Continuous Actions

Milan Straka

December 02, 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 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.

REINFORCE Algorithm

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

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

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

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.

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.

Actor-Critic

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.

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).

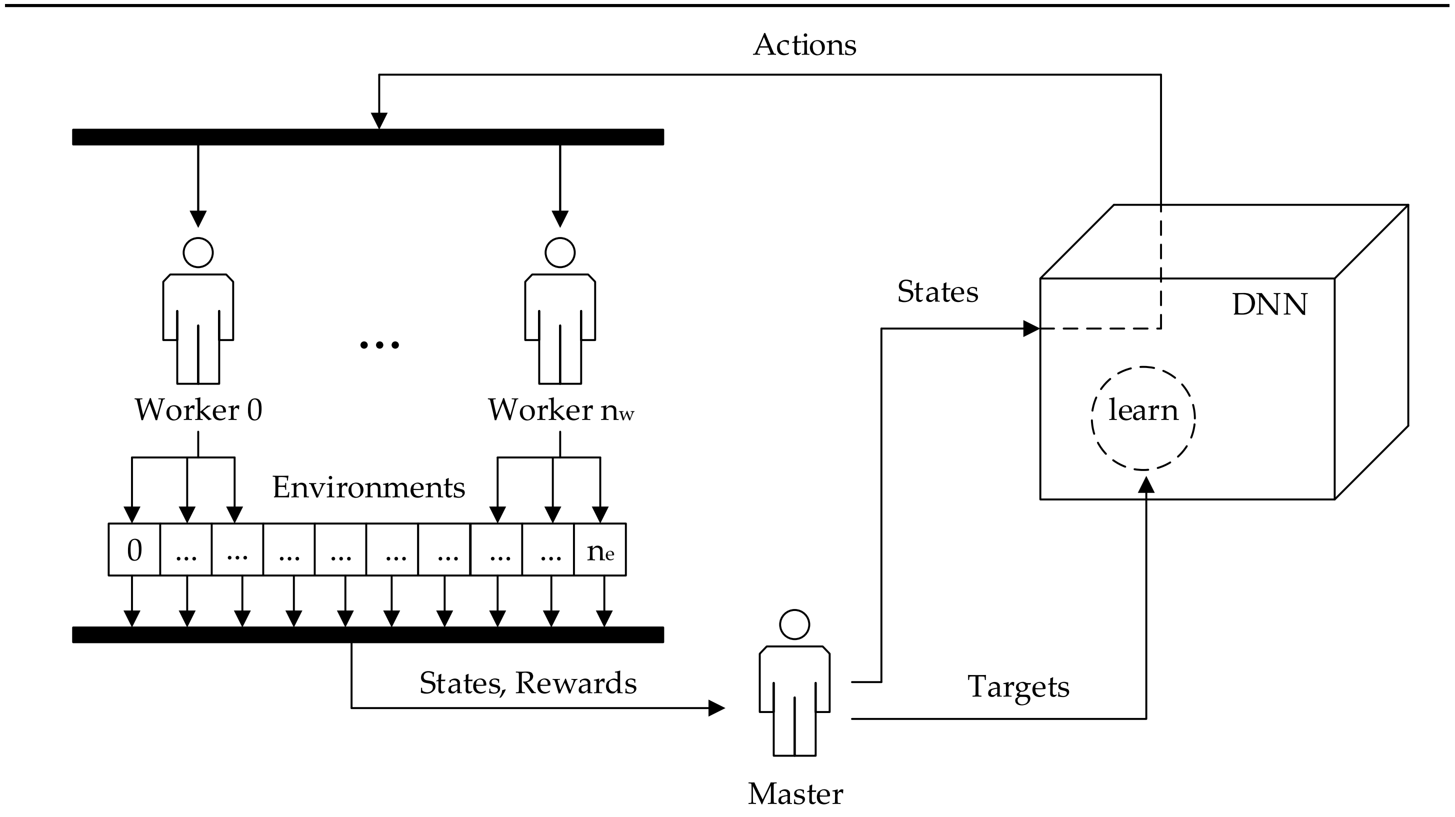


