Asynchronous Methods for Deep Reinforcement Learning

Ashwinee Panda, 6 Feb 2019
Reinforcement Learning Background
Value-based Methods

- Don’t learn policy explicitly
- Learn Q-function
  - Deep RL: Train neural network to approximate Q-function

\[ Q_\phi(s, a) \leftarrow r(s, a) + \gamma \max_{a'} Q_\phi(s', a') \]

\[ a = \arg \max_a Q_\phi(s, a) \]
Policy-based Methods

- Parametrize policy with theta and update theta with gradient descent

REINFORCE algorithm:

1. sample \( \{\tau^i\} \) from \( \pi_{\theta}(a_t|s_t) \) (run it on the robot)
2. \( \nabla_{\theta} J(\theta) \approx \sum_i \left( \sum_t \nabla_{\theta} \log \pi_{\theta}(a_t^i|s_t^i) \right) \left( \sum_t r(s_t^i, a_t^i) \right) \)
3. \( \theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta) \)

- Reduce variance by subtracting an NN baseline
- Use learned estimate of value function as baseline
- This baseline is a “critic” => “actor-critic”
Policy-based Methods

- Reduce variance by subtracting an NN baseline
- Use learned estimate of value function as baseline
- This baseline is a “critic” => “actor-critic”

batch actor-critic algorithm:

1. sample \( \{s_i, a_i\} \) from \( \pi_\theta(a|s) \) (run it on the robot)
2. fit \( \hat{V}_\phi(s) \) to sampled reward sums
3. evaluate \( \hat{A}_\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}_\phi(s'_i) - \hat{V}_\phi(s_i) \)
4. \( \nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}_\pi(s_i, a_i) \)
5. \( \theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \)
Motivations
General RL Computation
Deep Reinforcement Learning

● Problem: Instability
  ○ Cause: Correlation between samples
  ○ Cause: Incremental updates to Q change the policy => distribution
  ○ Cause: Correlation between Q-values and target values

● Solution: Experience Replay
  ○ Randomize over data distribution, removing correlations

● Problem: Limits us to off-policy RL
General RL Computation

When possible, reuse data

process 1: data collection

\((s, a, s', r)\)

\(\pi(a|s)\) (e.g., \(\epsilon\)-greedy)

data collection

dataset of transitions (“replay buffer”)

current parameters \(\phi\)

target parameters \(\phi'\)

process 2 target update

process 3 Q-function regression

evict old data
Ideas in Asynchronous RL
History of Dist. RL

- DQN (already saw this)
- Gorilla - param. Server
  - Replay buffer
  - Enables multiple machines
- Problems w parameter servers?
  - Communication cost
  - Unavoidable when using multiple machines
Asynchronous Actor-Learners

- Rather than separate machines coordinated by a parameter server...
- Multiple CPU threads on single machine coordinated by OS
- Removes communication costs (so what?)
- Actors walk through environment and send updates to learners
- Learners use observations to compute gradients
Parallel Exploration

- Multiple actors running in parallel
- Divergence => Divergence
- Exploring different parts of environment decorrelates observations
- Can also use diff. Exploration policies for each learner
- Can avoid instability due to data correlation w/o using replay buffer!
  - Allows us to use on-policy methods
- Almost-linear reduction in training time w/ more actor-learners
Asynchronous RL Algorithms
Asynchronous 1-step Q Learning

- Each thread interacts with a copy of the environment
- Computes the gradient of the Q-loss
- Problem: Actor-learners may overwrite!
  - Fix: Accumulate updates over several steps
- Optimization: Separate explorations

Asynchronous Sarsa

- Use $r + \gamma Q(s', a'; \theta^-)$ as the target with $a', s'$

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

```plaintext
// Assume global shared $\theta, \theta^-$, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$
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 $T > T_{max}$
```
Asynchronous n-step Q Learning

- To compute one update, the algo:
  - Selects $n \leq \frac{t_{\text{max}}}{t_{\text{max}}} \text{ terminal actions}$
  - Receives $n \leq t_{\text{max}} \text{ rewards}$
  - Computes gradients for each s/a pair
  - Each n-step update has n updates
  - Accumulated updates applied at once

Algorithm S2: Asynchronous n-step Q-learning - pseudocode for each actor

```
// Assume global shared parameter vector $\theta$.
// Assume global shared target parameter vector $\theta^{-}$.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^{-} \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' \leftarrow \theta$
  $\text{t}_{\text{start}} = t$
  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 $t - \text{t}_{\text{start}} = t_{\text{max}}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^{-}) & \text{for non-terminal } s_t \end{cases}$
  for $i \in \{t - 1, \ldots, t_{\text{start}}\}$ do
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta' \leftarrow d\theta' + \frac{\partial (R - Q(s_{t+1}; a_{t+1}; \theta'))}{\partial \theta'}$
  end for
  Perform asynchronous update of $\theta$ using $d\theta$.
  if $T \mod t_{\text{target}} == 0$ then
    $\theta^{-} \leftarrow \theta$
  end if
until $T > T_{\text{max}}$
```
A3C: Async. Advantage Actor-Critic
A3C (simplified)

- Each worker regularly syncs weights
- Collects sample from env
- Computes grad
- Async. Sends grad to global net
A3C

- Maintain policy
- Maintain value estimate
- Update both after t_max actions
- Grad of log policy
  - Scaled by advantage
- Advantage: diff between future and current value functions

**Algorithm S3** Asynchronous advantage actor-critic - pseudocode for each actor-

```
// 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 \( \pi(a_t|s_t; \theta') \)
    Perform 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 \( \theta' \): \( d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_t|s_i; \theta') \left( R - V(s_i; \theta_v') \right) \)
        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 \( T > T_{max} \)
A3C

- Share params b/t policy and value
- Policy: CNN (shared) w/softmax
- Value: CNN (shared) w/linear
- Entropy regularization
- “Critic” is value-function baseline
- Reduces variance
- Unbiased estimator
Key Metrics and 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>

Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table S3 shows the raw scores for all games.
<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>

*Table 2.* The average training speedup for each method and number of threads averaged over seven Atari games. To compute the training speed-up on a single game we measured the time to required reach a fixed reference score using each method and number of threads. The speedup from using \( n \) threads on a game was defined as the time required to reach a fixed reference score using one thread divided the time required to reach the reference score using \( n \) threads. The table shows the speedups averaged over seven Atari games (Beamrider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders).
Limitations and Conclusions
Limitations and Improvements

- A3C doesn’t scale and can’t take advantage of prioritization
  - Ape-X uses priorities from Prioritized-DQN adapted for distributed setting
- Asynchrony => actors working with outdated models
  - IMPALA further improves w importance weighting (fixes policy lag)
- Actors working w diff models => aggregated update is schizophrenic
  - Fixed in A2C by removing asynchrony -turns out the benefit outweighs the costs, A2C>A3C
- Authors themselves admit they should try to use replay
  - Ape-X reintroduces replay buffer
- Too many small changes => instability
- Unsurprisingly, too many large changes => instability
  - Some “bells and whistles” can help
Conclusions

- A3C made Dist. Deep RL possible w/o relying on replay buffer
- "Model-agnostic" DDRL
- Good performance without bells and whistles made it widely used
- Though suffering from problems, still a major step forward in DDRL

DQN -> Gorilla -> A3C -> A2C/Ape-X -> IMPALA -> ?