Deterministic Policy Gradient, Advanced RL Algorithms

Milan Straka

December 10, 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 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* $a_\pi(s, a) \stackrel{\text{def}}{=} q_\pi(s, a) - v_\pi(s)$.

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.

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 Celemente et al., who named their agent *parallel advantage actor-critic* (PAAC).

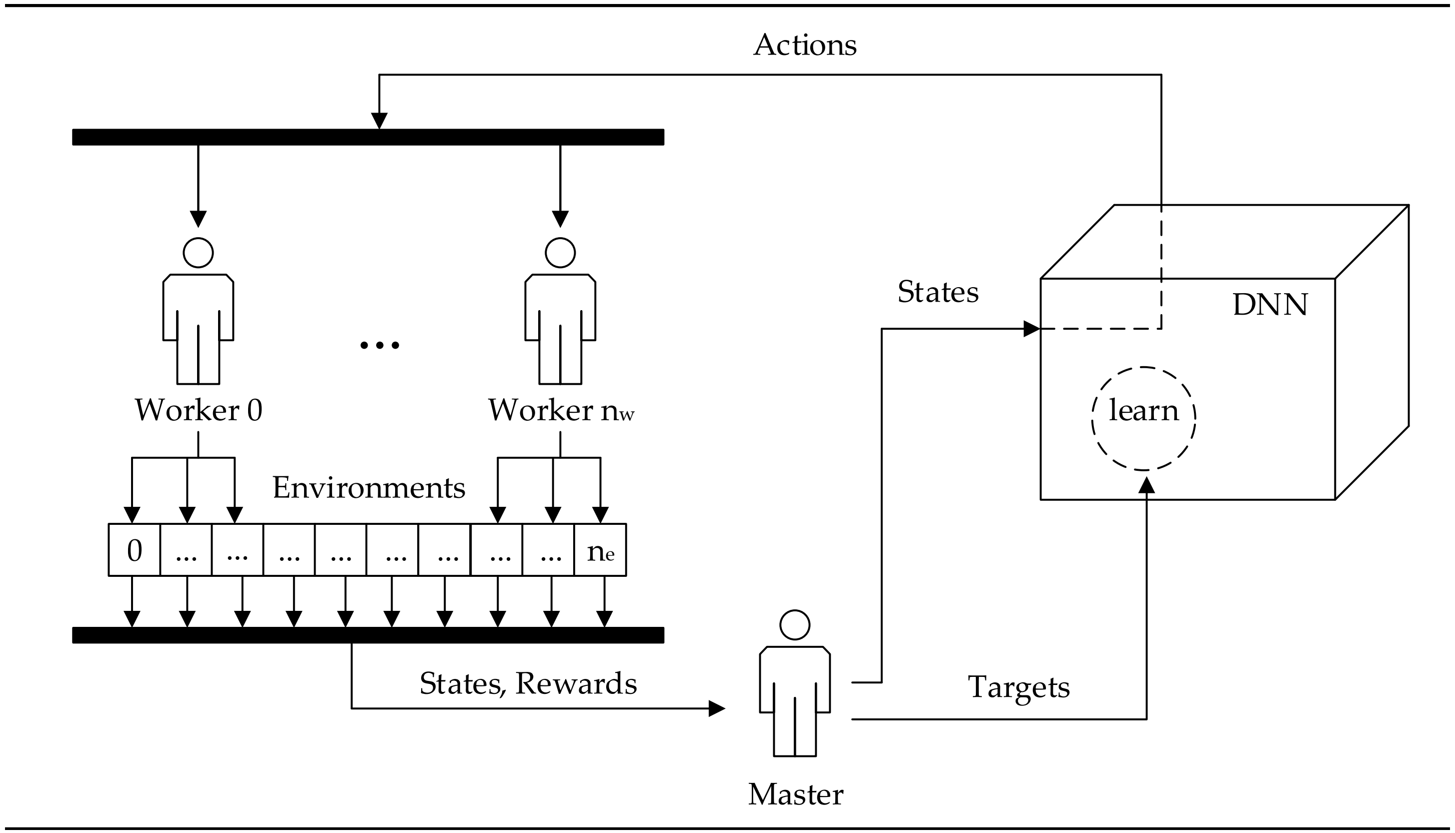


Figure 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

Continuous Action Space

Until now, the actions were discreet. 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

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

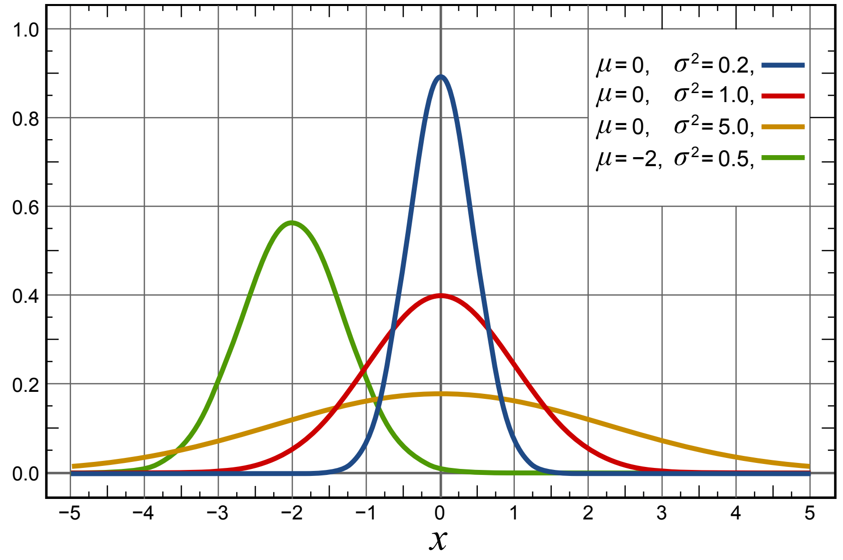


Figure from section 13.7 of "Reinforcement Learning: An Introduction, Second Edition".

