Advantage Actor-Critic, Continuous Action Space

Milan Straka

December 3, 2018

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

REINFORCE Algorithm

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

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

REINFORCE algorithm estimates the $q_\pi(s,a)$ by a single sample.

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 $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha G \nabla \ln \pi(A_t|S_t,\boldsymbol\theta)$
```

Modification of Algorithm 13.3 of "Reinforcement Learning: An Introduction, Second Edition".

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

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

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 value is also called an *advantage function* $a_\pi(s, a) \overset{\text{def}}{=} q_\pi(s, a) - v_\pi(s)$.

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

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 $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$                    $(G_t)$
        $\delta \leftarrow G - \hat v(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat v(S_t,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A_t|S_t,\boldsymbol\theta)$
```

Modification of Algorithm 13.4 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 $\hat v(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat v(S,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A|S, \boldsymbol\theta)$
        $S \leftarrow S'$
```

Modification of Algorithm 13.5 of "Reinforcement Learning: An Introduction, Second Edition".

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.

Asynchronous Methods for Deep RL

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

```
// 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 $T > T_{max}$
```

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

Asynchronous Methods for Deep RL

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 $T = 0$.
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 $\theta' = \theta$
    $t_{start} = t$
    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
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \frac{\partial \left(R - Q(s_i, a_i; \theta')\right)^2}{\partial \theta'}$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$.