Figure 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

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

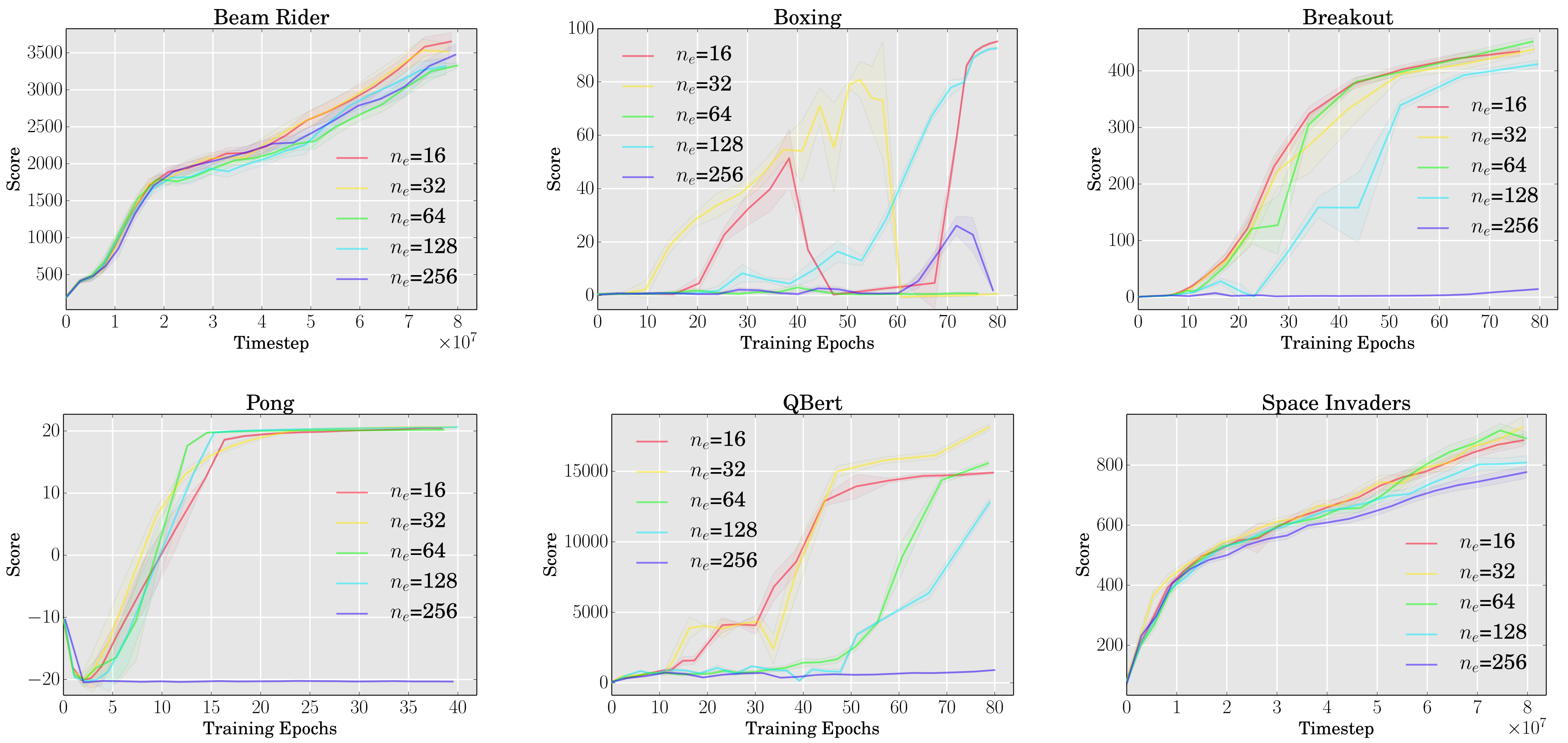


Figure 3 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic

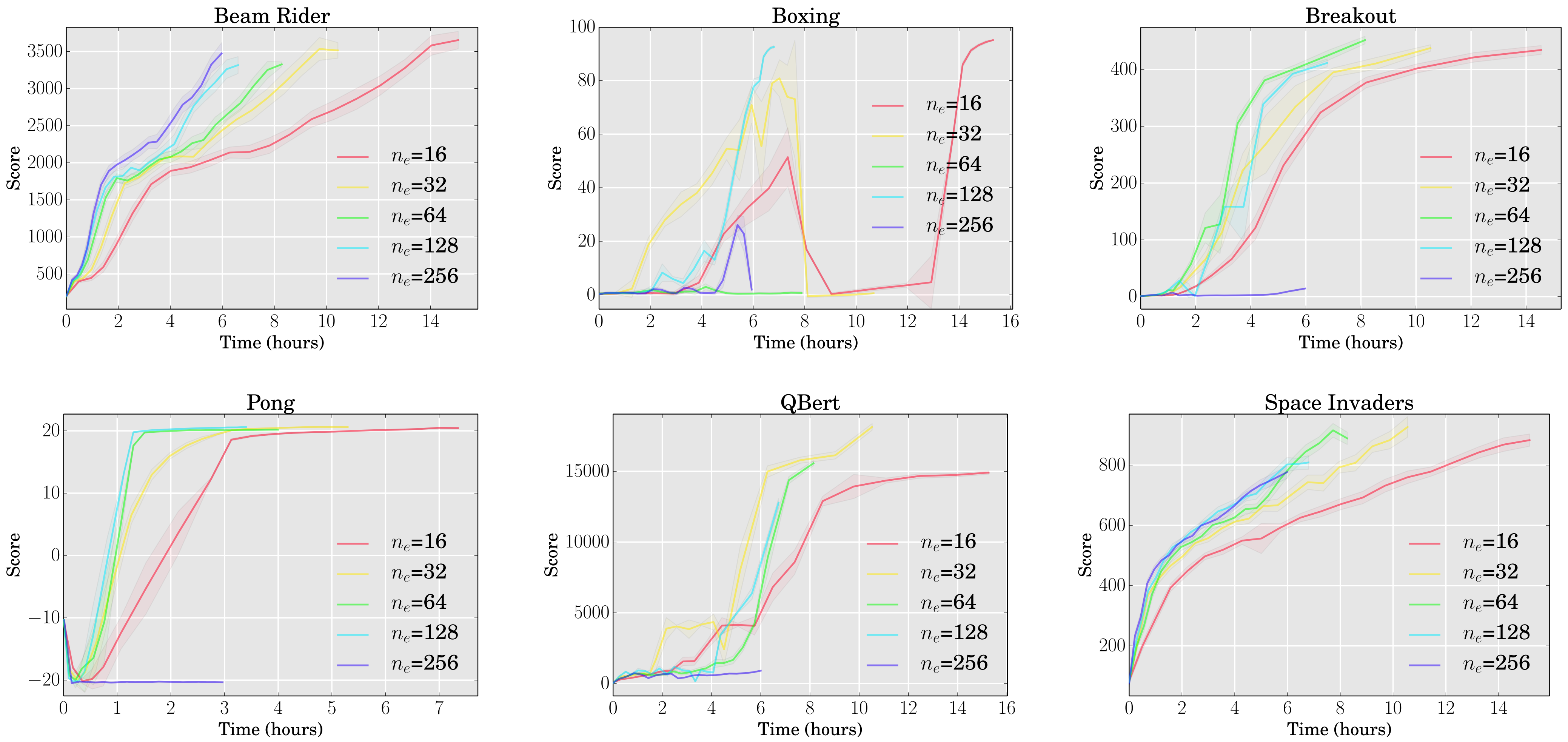
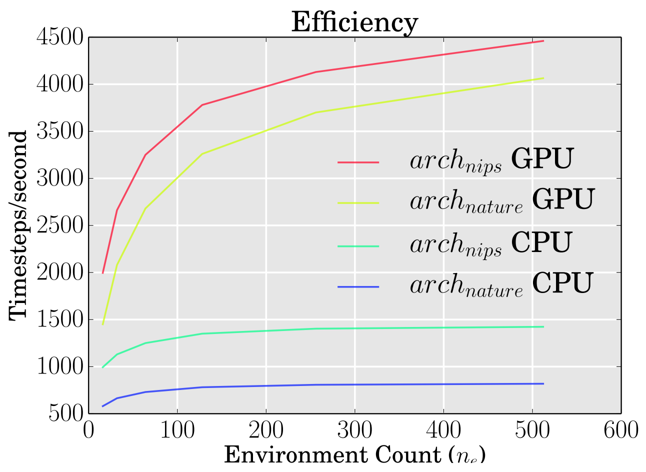
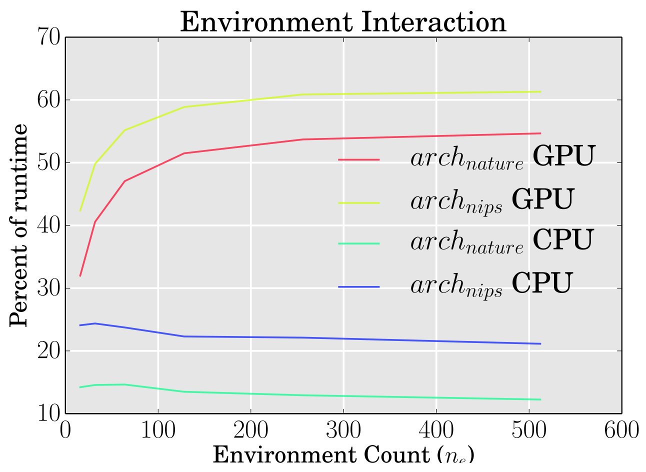


