized by running the actors at a quantized precision. It is observed that quantizing actors also leads to lesser pull time and deserialize time due to reduction in memory.

Effect of Model Pull Frequency



Finally, we investigate how much model staleness can affect the convergence of an agent. The figure above shows that in distributed RL training, model full frequency can be one of the most important hyperparameters affecting the final reward by $5\times$. In a large scale distributed RL setup with a networked cluster, quantizing communication can help reducing congestion and free up bandwidth for faster training.