    if $T \mod I_{target} == 0$ then
        $\theta^- \leftarrow \theta$
    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 $\theta$ 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 $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

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

Asynchronous Methods for Deep RL

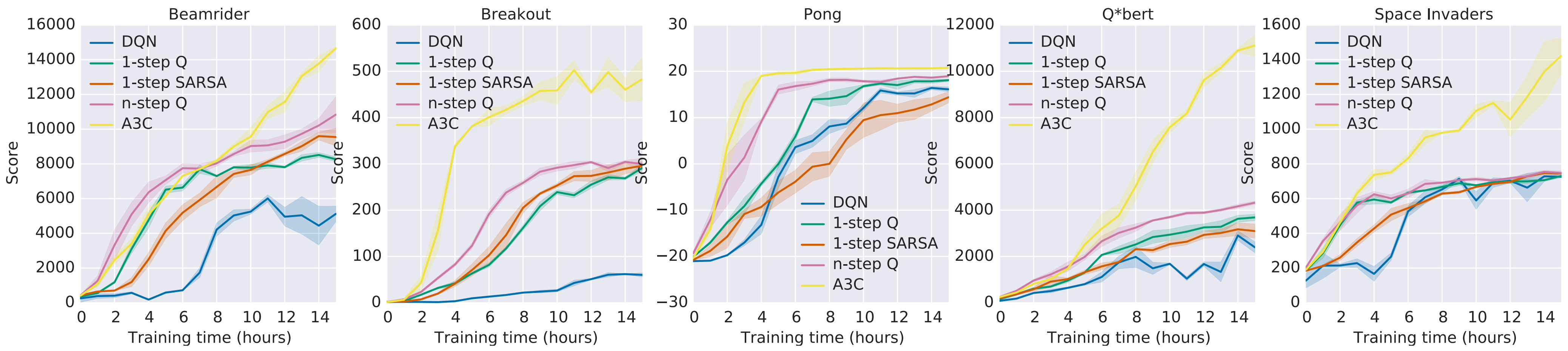


Figure 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

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

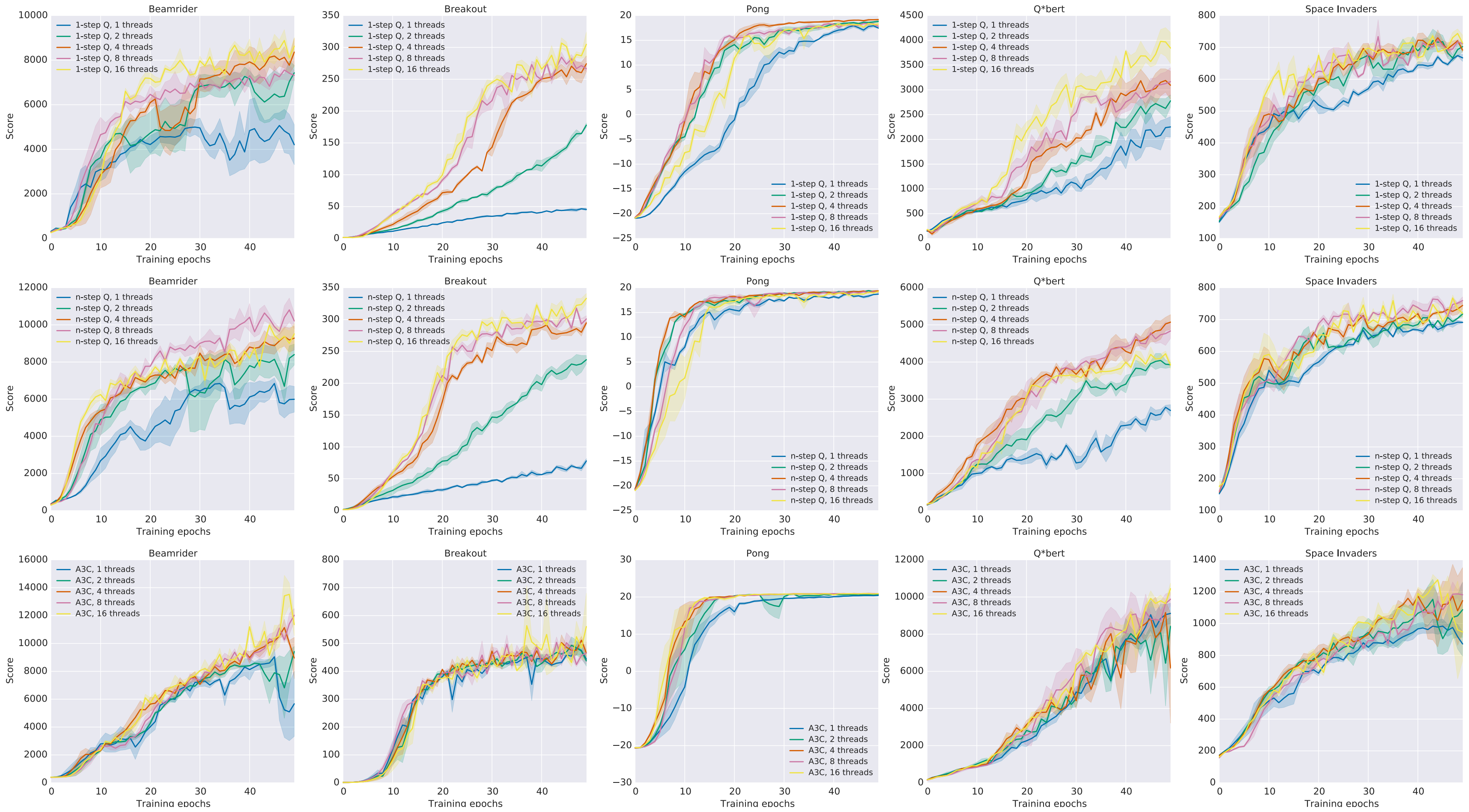


Figure 3 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Asynchronous Methods for Deep RL

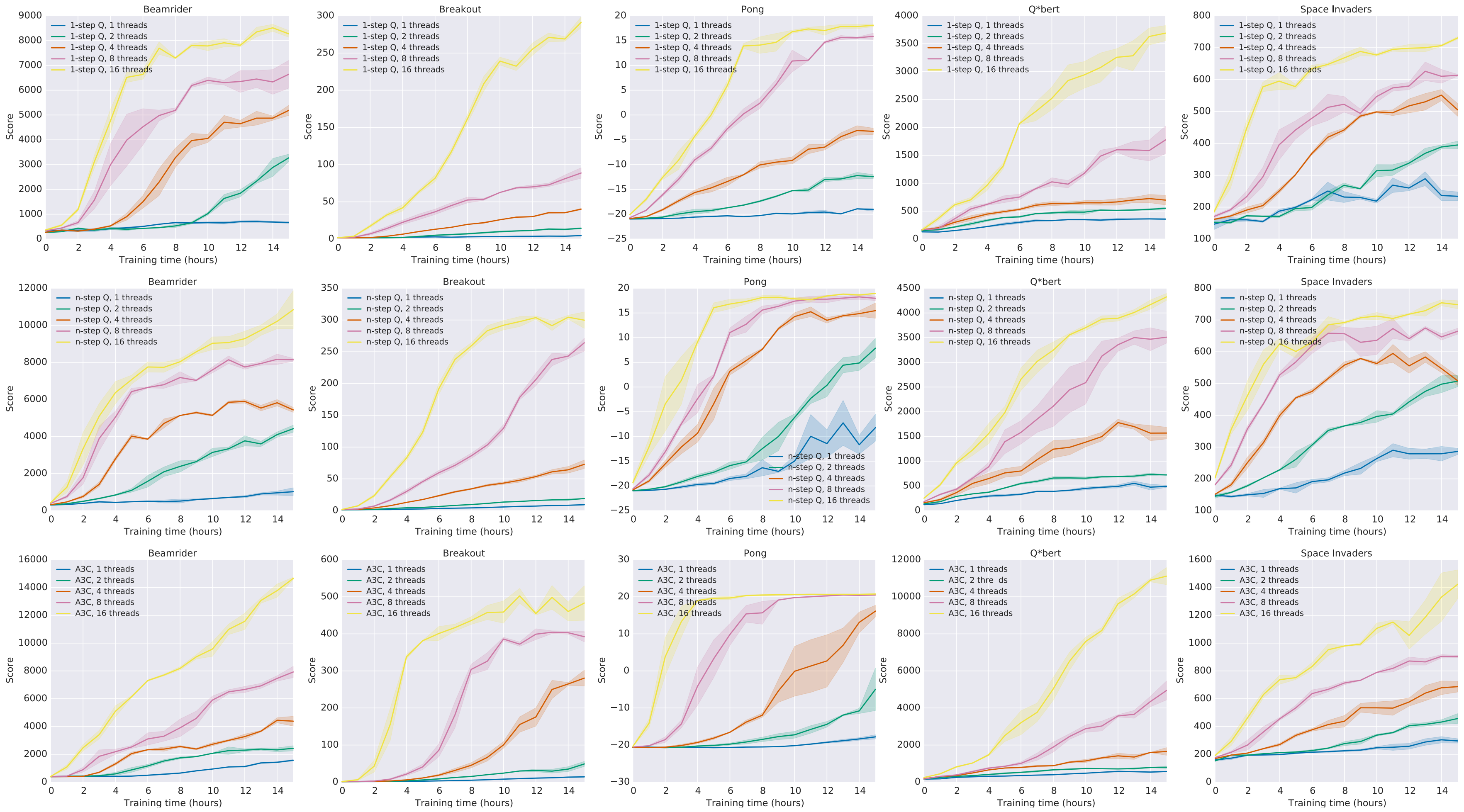


Figure 4 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Asynchronous Methods for Deep RL

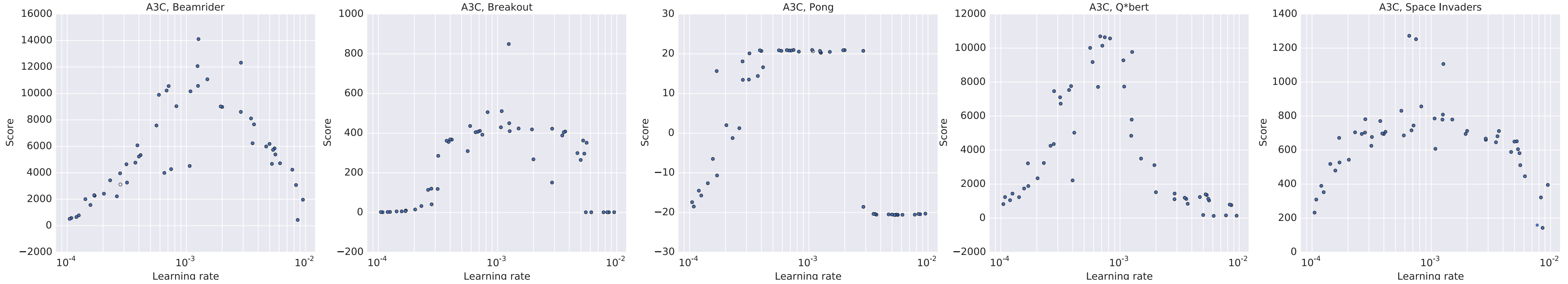


Figure 2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

Parallel Advantage Actor Critic

An alternative to independent workers is to train in a synchronous and centralized way by having the workes to only generate episodes. Such approach was described in May 2017 by Clemente et al., who named their agent *parallel advantage actor-critic* (PAAC).

