Accelerating Distributed Deep Reinforcement Learning by In-Network Experience Sampling

Masaki Furukawa, Hiroki Matsutani
(Keio University, Japan)
Background: Reinforcement learning

- **Goal**
  - Acquire an action-selection policy that maximizes a long-term reward by taking actions and observing the environment

- **Q-value**
  - Expected value for action $a$ in state $s$

- **Q-learning algorithm**
  - $Q(s, a)$ is updated by taking action $a_t$ and observing the next state $s_{t+1}$ and reward $r_t$

\[
Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left\{ r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t) \right\}
\]
Background: Deep Q-Network

- **Q-learning**
  \[
  Q(s_t, a_t) = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a)
  \]
  - Estimated optimal future value
  - Set of all the possible actions

- **DQN**: Q-learning + Deep neural network
  - Q-value is approximated with deep neural network \( \theta \)
  - \( \theta \) is updated to minimize the loss
    \[
    Q(s_t, a_t; \theta) = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta)
    \]
    \[
    L(\theta) = \left\{ r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right\}^2
    \]
  - Game AI, robot control, communication control, …
  - Various techniques are used for stability and convergence
Background: DQN techniques

• Experience replay
  – Remove correlations in the observation sequence
  – Reuse experiences to increase sampling efficiency

\[ a_t \leftarrow \varepsilon\text{-greedy}(A) \]

\[ (s_t, a_t, r_t, s_{t+1}) \]

Environment
Q-network

Replay buffer

DQN-loss

Replay
Store
Random
Shuffle

• Distributed deep reinforcement learning
  – Generate more experiences using many machines
  – Ape-X DQN [1] (Distributed prioritized experience replay)

Background: Ape-X

• Distributed prioritized experience replay [1]

1. Pull parameters $\theta$
2. Prediction
   Action:
   \[ a_t \leftarrow \text{\varepsilon-greedy}(A) \]
   Next state & reward:
   \[ s_{t+1}, r_t \leftarrow \text{Environment}(a_t, s_t) \]

LocalBuffer.ADD($s_t, a_t, r_t, s_{t+1}$)

if LocalBuffer.Size $\geq$ BatchSize:
   Priority:
   \[ |\delta_t| \leftarrow |r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)| \]

3. Push experience
   \[ \{a_t, s_t, r_t, s_{t+1}, \delta_t\} \times \text{BatchSize} \]

Background: Ape-X

- Distributed prioritized experience replay \[1\]

1. Experience sampling
   \[\{a_t, s_t, r_t, s_{t+1}, \delta_t\} \times \text{BatchSize} \]

2. Batch training
   \[Q(s_t, a_t; \theta) \leftarrow r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta)\]

3. Update priorities

4. Set parameters \(\theta\)

Background: Prioritized exp. replay

- **Stochastic sampling** [2]
  - Priority: \( p_t = |\delta_t| = |r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)| \)
  - Sampling probability for experience \( i \)
    \[
    P_i = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (p_i \neq 0)
    \]
- **Sum-tree structure**
  - Add/Sample experience: \( O(\log N) \)

![Sum-tree diagram]

Baseline: Our DRL implementation

- Distributed deep reinforcement learning (DRL)
  - Actors, shared memory (Redis), and learner

Atari's Breakout
https://gym.openai.com

1. **Pull** parameters (Dueling network architecture [3] 13MB)
2. Prediction
3. **Push** experiences (43MB)

Baseline: Our DRL implementation

- Distributed deep reinforcement learning (DRL)
Baseline: Execution time breakdown

- Distributed deep reinforcement learning (DRL)

Communication cost: 14.5% (1 actor) – 19.0% (8 actors)

<table>
<thead>
<tr>
<th>Actors</th>
<th>Communication cost</th>
<th>Learner machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Actor</td>
<td>14.5% (1 actor)</td>
<td>Ubuntu 20.04</td>
</tr>
<tr>
<td>Learner</td>
<td></td>
<td>Intel Xeon CPU E5-2637 v3 @3.50GHz</td>
</tr>
<tr>
<td>2 Actors</td>
<td>19.0% (8 actors)</td>
<td>Intel Xeon CPU E5-1620 v2 @3.5GHz</td>
</tr>
<tr>
<td>Learner</td>
<td></td>
<td>Intel Ethernet CNA XL710-QDA2</td>
</tr>
<tr>
<td>4 Actors</td>
<td></td>
<td>11.3 / 1.80+cu111</td>
</tr>
<tr>
<td>Learner</td>
<td></td>
<td>Intel Ethernet CNA XL710-QDA2</td>
</tr>
<tr>
<td>8 Actors</td>
<td></td>
<td>20.11.0</td>
</tr>
<tr>
<td>Learner</td>
<td></td>
<td>5.0.5</td>
</tr>
</tbody>
</table>

