Asynchronous Methods for Deep Reinforcement Learning

Paper by Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu

Presented by: Pihel Saatmann
Reinforcement learning

- **State** – „snapshot“ of the environment
- **Action** – leads to new state, sometimes reward
- **Reward** – time delayed, sparse
- **Policy** – rules for choosing action
So far

• Thought that online RL algorithms with deep NN-s are unstable.
• Problems - correlated and non-stationary input data.

• To counter these problems data can be stored in experience replay memory.
• This uses more memory/computational power.

• Deep RL methods require specialized hardware (GPUs) or massive distributed architectures.
Q-learning

• At each time step \( t \), the agent receives a state \( s_t \) and selects an action \( a \) according to its policy \( \pi \). Then the agent gets the next state \( s_{t+1} \) and a scalar reward \( r_t \).

• The goal is to maximize the expected return from each state \( s_t \).

• Q function estimates the action’s value.

• Each time the agent does an action the Q value is updated.

• Off-policy method – updating Q fn does not depend on policy.

\[
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \cdot \left( r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)
\]

\( Q(s_t, a_t) \) is the Q value, \( Q(s_t, a_t) \) is the old value, and \( \alpha_t \) is the learning rate.
Asynchronous RL framework

• Instead of experience replay they asynchronously execute multiple agents in parallel on multiple instances of the environment.

• Parallel actor-learners have a stabilizing effect on training.
• Runs on a single machine with a standard multi-core CPU.
Asynchronous RL framework II

• Async variants of four standard RL algorithms:
  • 1-step Q-learning
  • N-step Q-learning
  • 1-step Sarsa
  • Advantage actor-critic (A3C)
1-step Q-learning

- NN is used to approximate the $Q(s, a; \Theta)$ function.
- The parameters (weights) $\Theta$ are learned by iteratively minimizing a sequence of loss functions, where the i-th loss function is defined as:

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

- Each thread has own copy of environment.
- At each step computes a gradient of the Q-learning loss.
- Accumulate gradients over multiple timesteps before applying.
- Shared and slowly changing target network.

```
Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ~, and counter T = 0.
Initialize thread step counter t ← 0
Initialize target network weights θ~ ← θ
Initialize network gradients dθ ← 0
Get initial state s

repeat
    Take action a with ε-greedy policy based on Q(s, a; θ)
    Receive new state s' and reward r
    y = \begin{cases} 
        r & \text{for terminal } s' \\
        r + γ \max_{a'} Q(s', a'; θ~) & \text{for non-terminal } s'
    \end{cases}
    Accumulate gradients wrt θ: dθ ← dθ + \frac{∂(y - Q(s, a; θ))}{∂θ}
    s = s'
    T ← T + 1 and t ← t + 1
    if T mod T_{target} == 0 then
        Update the target network θ~ ← θ
    end if
    if t mod T_{AsyncUpdate} == 0 or s is terminal then
        Perform asynchronous update of θ using dθ.
        Clear asynchronous update of θ.
    end if
until T > T_{max}
```
Asynchronous 1-step Sarsa

- Same as 1-step Q-learning, but uses a different target value:

\[ r + \gamma Q(s', a'; \theta^-) \]
Asynchronous n-step Q-learning

- Potentially faster way to propagate rewards.
- Uses ‘forward-view’ - selects actions using its policy for up to n steps in the future.
- Receives up to $t_{\text{max}}$ rewards since last update.
- **Total accumulated return**: $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$
- Value fn is updated after every $t_{\text{max}}$ actions or after terminal state.
- For each update uses the longest possible n-step return.
Algorithm S2  Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared parameter vector $\theta$.
// Assume global shared target parameter vector $\theta^-$.  
// Assume global shared counter $T = 0$.
Initialization thread step counter $t \leftarrow 1$
Initialization target network parameters $\theta^- \leftarrow \theta$
Initialization thread-specific parameters $\theta' = \theta$
Initialization network gradients $d\theta \leftarrow 0$