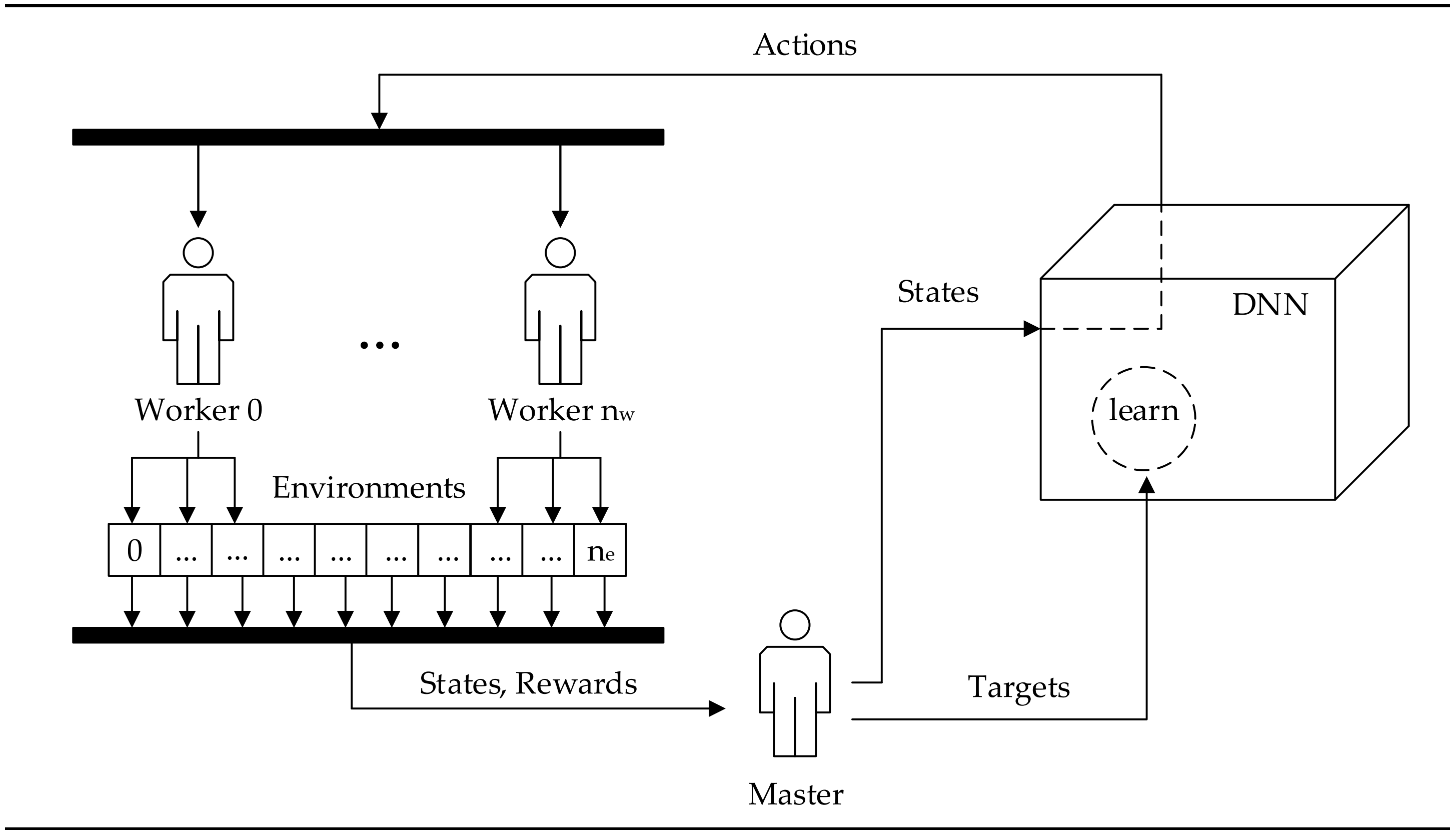


Figure 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic

Algorithm 1 Parallel advantage actor-critic

```
1:  Initialize timestep counter $N = 0$ and network weights $\theta$, $\theta_v$
2:  Instantiate set $\boldsymbol{e}$ of $n_e$ environments
3:  repeat
4:      for $t = 1$ to $t_{max}$ do
5:          Sample $\boldsymbol{a}_t$ from $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t;\theta)$
6:          Calculate $\boldsymbol{v}_t$ from $V(\boldsymbol{s}_t;\theta_v)$
7:          parallel for $i = 1$ to $n_e$ do
8:              Perform action $a_{t,i}$ in environment $e_i$
9:              Observe new state $s_{t+1,i}$ and reward $r_{t+1,i}$
10:         end parallel for
11:     end for
12:     $\boldsymbol{R}_{t_{\max}+1} = \begin{cases} 0 & \text{for terminal } \boldsymbol{s}_t \\ V(s_{t_{\max}+1};\theta) & \text{for non-terminal } \boldsymbol{s}_t \end{cases}$
13:     for $t = t_{\max}$ down to 1 do
14:         $\boldsymbol{R}_t = \boldsymbol{r}_t + \gamma \boldsymbol{R}_{t+1}$
15:     end for
16:     $d\theta = \frac{1}{n_e \cdot t_{max}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{max}} (R_{t,i} - v_{t,i}) \nabla_\theta \log \pi(a_{t,i}|s_{t,i};\theta) + \beta \nabla_\theta H(\pi(s_{e,t};\theta))$
17:     $d\theta_v = \frac{1}{n_e \cdot t_{max}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{max}} \nabla_{\theta_v} (R_{t,i} - V(s_{t,i};\theta_v))^2$
18:     Update $\theta$ using $d\theta$ and $\theta_v$ using $d\theta_v$.
