**ActorQ: Quantization for Actor-Learner Distributed Reinforcement Learning**

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

---

### 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, and rather than quantizing learner optimization, we may achieve speedups while maintaining convergence by quantizing just the actors’ experience generation.

---

### ActorQ

![ActorQ Diagram]

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 speedups (>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

<table>
<thead>
<tr>
<th>Environment</th>
<th>FPS32 Time to Reward (s)</th>
<th>FP32 Time to Reward (s)</th>
<th>FP32 Int8 Time to Reward (s)</th>
<th>FP32 Int8 Int8 Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartpole Balance</td>
<td>941.22</td>
<td>870.91</td>
<td>279.00</td>
<td>3.12</td>
</tr>
<tr>
<td>Walker Stand</td>
<td>947.74</td>
<td>871.32</td>
<td>534.37</td>
<td>1.63</td>
</tr>
<tr>
<td>Hopper Stand</td>
<td>836.41</td>
<td>2660.41</td>
<td>1699.17</td>
<td>1.57</td>
</tr>
<tr>
<td>Reacher Hard</td>
<td>948.12</td>
<td>1597.00</td>
<td>875.34</td>
<td>1.82</td>
</tr>
<tr>
<td>Cheetah Run</td>
<td>732.31</td>
<td>2517.30</td>
<td>891.84</td>
<td>2.82</td>
</tr>
<tr>
<td>Finger Spin</td>
<td>810.32</td>
<td>3256.56</td>
<td>1065.52</td>
<td>3.06</td>
</tr>
<tr>
<td>Humanoid Stand</td>
<td>884.89</td>
<td>13984.92</td>
<td>9302.82</td>
<td>1.51</td>
</tr>
<tr>
<td>Humanoid Walk</td>
<td>649.91</td>
<td>17996.66</td>
<td>6223.35</td>
<td>2.89</td>
</tr>
<tr>
<td>Cartpole (Gym)</td>
<td>198.22</td>
<td>963.67</td>
<td>260.10</td>
<td>3.70</td>
</tr>
<tr>
<td>Mountain Car (Gym)</td>
<td>120.62</td>
<td>2861.80</td>
<td>1284.32</td>
<td>2.22</td>
</tr>
<tr>
<td>Acrobat (Gym)</td>
<td>107.45</td>
<td>912.24</td>
<td>168.44</td>
<td>5.41</td>
</tr>
</tbody>
</table>

---

### Comparison

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. **deserialzier 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.

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 5x.

In a large scale distributed RL setup with a networked cluster, quantizing communication can help reducing congestion and free up bandwidth for faster training.