repeat
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state $s_t$
  repeat
    Take action $a_t$ according to the $\epsilon$-greedy policy based on $Q(s_t, a; \theta')$
    Receive reward $r_t$ and new state $s_{t+1}$
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal $s_t$ or $t - t_{start} = t_{max}$
  $R = \begin{cases} 
  0 & \text{for terminal } s_t \\
  \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t
  \end{cases}$
  for $i \in \{t - 1, \ldots, t_{start}\}$ do
    $R \leftarrow r_i + \gamma R$
  end for
  Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \frac{\partial (R - Q(s_t, a_t; \theta')^2)}{\partial \theta'}$
end repeat
Perform asynchronous update of $\theta$ using $d\theta$.
if $T \mod I_{target} = 0$ then
  $\theta^- \leftarrow \theta$
end if
until $T > T_{max}$
Asynchronous advantage actor-critic

- On-policy method - has a policy $\pi(a_t|s_t; \theta)$ and estimated value function $V(s_t; \theta_v)$.
- Uses ‘forward-view’.
- Receives up to $t_{\text{max}}$ rewards since last update.
- Policy and value fn-s are updated after every $t_{\text{max}}$ actions or after terminal state.
- For each update uses the longest possible n-step return.
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors $\theta$ and $\theta_v$, and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$

repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$. 
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  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 $t - t_{start} == t_{max}$
  $R = \begin{cases} 
  0 & \text{for terminal } s_t \\
  V(s_t, \theta'_v) & \text{for non-terminal } s_t
\end{cases}$// Bootstrap from last state
  for $i \in \{t - 1, \ldots, t_{start}\}$ 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 $T > T_{max}$
Performance evaluation

• Four different platforms:
  • Atari 2600 - different games
  • TORCS 3D - car racing simulator
  • MuJoCo - physics simulator for continuous motor control (A3C only)
  • Labyrinth - finding rewards in randomly generated 3D mazes (A3C only)
Atari 2600 games

• All four methods can successfully train NN controllers.
• Async methods mostly faster than DQN (Deep Q-Network).
• Advantage actor-critic was the best.
Async A3C on 57 atari games

<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>

*Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric.*
TORCS Car Racing Simulator

• Evaluated only the A3C algorithm.
• Agent had to drive a racecar using only raw pixels as input.
• During training, the agent was rewarded for maintaining high velocity along the center of the racetrack.

https://youtu.be/0xo1Ldx3L5Q
MuJoCo Physics Simulator

• Evaluated only the A3C algorithm.
• Rigid body physics with contact dynamics.
• Continuous actions.
• In all problems A3C found good solutions in less than 24 hours of training (typically a few hours).

https://youtu.be/0xo1Ldx3L5Q
Labyrinth

• The agent was placed in random maze and had 60s to collect points.
• Apples – 1 point
• Portals – 10 points, respawned apples and agent in random locations
• Visual input only.

• The agent learned a resonably good general strategy for exploring random mazes.

https://youtu.be/nMR5mjCFZCw
Scalability

• The framework scales well with the number of parallel workers.
• Even shows superlinear speedups for some methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1-step Q</td>
<td>1.0</td>
</tr>
<tr>
<td>1-step SARSA</td>
<td>1.0</td>
</tr>
<tr>
<td>n-step Q</td>
<td>1.0</td>
</tr>
<tr>
<td>A3C</td>
<td>1.0</td>
</tr>
</tbody>
</table>

*Table 2. The average training speedup for each method and number of threads averaged over seven Atari games.*
Figure 3. Data efficiency comparison of different numbers of actor-learners for three asynchronous methods on five Atari games. The x-axis shows the total number of training epochs where an epoch corresponds to four million frames (across all threads). The y-axis shows the total score achieved.
Figure 4. Training speed comparison of different numbers of actor-learners on five Atari games. The x-axis shows training time.
Robustness and stability

• Trained models on five games using 50 different learning rates and random initialization.

• Each game and algorithm combination had a range of learning rates for which all random initializations achieved good scores.

• Stability indicated by virtually no 0 scores in regions with good learning rates.
To summarize

• Able to train neural network controllers on a variety of domains in stable manner.
• Using parallel actor learners to update a shared model stabilized the learning process (alternative to experience replay).
• In Atari games the advantage actor-critic (A3C) surpassed the current state-of-the-art in half the training time.
• Superlinear speedup when increasing thread count for 1-step methods.