19:     $N \leftarrow N + n_e \cdot t_{\max}$
20: until $N \geq N_{max}$
```

Algorithm 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic

Game	Gorila	A3C FF	GA3C	PAAC arch_nips	PAAC $\text{arch}_\text{nature}$
Amidar	1189.70	263.9	218	701.8	1348.3
Centipede	8432.30	3755.8	7386	5747.32	7368.1
Beam Rider	3302.9	22707.9	N/A	4062.0	6844.0
Boxing	94.9	59.8	92	99.6	99.8
Breakout	402.2	681.9	N/A	470.1	565.3
Ms. Pacman	3233.50	653.7	1978	2194.7	1976.0
Name This Game	6182.16	10476.1	5643	9743.7	14068.0
Pong	18.3	5.6	18	20.6	20.9
Qbert	10815.6	15148.8	14966.0	16561.7	17249.2
Seaquest	13169.06	2355.4	1706	1754.0	1755.3
Space Invaders	1883.4	15730.5	N/A	1077.3	1427.8
Up n Down	12561.58	74705.7	8623	88105.3	100523.3
Training	4d CPU cluster	4d CPU	1d GPU	12h GPU	15h GPU

Table 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

The authors use 8 workers, $n_e = 32$ parallel environments, 5-step returns, $\gamma = 0.99$, $\varepsilon = 0.1$, $\beta = 0.01$ and a learning rate of $\alpha = 0.0007 \cdot n_e = 0.0224$.

The arch_nips is from A3C: 16 filters 8×8 stride 4, 32 filters 4×4 stride 2, a dense layer with 256 units. The $\text{arch}_\text{nature}$ is from DQN: 32 filters 8×8 stride 4, 64 filters 4×4 stride 2, 64 filters 3×3 stride 1 and 512-unit fully connected layer. All nonlinearities are ReLU.

Parallel Advantage Actor Critic

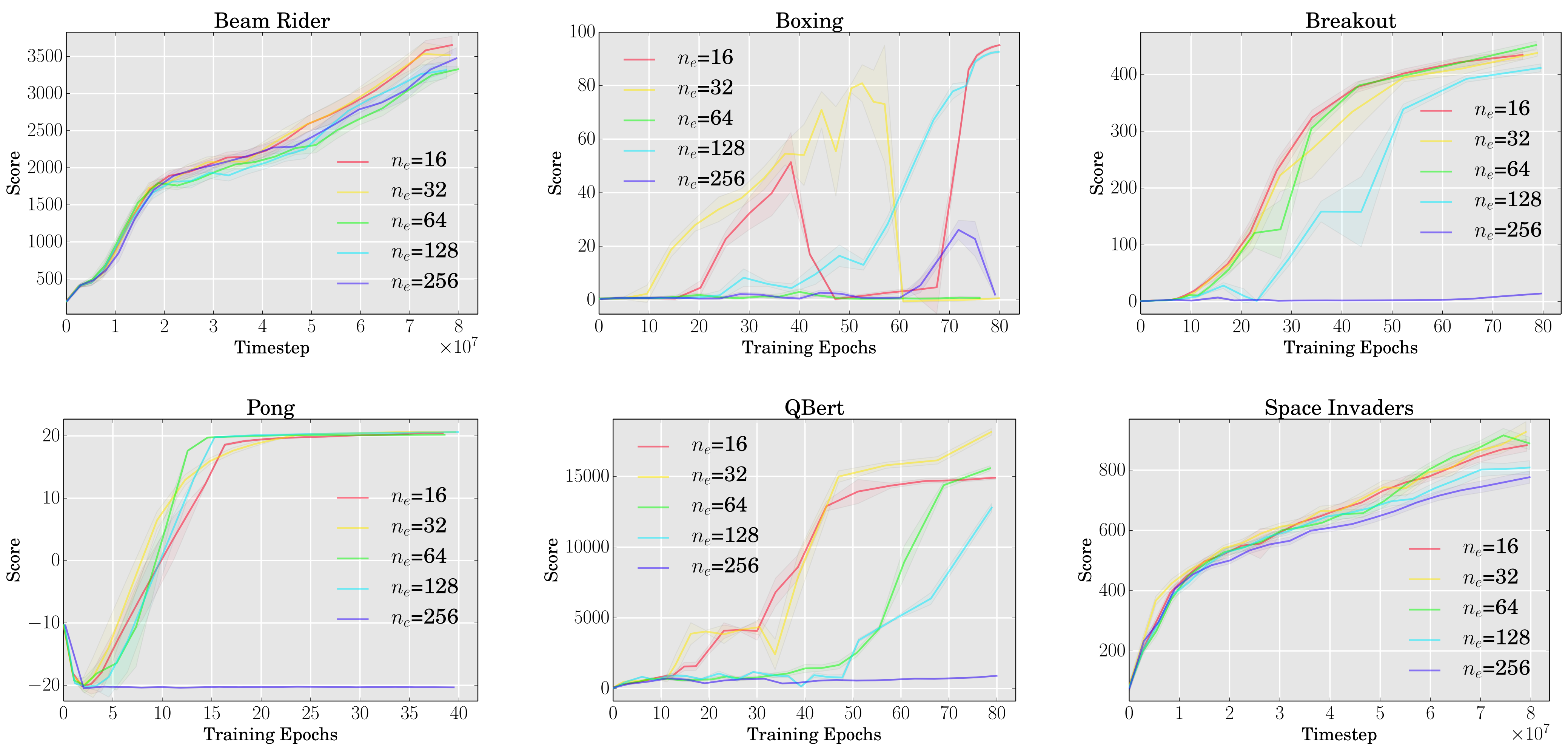


Figure 3 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic

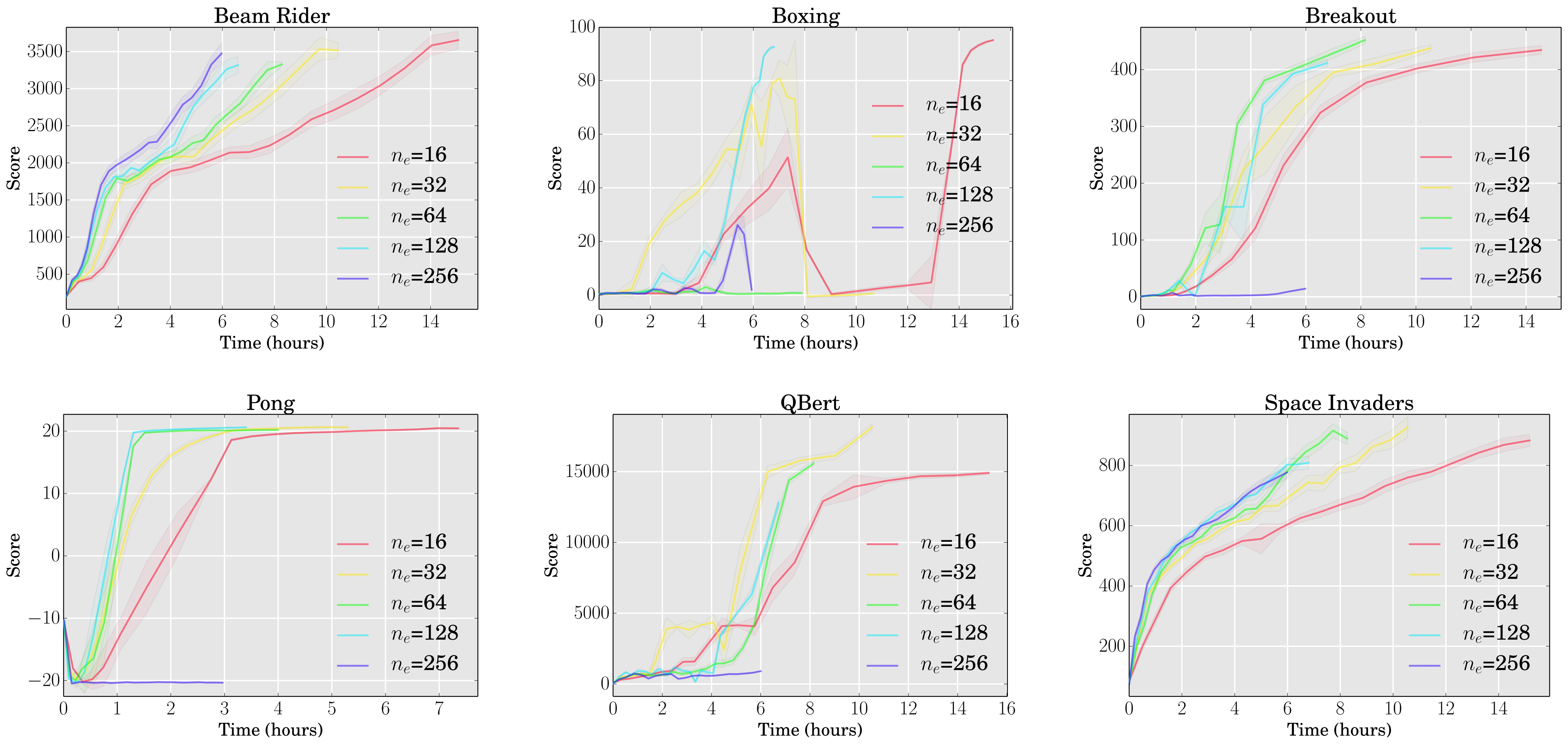
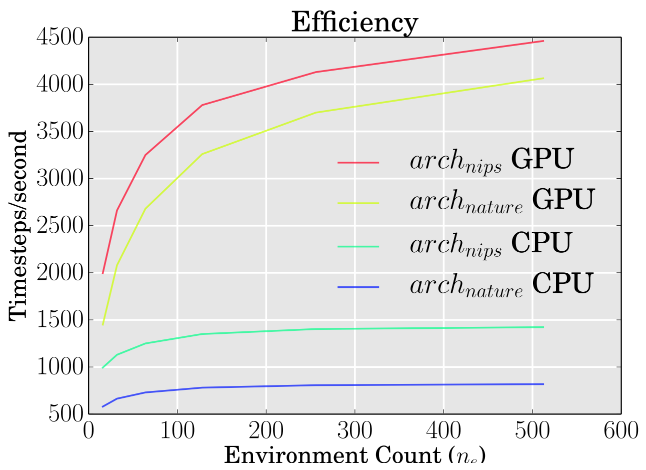
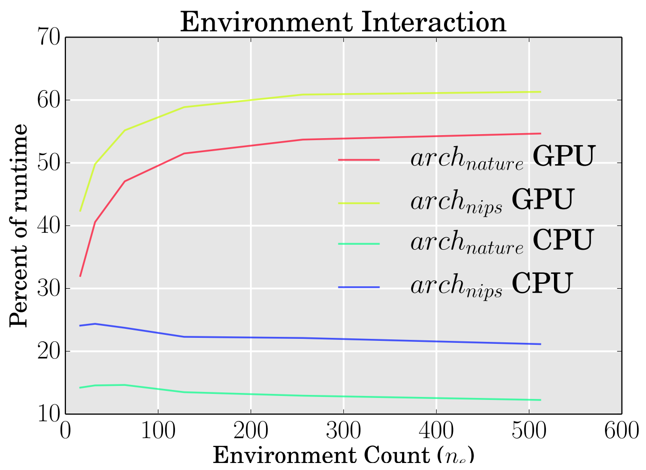


Figure 4 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic





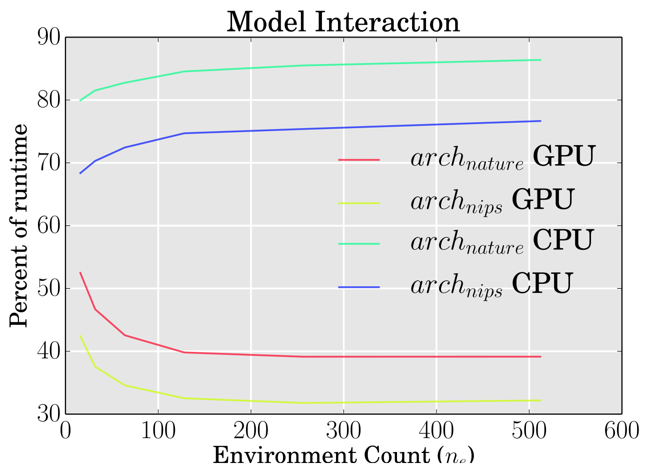


Figure 2 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Continuous Action Space

Until now, the actions were discrete. However, many environments naturally accept actions from continuous space. We now consider actions which come from range $[a, b]$ for $a, b \in \mathbb{R}$, or more generally from a Cartesian product of several such ranges:

$$\prod_i [a_i, b_i].$$

A simple way how to parametrize the action distribution is to choose them from the normal distribution.

