Policy Gradient Methods

Milan Straka

November 25, 2019

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Policy Gradient Methods

Instead of predicting expected returns, we could train the method to directly predict the policy

$$\pi(a|s;\boldsymbol\theta).$$

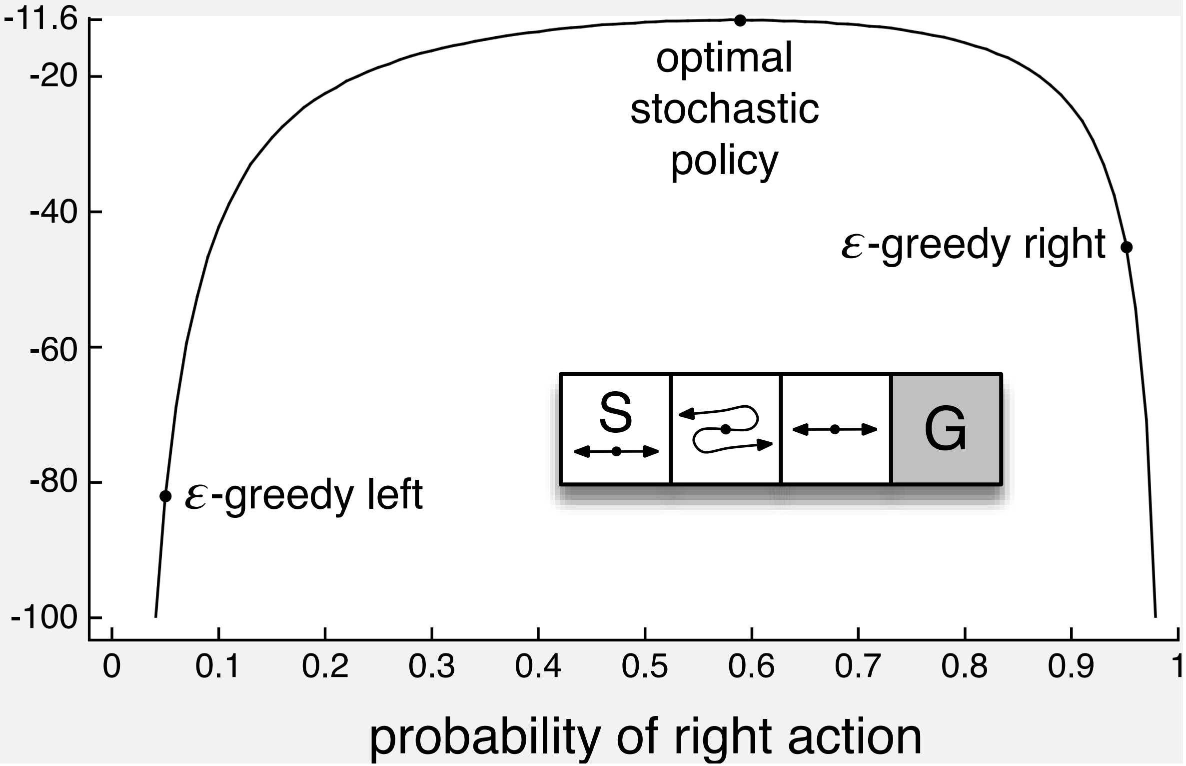
Obtaining the full distribution over all actions would also allow us to sample the actions according to the distribution π instead of just ε-greedy sampling.

However, to train the network, we maximize the expected return $v_\pi(s)$ and to that account we need to compute its *gradient* $\nabla_{\boldsymbol\theta} v_\pi(s)$.

Policy Gradient Methods

In addition to discarding ε-greedy action selection, policy gradient methods allow producing policies which are by nature stochastic, as in card games with imperfect information, while the action-value methods have no natural way of finding stochastic policies (distributional RL might be of some use though).

$$J(\boldsymbol\theta) = v_{\pi_{\boldsymbol\theta}}(\mathsf{S})$$



Example 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

Policy Gradient Theorem

Let $\pi(a|s;\boldsymbol\theta)$ be a parametrized policy. We denote the initial state distribution as $h(s)$ and the on-policy distribution under π as $\mu(s)$. Let also $J(\boldsymbol\theta) \stackrel{\text{def}}{=} \mathbb{E}_{h,\pi} v_\pi(s)$.

