Asynchronous Methods for Deep Reinforcement
Mnih, Badia, Mirza, Graves, Lillicrap, Harley, Silver, Kavukcuoglu

Topic: Actor Critic methods
Presenter: Adelin Travers
Motivation

Learn from raw pixels, not states
Motivation

- Experience replay
  - Data from previous experiences stored in dedicated memory
- At each step:
  - Can batch data
  - Can sample randomly

=> Augments stability

- reducing non-stationarity
- decorrelates updates
Problem

- Only off policy learning
  - Data generated from a previous policy.
- High memory usage
- High computational cost per interaction with the environment

Previous approaches based on compute parallelization:

- Specialized hardware such as GPU
- Massively distributed architectures
Outline

- Contributions
- Background
- Algorithms
- Experimental results
- Discussion
- Limitations and open issues
Contributions

• Investigate alternatives to replay memory
• Previous work parallelized agents and shared replay memory
• Propose to parallelize the learning experience
• Duplicate both the agents and environments
• Learning is shared among the agents but experience is not
  • Obtain a more stationary process and speed up exploration
• Demonstrate deep RL for value-, policy-based methods both On- and off-policy
• Divide by 2 the state of the art training time while on a single server’s 16 CPUs
Outline

• Contributions
• Background
• Algorithms
• Experimental results
• Discussion
• Limitations and open issues
Background

One-step Q-learning

\[ L_i(\theta_i) = \mathbb{E} \left( r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right)^2 \]

N-step Q-learning

\[ r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q(s_{t+n}, a) \]
Background

- Actor-critic
- Reduce Monte-carlo policy gradients variance
- Combine Value based methods and policy gradients

[David Silver, RL Lectures]
Background

- Parameterize the Q-value function

\[
Q_w(s, a) \approx Q^{\pi_\theta}(s, a)
\]

- Approximate policy gradient

\[
\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta} \left[ \nabla_\theta \log \pi_\theta(s, a) \ Q_w(s, a) \right]
\]

\[
\Delta \theta = \alpha \nabla_\theta \log \pi_\theta(s, a) \ Q_w(s, a)
\]

[David Silver, RL Lectures]
Background

• Critic can be a baseline
• Can take the value function

• Policy gradient on the advantage function

\[ A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \]
\[ \nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[ \nabla_\theta \log \pi_\theta(s, a) \ A^{\pi_\theta}(s, a) \right] \]

[David Silver, RL Lectures]
Outline

• Contributions
• Background
• Algorithms
• Experimental results
• Discussion
• Limitations and open issues
Algorithm: one-step Q-learning

repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} 
        r & \text{for terminal } s' \\
        r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s'
    \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial(y - Q(s, a; \theta))^2}{\partial\theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} == 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
Algorithm: one-step Q-learning

repeat

Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$

Receive new state $s'$ and reward $r$

$y = \begin{cases} 
  r & \text{for terminal } s' \\
  r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s'
\end{cases}$

Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial(y-Q(s,a;\theta))}{\partial\theta}$

$s = s'$

$T \leftarrow T + 1$ and $t \leftarrow t + 1$

if $T \mod I_{\text{target}} == 0$ then

   Update the target network $\theta^- \leftarrow \theta$

end if

if $t \mod I_{\text{AsyncUpdate}} == 0$ or $s$ is terminal then

   Perform asynchronous update of $\theta$ using $d\theta$.
   Clear gradients $d\theta \leftarrow 0$.

end if

until $T > T_{\text{max}}$
Algorithm: one-step Q-learning

repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} 
    r & \text{for terminal } s' \\
    r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s'
    \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial(y - Q(s,a;\theta))^2}{\partial\theta}$
    $s \leftarrow s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} = 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} = 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
Algorithm: one-step Q-learning

repeat
  Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
  Receive new state $s'$ and reward $r$
  
  $y = \begin{cases} 
  r & \text{for terminal } s' \\
  r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s'
  \end{cases}$

  Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial(y - Q(s, a; \theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$

  if $T \mod I_{\text{target}} == 0$ then
    Update the target network $\theta^- \leftarrow \theta$
  end if

  if $t \mod I_{\text{AsyncUpdate}} == 0$ or $s$ is terminal then
    Perform asynchronous update of $\theta$ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if

until $T > T_{max}$
Algorithm: A3C

repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta_v' = \theta_v$.
  $t_{\text{start}} = t$
  Get state $s_t$
  repeat
    Perform $a_t$ according to policy $\pi(a_t|s_t; \theta')$
    Receive reward $r_t$ and new state $s_{t+1}$
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal $s_t$ or $t - t_{\text{start}} = t_{\text{max}}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v') & \text{for non-terminal } s_t \end{cases}$
  for $i \in \{t - 1, \ldots, t_{\text{start}}\}$ do
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta_v'))$
    Accumulate gradients wrt $\theta_v'$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta_v'))^2 / \partial \theta_v'$
  end for
  Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{\text{max}}$
Outline

• Contributions
• Background
• Algorithms
• Experimental results
• Discussion
• Limitations and open issues
Experimental Results

All variants outperform DQN in training speed and performance
## Experimental Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Training Time</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>8 days on GPU</td>
<td>121.9%</td>
<td>47.5%</td>
</tr>
<tr>
<td>Gorila</td>
<td>4 days, 100 machines</td>
<td>215.2%</td>
<td>71.3%</td>
</tr>
<tr>
<td>D-DQN</td>
<td>8 days on GPU</td>
<td>332.9%</td>
<td>110.9%</td>
</tr>
<tr>
<td>Dueling D-DQN</td>
<td>8 days on GPU</td>
<td>343.8%</td>
<td>117.1%</td>
</tr>
<tr>
<td>Prioritized DQN</td>
<td>8 days on GPU</td>
<td>463.6%</td>
<td>127.6%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>1 day on CPU</td>
<td>344.1%</td>
<td>68.2%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>4 days on CPU</td>
<td>496.8%</td>
<td>116.6%</td>
</tr>
<tr>
<td>A3C, LSTM</td>
<td>4 days on CPU</td>
<td>623.0%</td>
<td>112.6%</td>
</tr>
</tbody>
</table>

2x speedup on CPU
Outline

• Contributions
• Background
• Algorithms
• Experimental results
• Discussion
• Limitations and open issues
### Discussion

Superlinear mean thread improvement for all methods but A3C

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-step Q</td>
<td>1.0</td>
<td>3.0</td>
<td>6.3</td>
<td>13.3</td>
<td>24.1</td>
</tr>
<tr>
<td>1-step SARSA</td>
<td>1.0</td>
<td>2.8</td>
<td>5.9</td>
<td>13.1</td>
<td>22.1</td>
</tr>
<tr>
<td>n-step Q</td>
<td>1.0</td>
<td>2.7</td>
<td>5.9</td>
<td>10.7</td>
<td>17.2</td>
</tr>
<tr>
<td>A3C</td>
<td>1.0</td>
<td>2.1</td>
<td>3.7</td>
<td>6.9</td>
<td>12.5</td>
</tr>
</tbody>
</table>
Discussion

1-step Q

N-step Q

A3C

Thread speedup is dependent on the games
Discussion

Capable of handling discrete and continuous state spaces
Outline

• Contributions
• Background
• Algorithms
• Experimental results
• Discussion
• Limitations and open issues
Limitations and Open Issues

• Performance very dependent on the game
• If interactions with the environment are expensive, limited success
  • Combine with experience replay?
• Forward view only
  • Backward view is more common in RL
• Better ways to estimate the advantage function
  • Generalized advantage estimation
Contributions (recap)

- Alternatives to replay memory
- Previous work parallelized replay memory/computation
- Parallelize the learning experience
- Duplicate both the agents and environments
- Learning is shared among the agents but experience is not
  - Obtain a more stationary process and speed up exploration
- Demonstrate deep RL for value-, policy-based methods both On- and off-policy
- Divide by 2 the state of the art training time while on a single server’s 16 CPUs