Given mean μ and variance σ^2, probability density function of $\mathcal{N}(\mu, \sigma^2)$ is

$$p(x) \stackrel{\text{def}}{=} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

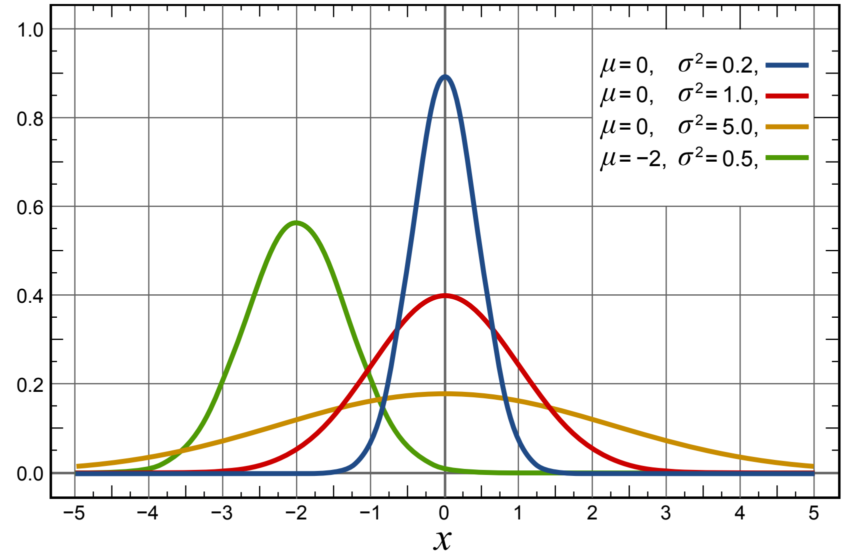


Figure from section 13.7 of "Reinforcement Learning: An Introduction, Second Edition".

Continuous Action Space in Gradient Methods

Utilizing continuous action spaces in gradient-based methods is straightforward. Instead of the $\operatorname{softmax}$ distribution we suitably parametrize the action value, usually using the normal distribution. Considering only one real-valued action, we therefore have

$$\pi(a|s;\boldsymbol\theta) \stackrel{\text{def}}{=} P\Big(a \sim \mathcal{N}\big(\mu(s;\boldsymbol\theta), \sigma(s;\boldsymbol\theta)^2\big)\Big),$$

where $\mu(s;\boldsymbol\theta)$ and $\sigma(s;\boldsymbol\theta)$ are function approximation of mean and standard deviation of the action distribution.

The mean and standard deviation are usually computed from the shared representation, with

- the mean being computed as a regular regression (i.e., one output neuron without activation);
- the standard variance (which must be positive) being computed again as a regression, followed most commonly by either $\exp$ or $\operatorname{softplus}$, where $\operatorname{softplus}(x) \stackrel{\text{def}}{=} \log(1+e^x)$.

Continuous Action Space in Gradient Methods

During training, we compute $\mu(s;\boldsymbol\theta)$ and $\sigma(s;\boldsymbol\theta)$ and then sample the action value (clipping it to $[a,b]$ if required). To compute the loss, we utilize the probability density function of the normal distribution (and usually also add the entropy penalty).

```
mu = tf.layers.dense(hidden_layer, 1)[:, 0]
sd = tf.layers.dense(hidden_layer, 1)[:, 0]
sd = tf.exp(log_sd)    # or sd = tf.nn.softplus(sd)

normal_dist = tf.distributions.Normal(mu, sd)

# Loss computed as - log π(a|s) - entropy_regularization
loss = - normal_dist.log_prob(self.actions) * self.returns \
       - args.entropy_regularization * normal_dist.entropy()
```

Continuous Action Space

When the action consists of several real values, i.e., action is a suitable subregion of $\mathbb{R}^n$ for $n > 1$, we can:

- either use multivariate Gaussian distribution;
- or factorize the probability into a product of univariate normal distributions.

Modeling the action distribution using a single normal distribution might be insufficient, in which case a mixture of normal distributions is usually used.

Sometimes, the continuous action space is used even for discrete output -- when modeling pixels intensities (256 values) or sound amplitude (2^{16} values), instead of a softmax we use discretized mixture of distributions, usually $\operatorname{logistic}$ (a distribution with a sigmoid cdf). Then,

$$\pi(a) = \sum_i p_i \Big(\sigma\big((a + 0.5 - \mu_i)/\sigma_i\big) - \sigma\big((a - 0.5 - \mu_i)/\sigma_i\big)\Big).$$

However, such mixtures are usually used in generative modeling, not in reinforcement learning.