Then

$$\nabla_{\boldsymbol\theta} v_\pi(s) \propto \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta)$$

and

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta),$$

where $P(s \rightarrow \ldots \rightarrow s'|\pi)$ is probability of transitioning from state s to s' using 0, 1, … steps.

Proof of Policy Gradient Theorem

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \Big[\sum\nolimits_a \pi(a|s;\boldsymbol\theta) q_\pi(s,a)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \nabla q_\pi(s,a)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \nabla \big(\sum\nolimits_{s'} p(s'|s,a)(r + v_\pi(s'))\big)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \big(\sum\nolimits_{s'} p(s'|s,a) \nabla v_\pi(s')\big)\Big]
\end{aligned}$$

We now expand $v_\pi(s')$.

$$\begin{aligned}
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \Big(\sum\nolimits_{s'} p(s'|s,a) \Big(\\
&\qquad \sum\nolimits_{a'} \Big[\nabla \pi(a'|s';\boldsymbol\theta) q_\pi(s',a') + \pi(a'|s';\boldsymbol\theta) \Big(\sum\nolimits_{s''} p(s''|s',a') \nabla v_\pi(s'')\Big)\Big)\Big)\Big]
\end{aligned}$$

Continuing to expand all $v_\pi(s'')$, *we obtain the following:*

$$\nabla v_\pi(s) = \sum_{s' \in \mathcal{S}} \sum_{k=0}^{\infty} P(s \rightarrow s' \text{ in } k \text{ steps } |\pi) \sum_{a \in \mathcal{A}} q_\pi(s',a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta).$$

Proof of Policy Gradient Theorem

To finish the proof of the first part, it is enough to realize that

$$\sum_{k=0}^{\infty} P(s \rightarrow s' \text{ in } k \text{ steps } |\pi) \propto P(s \rightarrow \ldots \rightarrow s'|\pi).$$

For the second part, we know that

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) = \mathbb{E}_{s\sim h} \nabla_{\boldsymbol\theta} v_\pi(s) \propto \mathbb{E}_{s\sim h} \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s'; \boldsymbol\theta),$$

therefore using the fact that $\mu(s') = \mathbb{E}_{s\sim h} \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi)$ we get

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a|s; \boldsymbol\theta).$$

REINFORCE Algorithm

The REINFORCE algorithm (Williams, 1992) uses directly the policy gradient theorem, minimizing $-J(\boldsymbol\theta) \stackrel{\text{def}}{=} -\mathbb{E}_{h,\pi} v_\pi(s)$. The loss gradient is then

$$\nabla_{\boldsymbol\theta} - J(\boldsymbol\theta) \propto -\sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = -\mathbb{E}_{s\sim\mu} \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

However, the sum over all actions is problematic. Instead, we rewrite it to an expectation which we can estimate by sampling:

$$\nabla_{\boldsymbol\theta} - J(\boldsymbol\theta) \propto \mathbb{E}_{s\sim\mu} \mathbb{E}_{a\sim\pi} q_\pi(s,a) \nabla_{\boldsymbol\theta} - \ln \pi(a|s;\boldsymbol\theta),$$

where we used the fact that

$$\nabla_{\boldsymbol\theta} \ln \pi(a|s;\boldsymbol\theta) = \frac{1}{\pi(a|s;\boldsymbol\theta)} \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

REINFORCE Algorithm

REINFORCE therefore minimizes the loss

$$\mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi} q_\pi(s,a) \nabla_{\boldsymbol\theta} - \ln \pi(a|s;\boldsymbol\theta),$$

estimating the $q_\pi(s,a)$ by a single sample.

Note that the loss is just a weighted variant of negative log likelihood (NLL), where the sampled actions play a role of gold labels and are weighted according to their return.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol\theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol\theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha G \nabla \ln \pi(A_t|S_t,\boldsymbol\theta)$

Modified from Algorithm 13.3 of "Reinforcement Learning: An Introduction, Second Edition" by removing γ^t from the update of θ.

REINFORCE with Baseline

The returns can be arbitrary – better-than-average and worse-than-average returns cannot be recognized from the absolute value of the return.