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()
```

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.

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

Similarly to the gradient theorem, we finish the proof 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

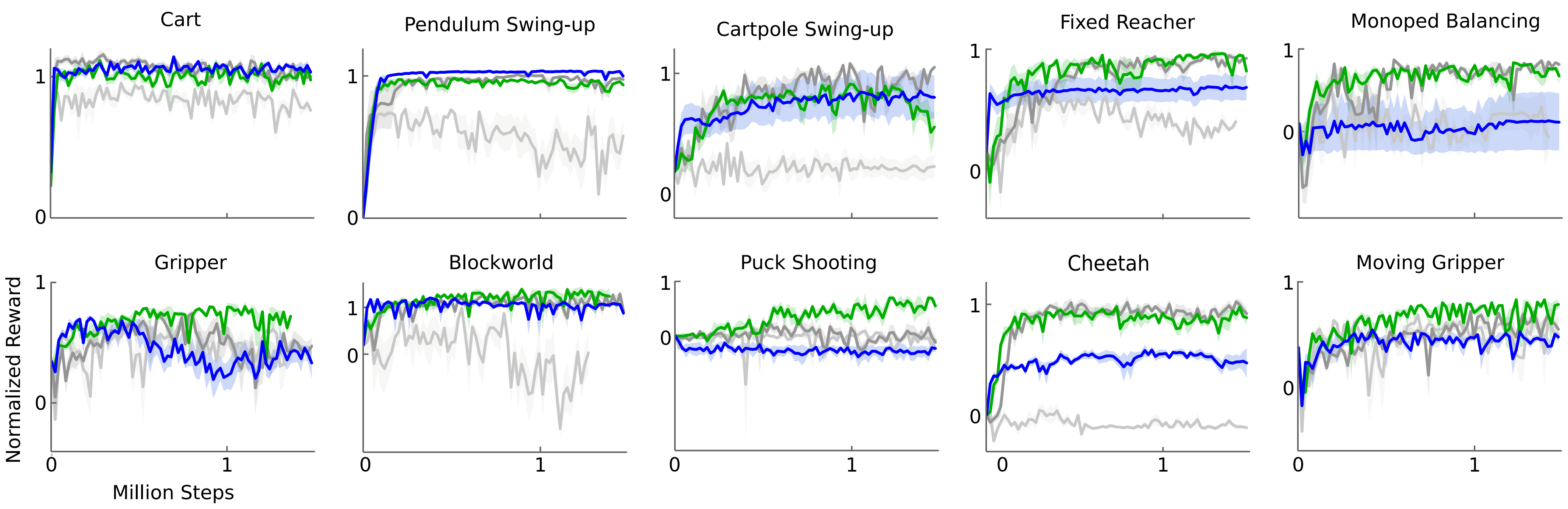
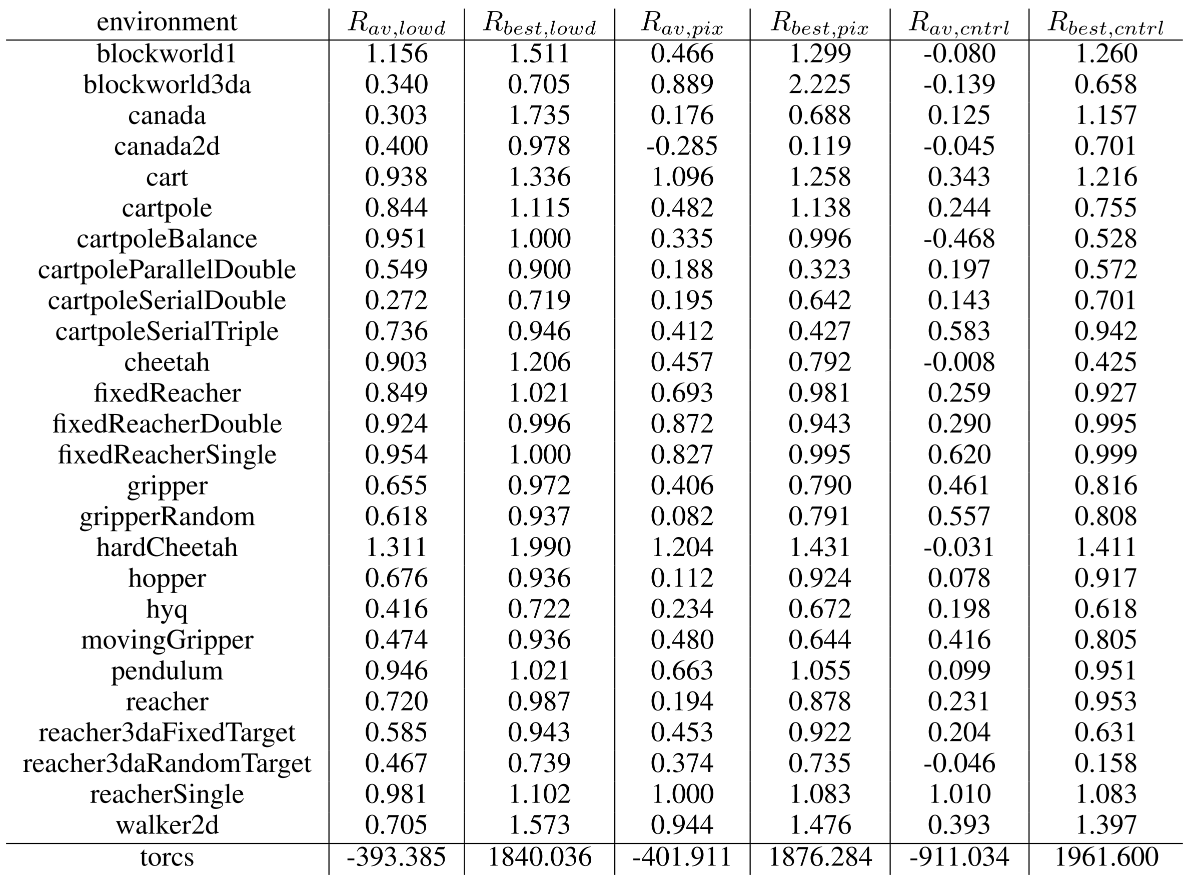


Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target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

Natural Policy Gradient

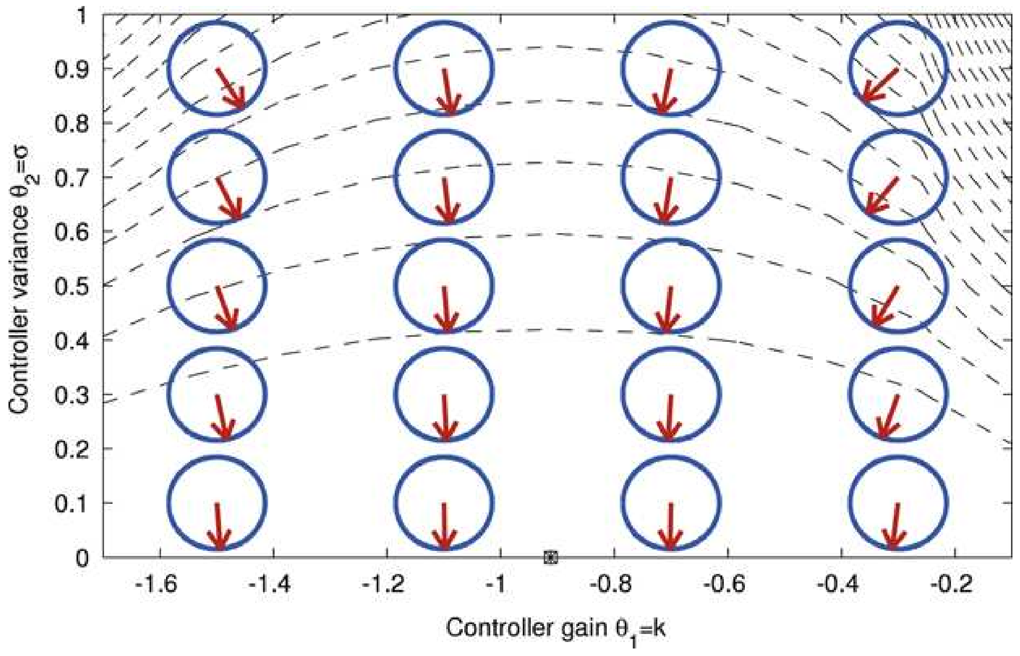
The following approach has been introduced by Kakade (2002).

Using policy gradient theorem, we are able to compute ∇v_π. Normally, we update the parameters by using directly this gradient. This choice is justified by the fact that a vector $\boldsymbol{d}$ which maximizes $v_\pi(s; \boldsymbol{\theta} + \boldsymbol{d})$ under the constraint that $|\boldsymbol{d}|^2$ is bounded by a small constant is exactly the gradient ∇v_π.

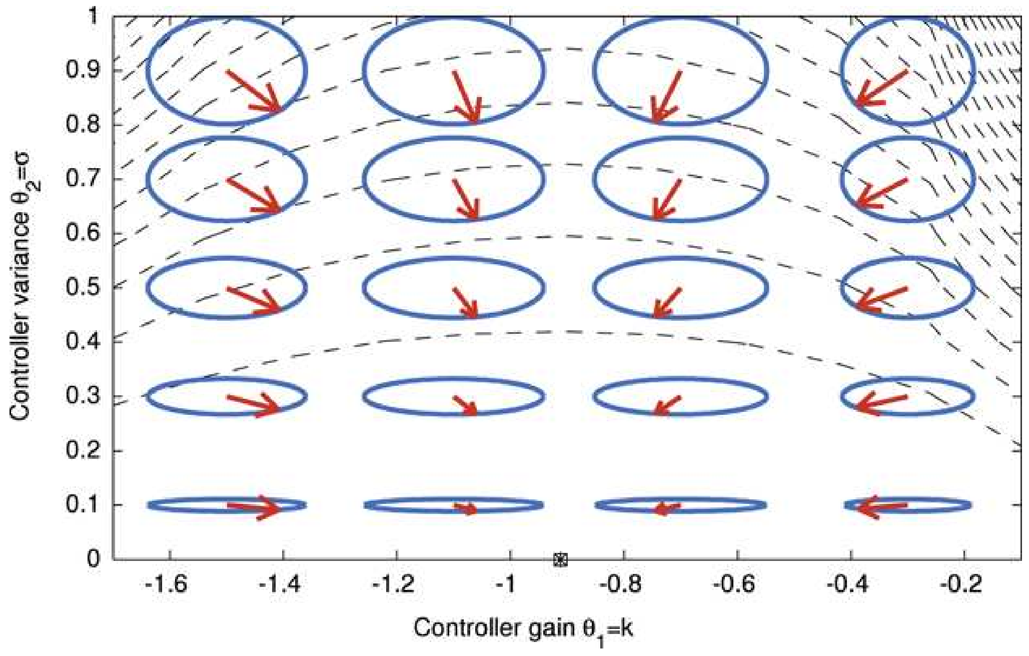
Normally, the length $|\boldsymbol{d}|^2$ is computed using Euclidean metric. But in general, any metric could be used. Representing a metric using a positive-definite matrix $\boldsymbol{G}$ (identity matrix for Euclidean metric), we can compute the distance as $|\boldsymbol{d}|^2 = \sum_{ij} G_{ij} d_i d_j = \boldsymbol{d}^T \boldsymbol{G} \boldsymbol{d}$. The steepest ascent direction is then given by $\boldsymbol{G}^{-1} \nabla v_\pi$.

Note that when $\boldsymbol{G}$ is the Hessian $\boldsymbol{H} v_\pi$, the above process is exactly Newton's method.

Natural Policy Gradient



(a) Vanilla policy gradient.



(b) Natural policy gradient.

Figure 3 of the paper "Reinforcement learning of motor skills with policy gradients" by Jan Peters et al.

Natural Policy Gradient

A suitable choice for the metric is *Fisher information matrix* defined as

$$F_s(\boldsymbol\theta) \stackrel{\text{def}}{=} \mathbb{E}_{\pi(a|s;\boldsymbol\theta)}\left[\frac{\partial \log \pi(a|s;\boldsymbol\theta)}{\partial \boldsymbol\theta_i}\frac{\partial \log \pi(a|s;\boldsymbol\theta)}{\partial \boldsymbol\theta_j}\right] = \mathbb{E}[\nabla\pi(a|s;\boldsymbol\theta)]\mathbb{E}[\nabla\pi(a|s;\boldsymbol\theta)]^T.$$