Figure 4 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Parallel Advantage Actor Critic





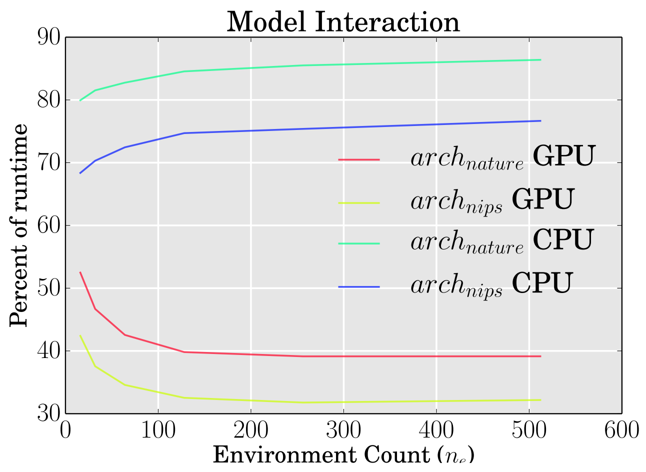


Figure 2 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

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

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

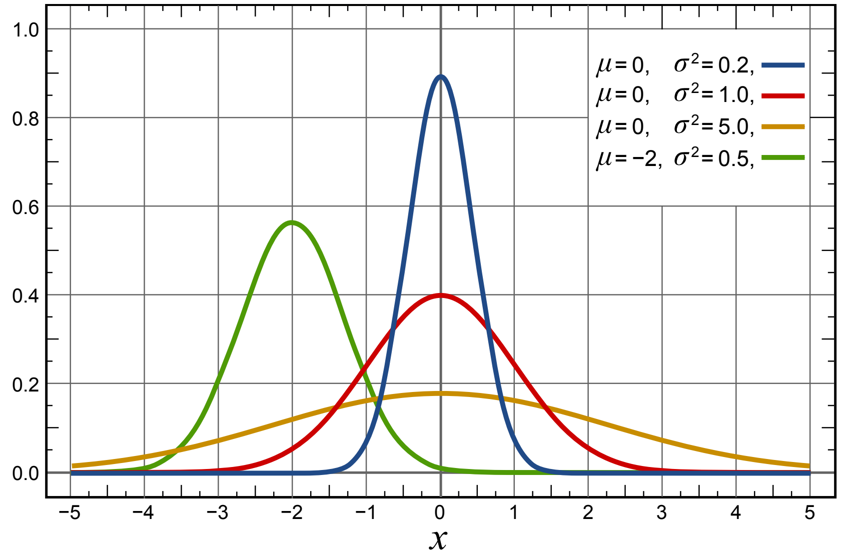


Figure from section 13.7 of "Reinforcement Learning: An Introduction, Second Edition".