Hopefully, we can generalize the policy gradient theorem using a baseline $b(s)$ to

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} \big(q_\pi(s, a) - b(s)\big) \nabla_{\boldsymbol\theta} \pi(a|s; \boldsymbol\theta).$$

The baseline $b(s)$ can be a function or even a random variable, as long as it does not depend on a, because

$$\sum_a b(s) \nabla_{\boldsymbol\theta} \pi(a|s; \boldsymbol\theta) = b(s) \sum_a \nabla_{\boldsymbol\theta} \pi(a|s; \boldsymbol\theta) = b(s) \nabla 1 = 0.$$

REINFORCE with Baseline

A good choice for $b(s)$ is $v_\pi(s)$, which can be shown to minimize variance of the estimator. Such baseline reminds centering of returns, given that

$$v_\pi(s) = \mathbb{E}_{a\sim\pi} q_\pi(s, a).$$

Then, better-than-average returns are positive and worse-than-average returns are negative.

The resulting $q_\pi(s, a) - v_\pi(s)$ function is also called an *advantage function*

$$a_\pi(s, a) \stackrel{\text{def}}{=} q_\pi(s, a) - v_\pi(s).$$

Of course, the $v_\pi(s)$ baseline can be only approximated. If neural networks are used to estimate $\pi(a|s; \boldsymbol\theta)$, then some part of the network is usually shared between the policy and value function estimation, which is trained using mean square error of the predicted and observed return.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat v(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol\theta)$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
        $\delta \leftarrow G - \hat v(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat v(S_t,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol\theta)$

Modified from Algorithm 13.4 of "Reinforcement Learning: An Introduction, Second Edition" by removing γ^t from the update of θ.