It can be shown that the Fisher information metric is the only Riemannian metric (up to rescaling) invariant to change of parameters under sufficient statistic.

Recall Kullback-Leibler distance (or relative entropy) defined as

$$D_{\mathrm{KL}}(\boldsymbol p||\boldsymbol q) \stackrel{\text{def}}{=} \sum_i p_i \log\frac{p_i}{q_i} = H(p,q) - H(p).$$

The Fisher information matrix is also a Hessian of the $D_{\mathrm{KL}}(\pi(a|s;\boldsymbol\theta)||\pi(a|s;\boldsymbol\theta'))$:

$$F_s(\boldsymbol\theta) = \frac{\partial^2}{\partial\theta'_i\partial\theta'_j}D_{\mathrm{KL}}(\pi(a|s;\boldsymbol\theta)||\pi(a|s;\boldsymbol\theta'))\Big|_{\boldsymbol\theta'=\boldsymbol\theta}.$$

Natural Policy Gradient

Using the metric

$$F(\boldsymbol\theta) = \mathbb{E}_{s\sim\mu_{\boldsymbol\theta}} F_s(\boldsymbol\theta)$$

we want to update the parameters using $\boldsymbol d_F \stackrel{\text{def}}{=} F(\boldsymbol\theta)^{-1}\nabla v_\pi$.

An interesting property of using the $\boldsymbol d_F$ to update the parameters is that

- updating $\boldsymbol\theta$ using ∇v_π will choose an arbitrary *better* action in state s;
- updating $\boldsymbol\theta$ using $F(\boldsymbol\theta)^{-1}\nabla v_\pi$ chooses the *best* action (maximizing expected return), similarly to tabular greedy policy improvement.

However, computing $\boldsymbol d_F$ in a straightforward way is too costly.

Truncated Natural Policy Gradient

Duan et al. (2016) in paper *Benchmarking Deep Reinforcement Learning for Continuous Control* propose a modification to the NPG to efficiently compute $\boldsymbol{d}_F$.

Following Schulman et al. (2015), they suggest to use *conjugate gradient algorithm*, which can solve a system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ in an iterative manner, by using $\boldsymbol{A}$ only to compute products $\boldsymbol{A}\boldsymbol{v}$ for a suitable $\boldsymbol{v}$.

Therefore, $\boldsymbol{d}_F$ is found as a solution of

$$F(\boldsymbol{\theta})\boldsymbol{d}_F = \nabla v_\pi$$

and using only 10 iterations of the algorithm seem to suffice according to the experiments.