System specifications:
- Ubuntu 20.04
- Intel Xeon CPU E5-2637 v3 @3.50GHz
- Intel Xeon CPU E5-1620 v2 @3.5GHz
- Intel Ethernet CNA XL710-QDA2
- 11.3 / 1.80+cu111
- 20.11.0
- 5.0.5
- Mellanox 40G Switch SX1012
Baseline: Shared mem throughput

- Distributed deep reinforcement learning (DRL)
- Actor push frequency
  - Increase as # of actors
- Increase of # of actors
  - Network traffic increases
  - Gap between ideal and actual frequency increases

---

### Actors machine

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 20.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2637 v3 @3.5GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>GeForce RTX 3080 ×1</td>
</tr>
<tr>
<td>CUDA/PyTorch</td>
<td>11.3 / 1.8.0+cu111</td>
</tr>
<tr>
<td>NIC</td>
<td>Intel Ethernet CNA XL710-QDA2</td>
</tr>
<tr>
<td>DPDK</td>
<td>—</td>
</tr>
<tr>
<td>Redis</td>
<td>—</td>
</tr>
<tr>
<td>Network</td>
<td>Mellanox 40G Switch SX1012</td>
</tr>
</tbody>
</table>

### Shared/Replay memory machine

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 20.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon CPU E5-2637 v3 @3.50GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>512 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>—</td>
</tr>
<tr>
<td>CUDA/PyTorch</td>
<td>11.3 / 1.8.0+cu111</td>
</tr>
<tr>
<td>NIC</td>
<td>Intel Ethernet CNA XL710-QDA2</td>
</tr>
<tr>
<td>DPDK</td>
<td>20.11.0</td>
</tr>
<tr>
<td>Redis</td>
<td>5.0.5</td>
</tr>
<tr>
<td>Network</td>
<td>—</td>
</tr>
</tbody>
</table>

### Learner machine

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 20.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon CPU E5-1620 v2 @3.50GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>GeForce GTX 1080Ti ×2</td>
</tr>
<tr>
<td>CUDA/PyTorch</td>
<td>11.3 / 1.8.0+cu111</td>
</tr>
<tr>
<td>NIC</td>
<td>Intel Ethernet CNA XL710-QDA2</td>
</tr>
<tr>
<td>DPDK</td>
<td>—</td>
</tr>
<tr>
<td>Redis</td>
<td>—</td>
</tr>
<tr>
<td>Network</td>
<td>—</td>
</tr>
</tbody>
</table>
Network optimizations on DRL

• Our approach
  – Accelerating **shared memory & experience replay** using DPDK

• DPDK (Data Plane Development Kit)
  – Network processing at application layer
  – Polling by dedicated CPU cores

• F-stack
  – Light-weight TCP/IP stack for DPDK applications

• Related work [4]
  – Accelerating gradient aggregation of distributed deep learning at DPDK-based network switch

Optimization 1: Shared memory

- In-network shared memory (Redis) by DPDK
  - Low-latency shared memory by DPDK and F-stack
Optimization 1: Shared memory

- In-network shared memory (Redis) by DPDK
  - Low-latency shared memory by DPDK and F-stack

Redis access latency:
32.7% to 58.9% decrease

Replay memory pull throughput:
21.9% to 31.9% increase
Optimization 2: Experience replay

- In-network experience sampling by DPDK
  - Moved from leaner machine to in-network switch

![Diagram]

- Actors
- Environment
- DPDK+F-stack (Userspace TCP/IP stack)
- Sampling
- Replay memory
- Learner (Train)
- Kernel
- 40GbE NIC
- Sampled experiences (Batch size: 512)
Optimization 2: Experience replay

- In-network experience sampling by DPDK
  - Moved from leaner machine to in-network switch

A micro-thread is launched for each Actor process to multiplex the network I/O.
Optimization 2: Experience replay

- In-network experience sampling by DPDK
  - Moved from leaner machine to in-network switch

Replay memory access latency is reduced by 11.7%-28.1%
Sampling latency of learner is reduced by 21.9%-29.8%
Future work: Deployed in edge server

- Experience sampling in edge server
  - Actors located in edge, and learner located in cloud

![Diagram showing the connections between edge computers, edge server, datacenter, and cloud. The diagram includes arrows indicating the flow of information and a 10km optical cable.]
Summary: Optimizations in DRL

- Communication cost reduction by DPDK+F-stack
  - Optimization 1: In-network shared memory (Redis)
  - Optimization 2: In-network experience sampling

Future work: Deployment in edge-cloud system