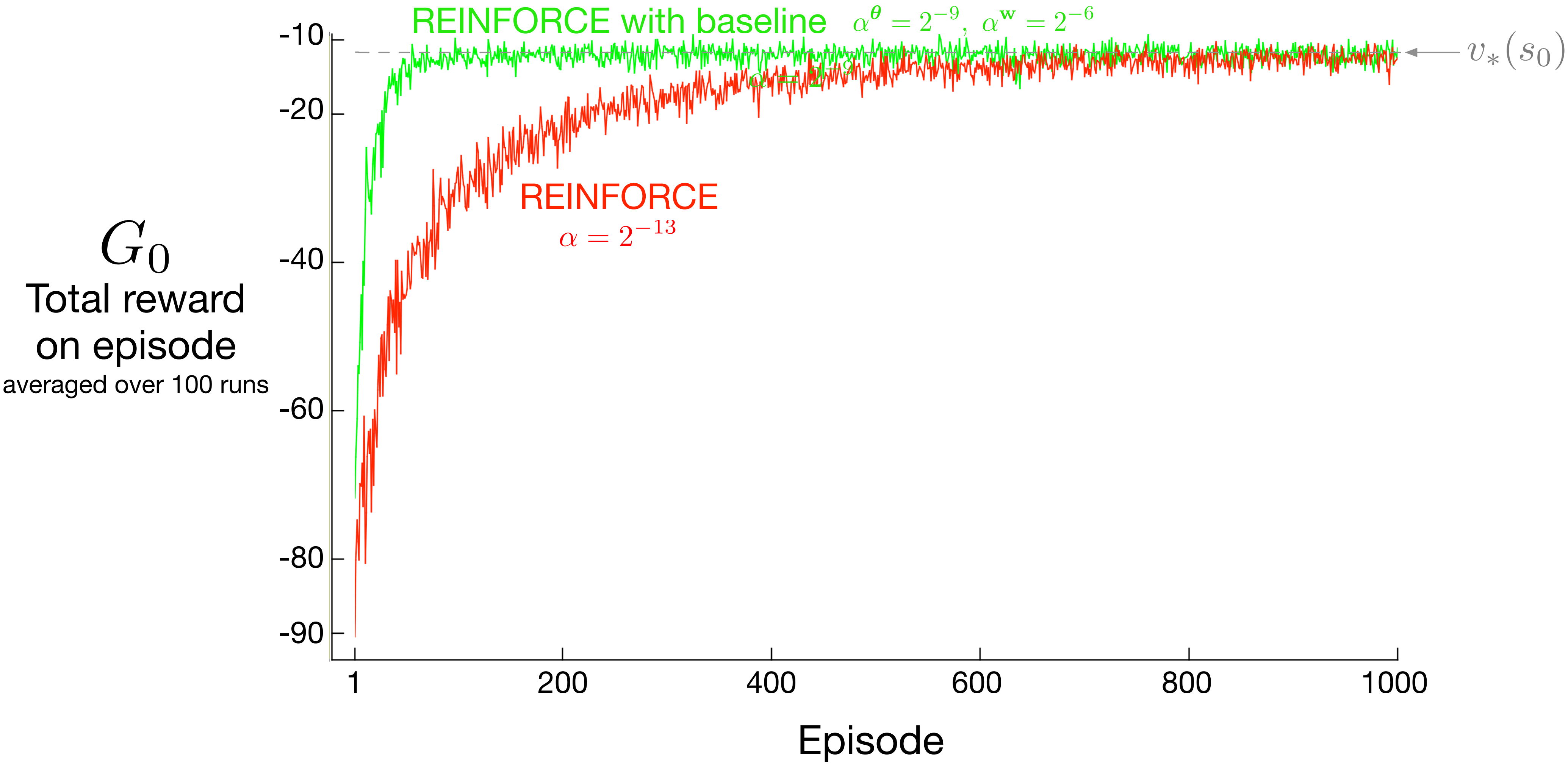


Figure 13.2 of "Reinforcement Learning: An Introduction, Second Edition".

Actor-Critic

It is possible to combine the policy gradient methods and temporal difference methods, creating a family of algorithms usually called *actor-critic* methods.

The idea is straightforward – instead of estimating the episode return using the whole episode rewards, we can use n-step temporal difference estimation.

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat v(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize S (first state of episode)

    Loop while S is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol\theta)$
        Take action A, observe S', R
        $\delta \leftarrow R + \gamma \hat v(S',\mathbf{w}) - \hat v(S,\mathbf{w})$ (if S' is terminal, then $\hat v(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat v(S,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A|S, \boldsymbol\theta)$
        $S \leftarrow S'$

Modified from Algorithm 13.5 of "Reinforcement Learning: An Introduction, Second Edition" by removing I.

Asynchronous Methods for Deep RL

A 2015 paper from Volodymyr Mnih et al., the same group as DQN.

The authors propose an asynchronous framework, where multiple workers share one neural network, each training using either an off-line or on-line RL algorithm.

They compare 1-step Q-learning, 1-step Sarsa, n-step Q-learning and A3C (an *asynchronous advantage actor-critic* method). For A3C, they compare a version with and without LSTM.

The authors also introduce *entropy regularization term* $\beta H(\pi(s;\boldsymbol\theta))$ to the loss to support exploration and discourage premature convergence.

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 = { r                            for terminal s'
        { r + γ max_a' Q(s', a'; θ⁻)   for non-terminal s'
    Accumulate gradients wrt θ: dθ ← dθ + ∂(y − Q(s,a;θ))²/∂θ
    s = s'
    T ← T + 1 and t ← t + 1
    if T mod I_target == 0 then
        Update the target network θ⁻ ← θ
    end if
    if t mod I_AsyncUpdate == 0 or s is terminal then
        Perform asynchronous update of θ using dθ.
        Clear gradients dθ ← 0.
    end if
until T > T_max
```

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

$$d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$$

Algorithm 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Algorithm S2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ⁻.
// Assume global shared counter T = 0.
Initialize thread step counter t ← 1
Initialize target network parameters θ⁻ ← θ
Initialize thread-specific parameters θ' = θ
Initialize network gradients dθ ← 0
repeat
    Clear gradients dθ ← 0
    Synchronize thread-specific parameters θ' = θ
    t_start = t
    Get state s_t
    repeat
        Take action a_t according to the ε-greedy policy based on Q(s_t, a; θ')
        Receive reward r_t and new state s_{t+1}
        t ← t + 1
        T ← T + 1
    until terminal s_t or t − t_start == t_max
    R = { 0                    for terminal s_t
        { max_a Q(s_t, a; θ⁻)  for non-terminal s_t
    for i ∈ {t − 1, ..., t_start} do
        R ← r_i + γR
        Accumulate gradients wrt θ': dθ ← dθ + ∂(R − Q(s_i, a_i; θ'))² / ∂θ'
    end for
    Perform asynchronous update of θ using dθ.
    if T mod I_target == 0 then
        θ⁻ ← θ
    end if
until T > T_max
```

Algorithm S2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Asynchronous Methods for Deep RL

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_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** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Algorithm S3 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

All methods performed updates every 5 actions ($t_{\max} = I_{\mathrm{AsyncUpdate}} = 5$), updating the target network each $40\,000$ frames.