Furthermore, Duan et al. suggest to use a specific learning rate suggested by Peters et al (2008) of

$$\frac{\alpha}{\sqrt{(\nabla v_\pi)^T F(\boldsymbol{\theta})^{-1} \nabla v_\pi}}.$$

Trust Region Policy Optimization

Schulman et al. in 2015 wrote an influential paper introducing TRPO as an improved variant of NPG.

Considering two policies $\pi, \tilde\pi$, we can write

$$v_{\tilde\pi} = v_\pi + \mathbb{E}_{s\sim\mu(\tilde\pi)} \mathbb{E}_{a\sim\tilde\pi(a|s)} a_\pi(a|s),$$

where $a_\pi(a|s)$ is the advantage function $q_\pi(a|s) - v_\pi(s)$ and $\mu(\tilde\pi)$ is the on-policy distribution of the policy $\tilde\pi$.

Analogously to policy improvement, we see that if $a_\pi(a|s) \geq 0$, policy $\tilde\pi$ performance increases (or stays the same if the advantages are zero everywhere).

However, sampling states $s \sim \mu(\tilde\pi)$ is costly. Therefore, we instead consider

$$L_\pi(\tilde\pi) = v_\pi + \mathbb{E}_{s\sim\mu(\pi)} \mathbb{E}_{a\sim\tilde\pi(a|s)} a_\pi(a|s).$$

Trust Region Policy Optimization

$$L_\pi(\tilde\pi) = v_\pi + \mathbb{E}_{s\sim\mu(\pi)}\mathbb{E}_{a\sim\tilde\pi(a|s)} a_\pi(a|s)$$

It can be shown that for parametrized $\pi(a|s;\boldsymbol\theta)$ the $L_\pi(\tilde\pi)$ matches $v_{\tilde\pi}$ to the first order.

Schulman et al. additionally proves that if we denote $\alpha = D_\textrm{KL}^\textrm{max}(\pi_\textrm{old}||\pi_\textrm{new}) = \max_s D_\textrm{KL}\big(\pi_\textrm{old}(\cdot|s)||\pi_\textrm{new}(\cdot|s)\big)$, then

$$v_{\pi_\textrm{new}} \geq L_{\pi_\textrm{old}}(\pi_\textrm{new}) - \frac{4\varepsilon\gamma}{(1-\gamma)^2}\alpha \quad\textrm{where}\quad \varepsilon = \max_{s,a}|a_\pi(s,a)|.$$

Therefore, TRPO minimizes $L_{\pi_{\boldsymbol\theta_0}}(\pi_{\boldsymbol\theta})$ subject to $D_\textrm{KL}^{\boldsymbol\theta_0}(\pi_{\boldsymbol\theta_0}||\pi_{\boldsymbol\theta}) < \delta$, where

- $D_\textrm{KL}^{\boldsymbol\theta_0}(\pi_{\boldsymbol\theta_0}||\pi_{\boldsymbol\theta}) = \mathbb{E}_{s\sim\mu(\pi_{\boldsymbol\theta_0})}\big[D_\textrm{KL}\big(\pi_\textrm{old}(\cdot|s)||\pi_\textrm{new}(\cdot|s)\big)\big]$ is used instead of $D_\textrm{KL}^\textrm{max}$ for performance reasons;