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).

```
mus = tf.keras.layers.Dense(actions)(hidden_layer)
sds = tf.keras.layers.Dense(actions)(hidden_layer)
sds = tf.math.exp(sds)    # or sds = tf.math.softplus(sds)

action_dist = tfp.distributions.Normal(mus, sds)

# Loss computed as - log π(a|s) - entropy_regularization
loss = - action_dict.log_prob(actions) * returns \
       - args.entropy_regularization * action_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.

Deterministic Policy Gradient Theorem

Combining continuous actions and Deep Q Networks is not straightforward. In order to do so, we need a different variant of the policy gradient theorem.

Recall that in policy gradient theorem,

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

Deterministic Policy Gradient Theorem

Assume that the policy $\pi(s; \boldsymbol\theta)$ is deterministic and computes an action $a \in \mathbb{R}$. Then under several assumptions about continuousness, the following holds:

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \mathbb{E}_{s\sim\mu(s)} \Big[\nabla_{\boldsymbol\theta} \pi(s; \boldsymbol\theta) \nabla_a q_\pi(s, a)\big|_{a=\pi(s;\boldsymbol\theta)}\Big].$$

The theorem was first proven in the paper Deterministic Policy Gradient Algorithms by David Silver et al.

Deterministic Policy Gradient Theorem – Proof

The proof is very similar to the original (stochastic) policy gradient theorem. We assume that $p(s'|s,a), \nabla_a p(s'|s,a), r(s,a), \nabla_a r(s,a), \pi(s;\boldsymbol\theta), \nabla_{\boldsymbol\theta}\pi(s;\boldsymbol\theta)$ are continuous in all params.

$$\begin{aligned}
\nabla_{\boldsymbol\theta} v_\pi(s) &= \nabla_{\boldsymbol\theta} q_\pi(s, \pi(s;\boldsymbol\theta)) \\
&= \nabla_{\boldsymbol\theta}\Big(r\big(s,\pi(s;\boldsymbol\theta)\big) + \int_{s'} p\big(s'|s,\pi(s;\boldsymbol\theta)\big) v_\pi(s')\,\mathrm{d}s'\Big) \\
&= \nabla_{\boldsymbol\theta}\pi(s;\boldsymbol\theta)\nabla_a r(s,a)\big|_{a=\pi(s;\boldsymbol\theta)} + \nabla_{\boldsymbol\theta}\int_{s'} p\big(s'|s,\pi(s;\boldsymbol\theta)\big) v_\pi(s')\,\mathrm{d}s' \\
&= \nabla_{\boldsymbol\theta}\pi(s;\boldsymbol\theta)\nabla_a \Big(r(s,a) + \int_{s'} p\big(s'|s,a\big) v_\pi(s')\,\mathrm{d}s'\Big)\Big|_{a=\pi(s;\boldsymbol\theta)} \\
&\qquad + \int_{s'} p\big(s'|s,\pi(s;\boldsymbol\theta)\big)\nabla_{\boldsymbol\theta} v_\pi(s')\,\mathrm{d}s' \\
&= \nabla_{\boldsymbol\theta}\pi(s;\boldsymbol\theta)\nabla_a q_\pi(s,a)\big|_{a=\pi(s;\boldsymbol\theta)} + \int_{s'} p\big(s'|s,\pi(s;\boldsymbol\theta)\big)\nabla_{\boldsymbol\theta} v_\pi(s')\,\mathrm{d}s'
\end{aligned}$$

We finish the proof analogously to the gradient theorem by continually expanding $\nabla_{\boldsymbol\theta} v_\pi(s')$.

Deep Deterministic Policy Gradients

Note that the formulation of deterministic policy gradient theorem allows an off-policy algorithm, because the loss functions no longer depends on actions (similarly to how expected Sarsa is also an off-policy algorithm).

We therefore train function approximation for both $\pi(s; \boldsymbol\theta)$ and $q(s, a; \boldsymbol\theta)$, training $q(s, a; \boldsymbol\theta)$ using a deterministic variant of the Bellman equation:

$$q(S_t, A_t; \boldsymbol\theta) = \mathbb{E}_{R_{t+1}, S_{t+1}}\big[R_{t+1} + \gamma q(S_{t+1}, \pi(S_{t+1}; \boldsymbol\theta))\big]$$

and $\pi(s; \boldsymbol\theta)$ according to the deterministic policy gradient theorem.

The algorithm was first described in the paper Continuous Control with Deep Reinforcement Learning by Timothy P. Lillicrap et al. (2015).

The authors utilize a replay buffer, a target network (updated by exponential moving average with $\tau = 0.001$), batch normalization for CNNs, and perform exploration by adding a normal-distributed noise to predicted actions. Training is performed by Adam with learning rates of 1e-4 and 1e-3 for the policy and critic network, respectively.

Deep Deterministic Policy Gradients

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    **for** t = 1, T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    **end for**
**end for**
```

Algorithm 1 of the paper "Continuous Control with Deep Reinforcement Learning" by Timothy P. Lillicrap et al.