The Atari inputs were processed as in DQN, using also action repeat 4.

The network architecture is: 16 filters 8×8 stride 4, 32 filters 4×4 stride 2, followed by a fully connected layer with 256 units. All hidden layers apply a ReLU non-linearity. Values and/or action values were then generated from the (same) last hidden layer.

The LSTM methods utilized a 256-unit LSTM cell after the dense hidden layer.

All experiments used a discount factor of $\gamma = 0.99$ and used RMSProp with momentum decay factor of 0.99.

Asynchronous Methods for Deep RL

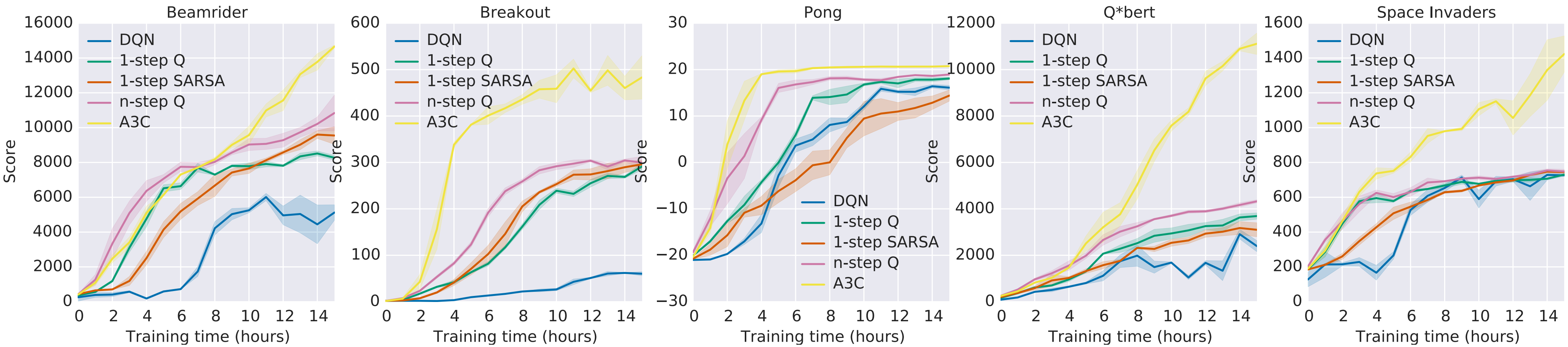


Figure 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

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

Table 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

	Number of threads				
Method	1	2	4	8	16
1-step Q	1.0	**3.0**	**6.3**	**13.3**	**24.1**
1-step SARSA	1.0	**2.8**	**5.9**	**13.1**	**22.1**
n-step Q	1.0	**2.7**	**5.9**	**10.7**	**17.2**
A3C	1.0	2.1	3.7	6.9	12.5

Table 2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Asynchronous Methods for Deep RL