- δ is a constant found empirically, as the one implied by the above equation is too small;
- importance sampling is used to account for sampling actions from π.

Trust Region Policy Optimization

$$\text{minimize } L_{\pi_{\boldsymbol\theta_0}}(\pi_{\boldsymbol\theta}) \text{ subject to } D_{\mathrm{KL}}^{\boldsymbol\theta_0}(\pi_{\boldsymbol\theta_0}||\pi_{\boldsymbol\theta}) < \delta$$

The parameters are updated using $\boldsymbol{d}_F = F(\boldsymbol\theta)^{-1}\nabla L_{\pi_{\boldsymbol\theta_0}}(\pi_{\boldsymbol\theta})$, utilizing the conjugate gradient algorithm as described earlier for TNPG (note that the algorithm was designed originally for TRPO and only later employed for TNPG).

To guarantee improvement and respect the D_{KL} constraint, a line search is in fact performed. We start by the learning rate of $\sqrt{\delta/(\boldsymbol{d}_F^T F(\boldsymbol\theta)^{-1}\boldsymbol{d}_F)}$ and shrink it exponentially until the constraint is satistifed and the objective improves.

Trust Region Policy Optimization

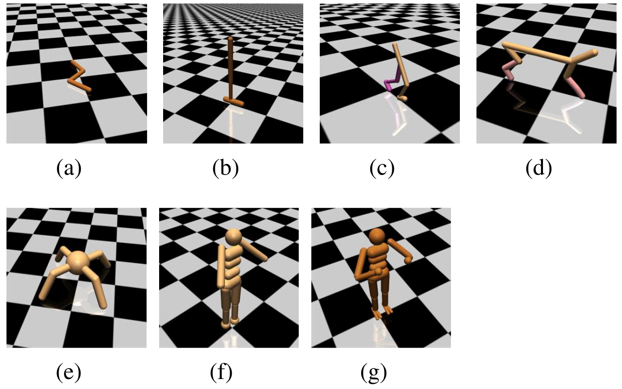


Figure 1. Illustration of locomotion tasks: (a) Swimmer; (b) Hopper; (c) Walker; (d) Half-Cheetah; (e) Ant; (f) Simple Humanoid; and (g) Full Humanoid.

Figure 1 of the paper "Benchmarking Deep Reinforcement Learning for Continuous Control" by Duan et al.

Task	Random	REINFORCE	TNPG	RWR	REPS	TRPO	CEM	CMA-ES	DDPG
Cart-Pole Balancing	77.1 ± 0.0	4693.7 ± 14.0	**3986.4 ± 748.9**	**4861.5 ± 12.3**	565.6 ± 137.6	**4869.8 ± 37.6**	4815.4 ± 4.8	2440.4 ± 568.3	4634.4 ± 87.8
Inverted Pendulum*	−153.4 ± 0.2	13.4 ± 18.0	**209.7 ± 55.5**	84.7 ± 13.8	−113.3 ± 4.6	**247.2 ± 76.1**	38.2 ± 25.7	−40.1 ± 5.7	40.0 ± 244.6
Mountain Car	−415.4 ± 0.0	−67.1 ± 1.0	**-66.5 ± 4.5**	−79.4 ± 1.1	−275.6 ± 166.3	**-61.7 ± 0.9**	−66.0 ± 2.4	−85.0 ± 7.7	−288.4 ± 170.3
Acrobot	−1904.5 ± 1.0	−508.1 ± 91.0	−395.8 ± 121.2	−352.7 ± 35.9	−1001.5 ± 10.8	−326.0 ± 24.4	−436.8 ± 14.7	−785.6 ± 13.1	**-223.6 ± 5.8**
Double Inverted Pendulum*	149.7 ± 0.1	4116.5 ± 65.2	**4455.4 ± 37.6**	3614.8 ± 368.1	446.7 ± 114.8	**4412.4 ± 50.4**	2566.2 ± 178.9	1576.1 ± 51.3	2863.4 ± 154.0