Deep Deterministic Policy Gradients

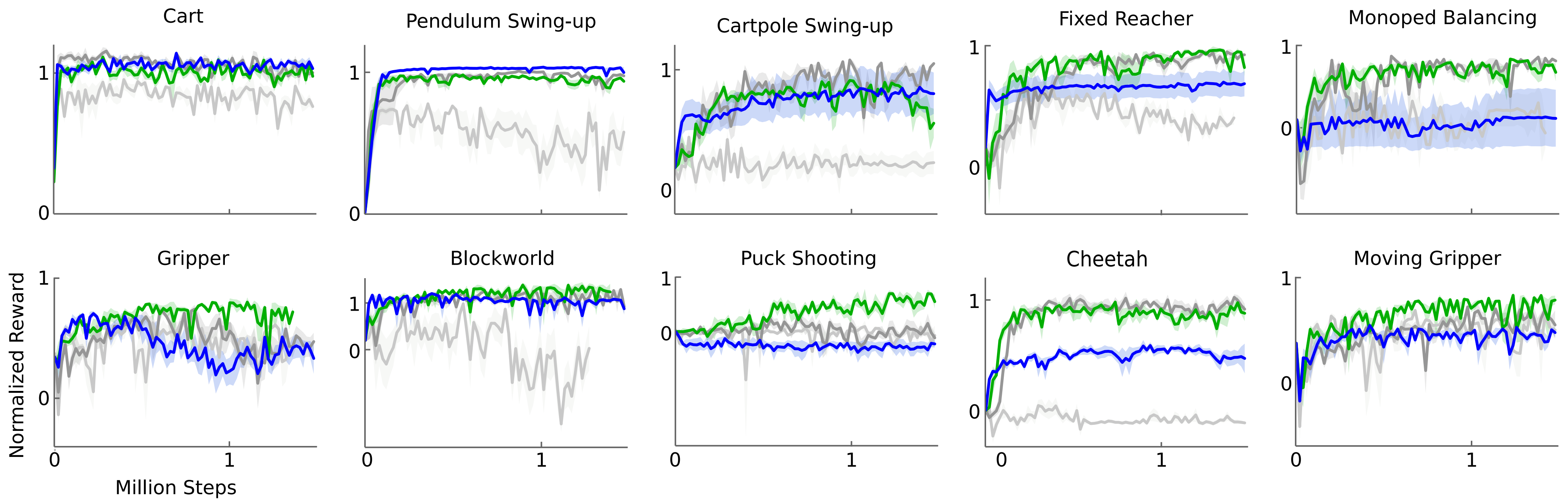
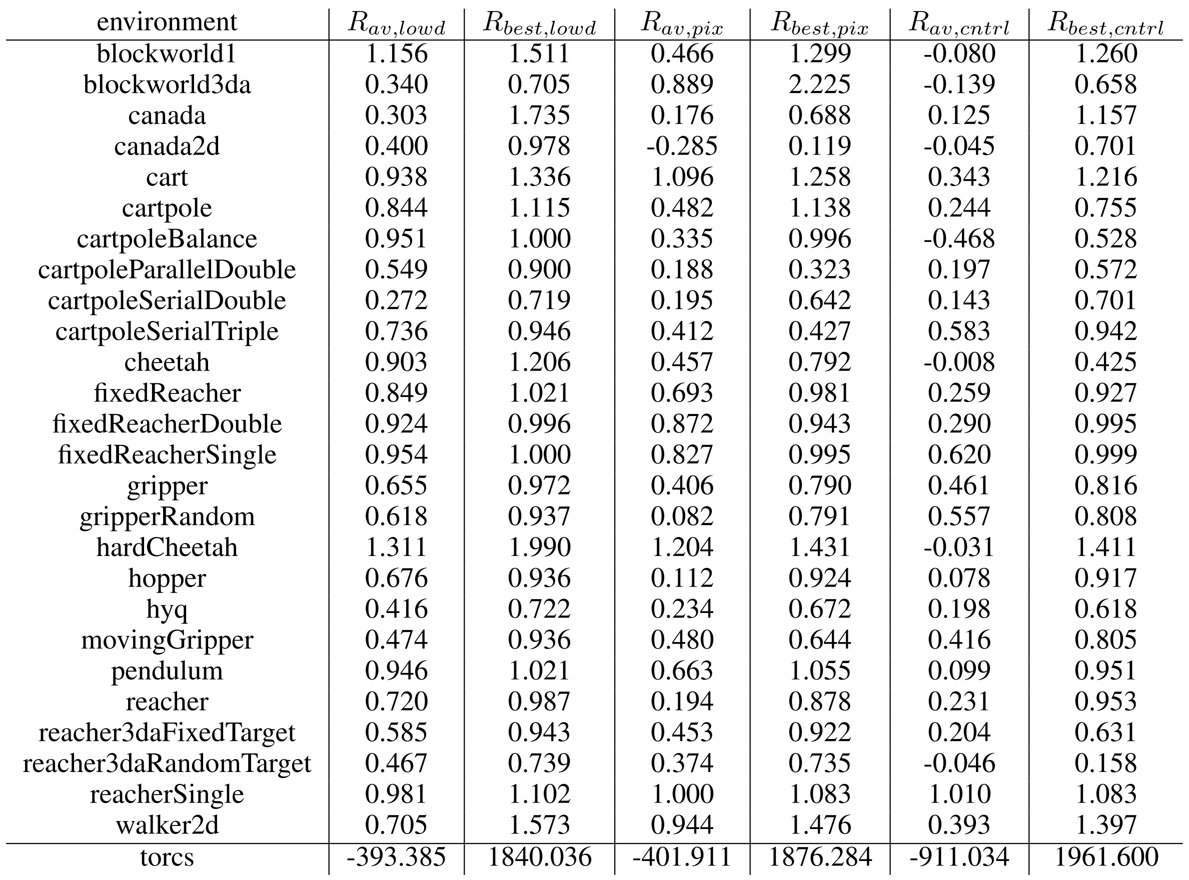


Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target networks are crucial.

Figure 3 of the paper "Continuous Control with Deep Reinforcement Learning" by Timothy P. Lillicrap et al.

Deep Deterministic Policy Gradients

Results using low-dimensional (*lowd*) version of the environment, pixel representation (*pix*) and DPG reference (*cntrl*).



environment	$R_{av,lowd}$	$R_{best,lowd}$	$R_{av,pix}$	$R_{best,pix}$	$R_{av,cntrl}$	$R_{best,cntrl}$
blockworld1	1.156	1.511	0.466	1.299	-0.080	1.260
blockworld3da	0.340	0.705	0.889	2.225	-0.139	0.658
canada	0.303	1.735	0.176	0.688	0.125	1.157
canada2d	0.400	0.978	-0.285	0.119	-0.045	0.701
cart	0.938	1.336	1.096	1.258	0.343	1.216
cartpole	0.844	1.115	0.482	1.138	0.244	0.755
cartpoleBalance	0.951	1.000	0.335	0.996	-0.468	0.528
cartpoleParallelDouble	0.549	0.900	0.188	0.323	0.197	0.572
cartpoleSerialDouble	0.272	0.719	0.195	0.642	0.143	0.701
cartpoleSerialTriple	0.736	0.946	0.412	0.427	0.583	0.942
cheetah	0.903	1.206	0.457	0.792	-0.008	0.425
fixedReacher	0.849	1.021	0.693	0.981	0.259	0.927
fixedReacherDouble	0.924	0.996	0.872	0.943	0.290	0.995
fixedReacherSingle	0.954	1.000	0.827	0.995	0.620	0.999
gripper	0.655	0.972	0.406	0.790	0.461	0.816
gripperRandom	0.618	0.937	0.082	0.791	0.557	0.808
hardCheetah	1.311	1.990	1.204	1.431	-0.031	1.411
hopper	0.676	0.936	0.112	0.924	0.078	0.917
hyq	0.416	0.722	0.234	0.672	0.198	0.618
movingGripper	0.474	0.936	0.480	0.644	0.416	0.805
pendulum	0.946	1.021	0.663	1.055	0.099	0.951
reacher	0.720	0.987	0.194	0.878	0.231	0.953
reacher3daFixedTarget	0.585	0.943	0.453	0.922	0.204	0.631
reacher3daRandomTarget	0.467	0.739	0.374	0.735	-0.046	0.158
reacherSingle	0.981	1.102	1.000	1.083	1.010	1.083
walker2d	0.705	1.573	0.944	1.476	0.393	1.397
torcs	-393.385	1840.036	-401.911	1876.284	-911.034	1961.600

Table 1 of the paper "Continuous Control with Deep Reinforcement Learning" by Timothy P. Lillicrap et al.