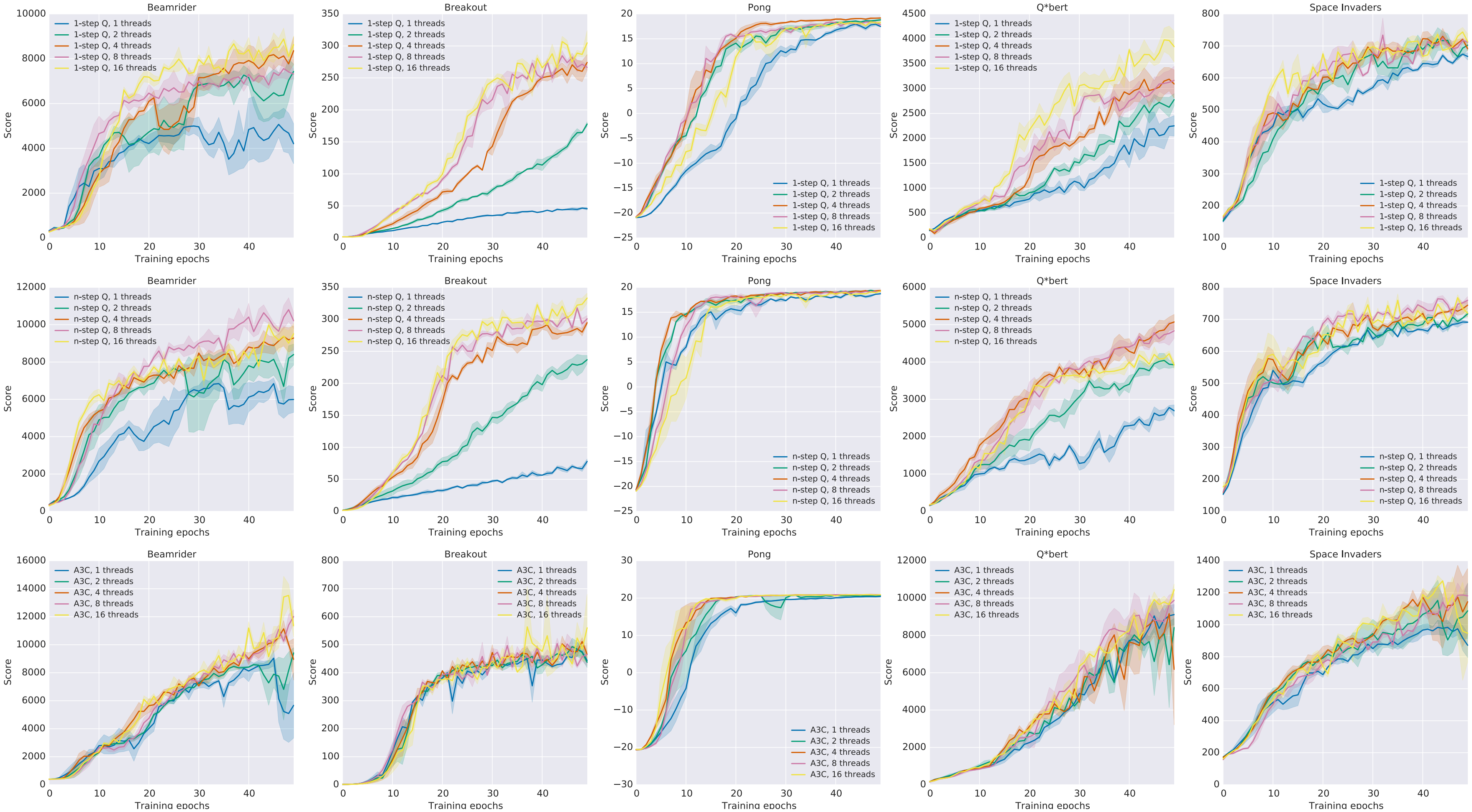


Figure 3 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

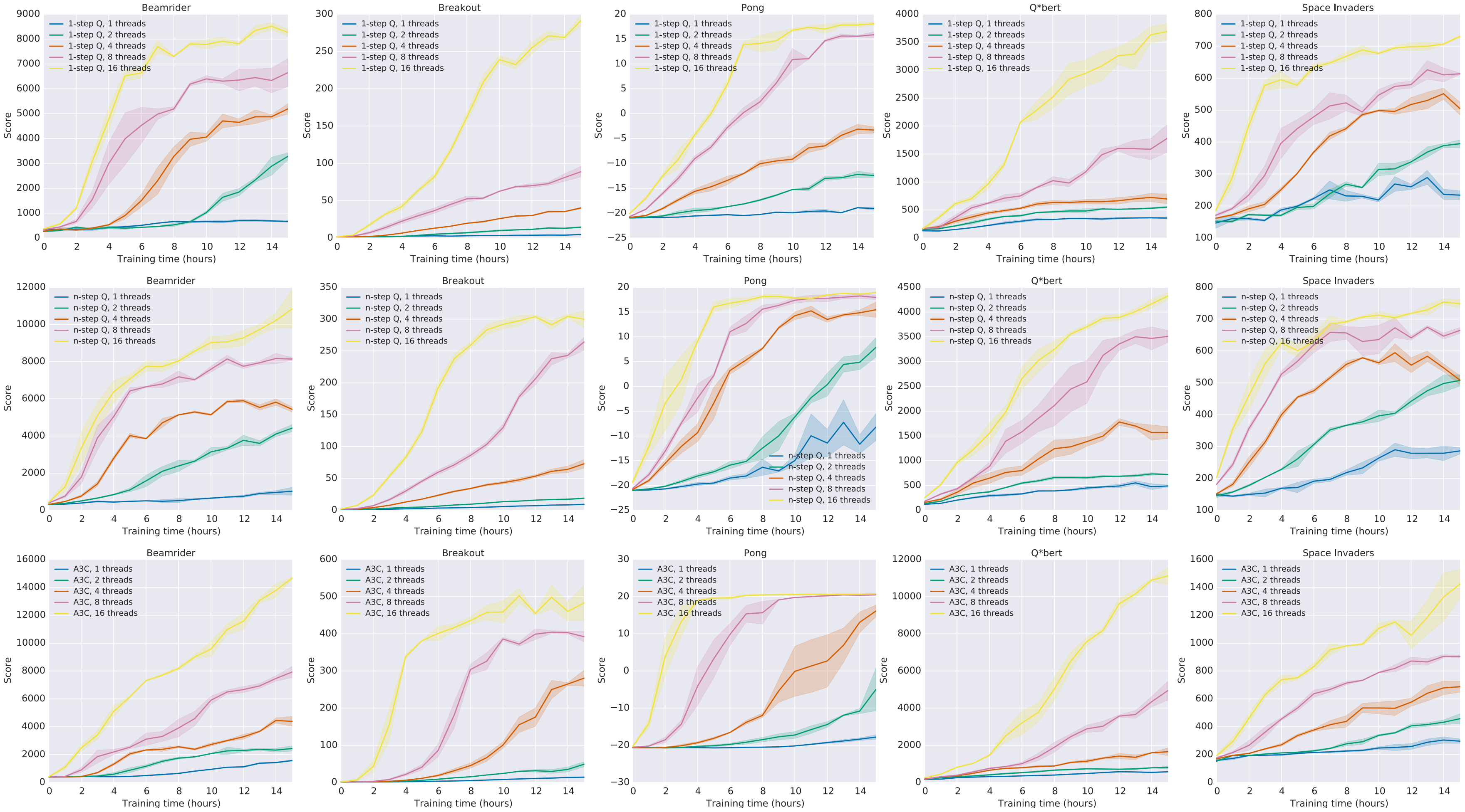