Swimmer*	−1.7 ± 0.1	92.3 ± 0.1	**96.0 ± 0.2**	60.7 ± 5.5	3.8 ± 3.3	**96.0 ± 0.2**	68.8 ± 2.4	64.9 ± 1.4	85.8 ± 1.8
Hopper	8.4 ± 0.0	714.0 ± 29.3	**1155.1 ± 57.9**	553.2 ± 71.0	86.7 ± 17.6	**1183.3 ± 150.0**	63.1 ± 7.8	20.3 ± 14.3	267.1 ± 43.5
2D Walker	−1.7 ± 0.0	506.5 ± 78.8	**1382.6 ± 108.2**	136.0 ± 15.9	−37.0 ± 38.1	**1353.8 ± 85.0**	84.5 ± 19.2	77.1 ± 24.3	318.4 ± 181.6
Half-Cheetah	−90.8 ± 0.3	1183.1 ± 69.2	**1729.5 ± 184.6**	376.1 ± 28.2	34.5 ± 38.0	**1914.0 ± 120.1**	330.4 ± 274.8	441.3 ± 107.6	**2148.6 ± 702.7**
Ant*	13.4 ± 0.7	548.3 ± 55.5	**706.0 ± 127.7**	37.6 ± 3.1	39.0 ± 9.8	**730.2 ± 61.3**	49.2 ± 5.9	17.8 ± 15.5	326.2 ± 20.8
Simple Humanoid	41.5 ± 0.2	128.1 ± 34.0	**255.0 ± 24.5**	93.3 ± 17.4	28.3 ± 4.7	**269.7 ± 40.3**	60.6 ± 12.9	28.7 ± 3.9	99.4 ± 28.1
Full Humanoid	13.2 ± 0.1	262.2 ± 10.5	**288.4 ± 25.2**	46.7 ± 5.6	41.7 ± 6.1	**287.0 ± 23.4**	36.9 ± 2.9	N/A ± N/A	119.0 ± 31.2

Table 1 of the paper "Benchmarking Deep Reinforcement Learning for Continuous Control" by Duan et al.

Proximal Policy Optimization

A simplification of TRPO which can be implemented using a few lines of code.

Let $r_t(\boldsymbol\theta) \stackrel{\text{def}}{=} \frac{\pi(A_t|S_t;\boldsymbol\theta)}{\pi(A_t|S_t;\boldsymbol\theta_\text{old})}$. PPO minimizes the objective

$$L^{\text{CLIP}}(\boldsymbol\theta) \stackrel{\text{def}}{=} \mathbb{E}_t\Big[\min\big(r_t(\boldsymbol\theta)\hat A_t, \operatorname{clip}(r_t(\boldsymbol\theta), 1-\varepsilon, 1+\varepsilon)\hat A_t\big)\Big].$$

Such $L^{\text{CLIP}}(\boldsymbol\theta)$ is a lower (pessimistic) bound.

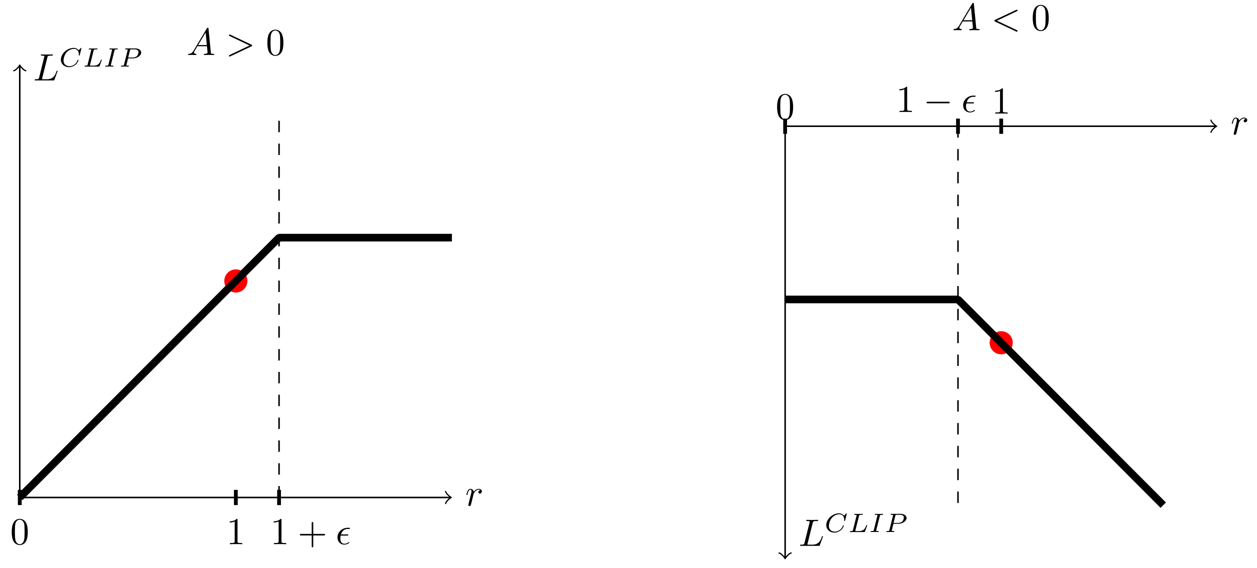


Figure 1 of the paper "Proximal Policy Optimization Algorithms" by Schulman et al.

Proximal Policy Optimization

The advantages $\hat{A}_t$ are additionally estimated using *generalized advantage estimation*. Instead of the usual $\hat{A}_t \overset{\text{def}}{=} \sum_{i=0}^{T-t-1} \gamma^i R_{t+1+i} + \gamma^{T-t} V(S_T) - V(S_t)$ the authors employ

$$\hat{A}_t \overset{\text{def}}{=} \sum_{i=0}^{T-t-1} (\gamma\lambda)^i \delta_{t+i},$$

where $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$.

Algorithm 1 PPO, Actor-Critic Style

```
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \le NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Algorithm 1 of the paper "Proximal Policy Optimization Algorithms" by Schulman et al.

Proximal Policy Optimization

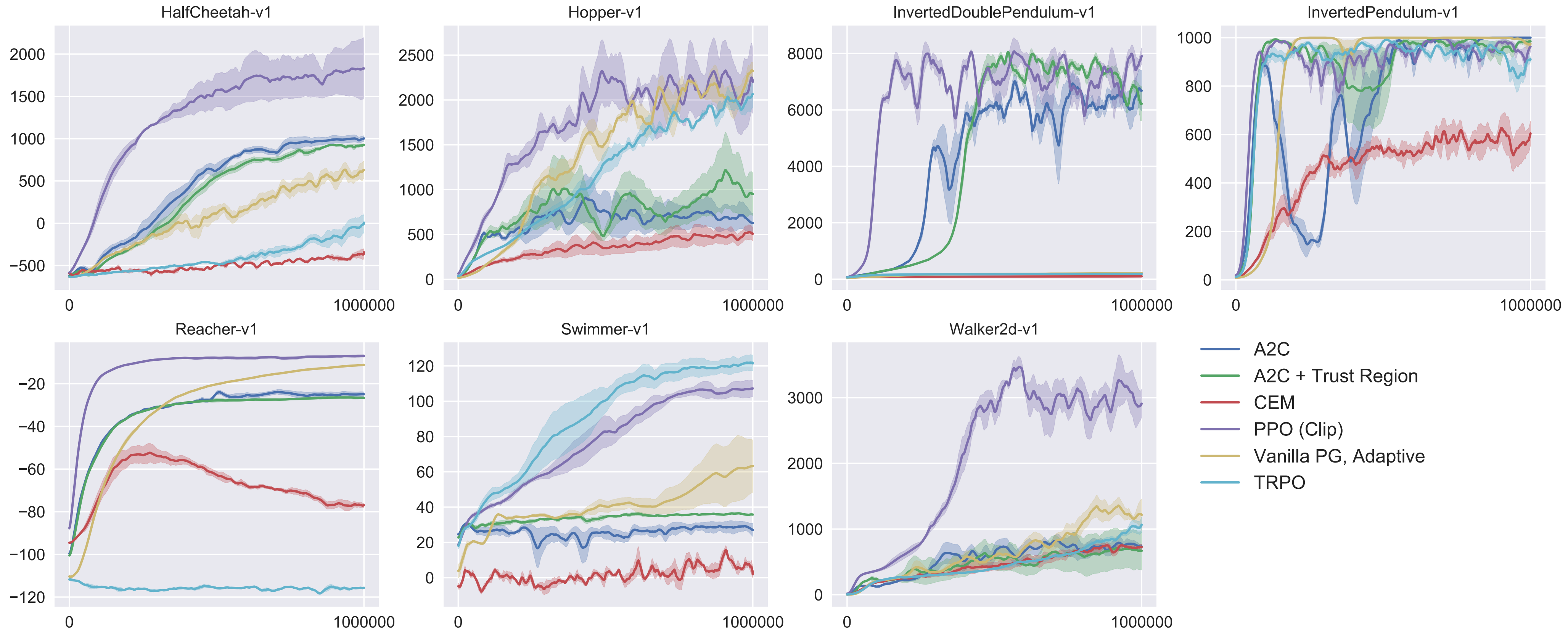


Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.

Figure 3 of the paper "Proximal Policy Optimization Algorithms" by Schulman et al.

Soft Actor Critic

The paper Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor by Tuomas Haarnoja et al. introduces a different off-policy algorithm for continuous action space.

The general idea is to introduce entropy directly in the value function we want to maximize.