Figure 4 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Asynchronous Methods for Deep RL

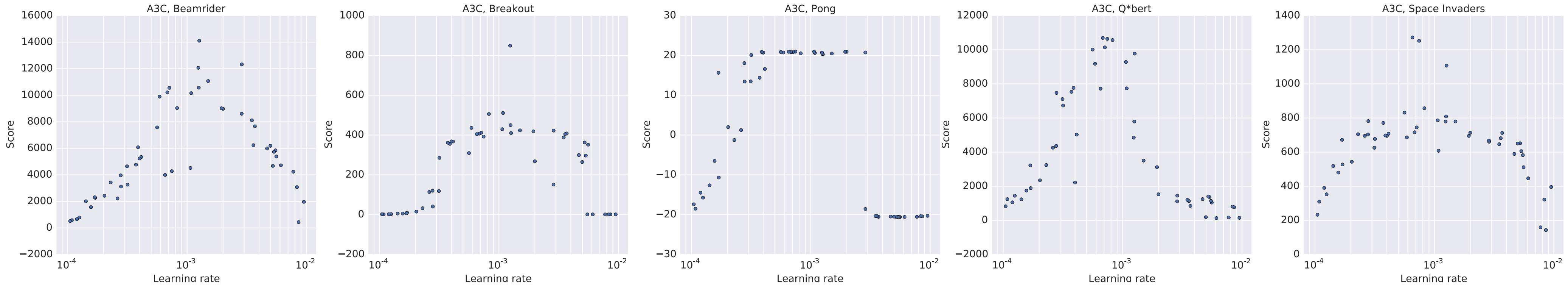


Figure 2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.