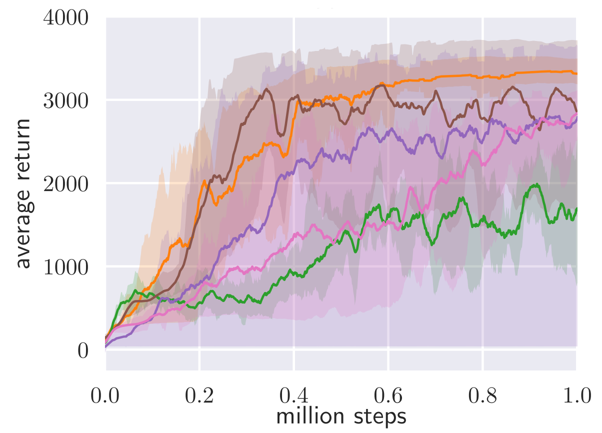
Soft Actor Critic

Algorithm 1 Soft Actor-Critic

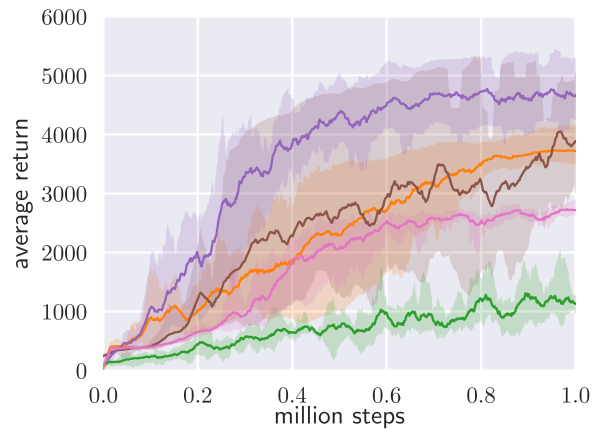
```
Initialize parameter vectors $\psi$, $\bar{\psi}$, $\theta$, $\phi$.
for each iteration do
    for each environment step do
        $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
        $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
    for each gradient step do
        $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
end for
```

Algorithm 1 of the paper "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" by Tuomas Haarnoja et al.

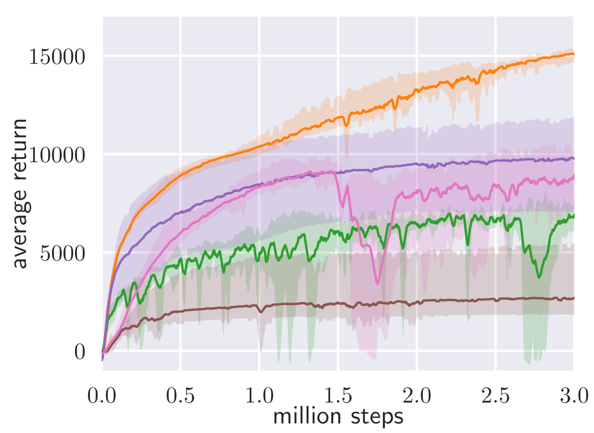
Soft Actor Critic



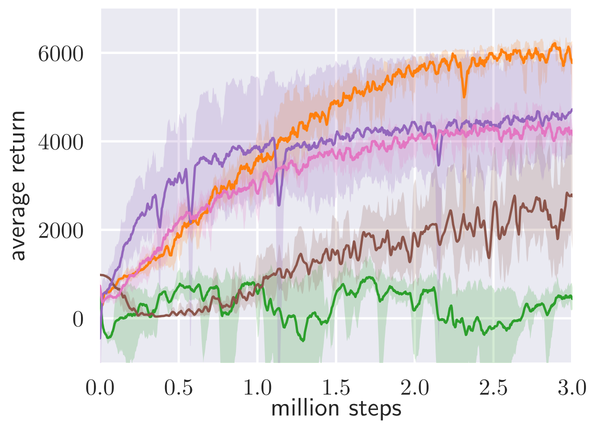
(a) Hopper-v1



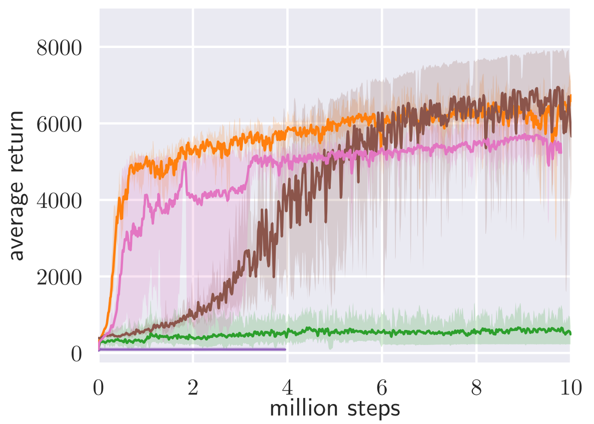
(b) Walker2d-v1



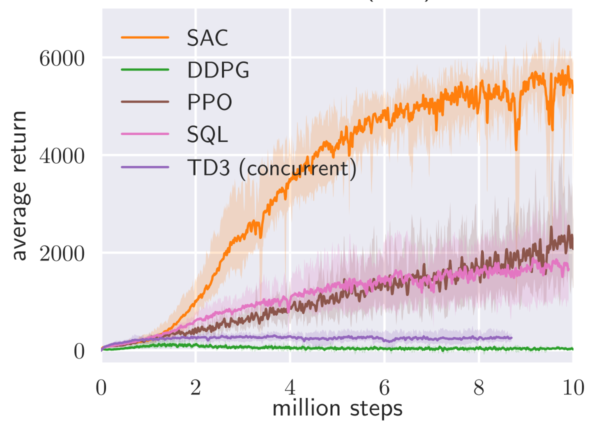
(c) HalfCheetah-v1



(d) Ant-v1



(e) Humanoid-v1



(f) Humanoid (rllab)

Figure 1 of the paper "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" by Tuomas Haarnoja et al.