Continuous control with deep reinforcement learning

By Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess,Tom Erez, Yuval Tassa, David Silver & Daan Wierstra - Google Deep Mind

Elad Hoffer

Deep reinforcement learning journal club
December 2015

Outline

Overview

With the success of DQN, a natural expansion is to try and learn actions on continuous domains.
This can be challenging due to curse-of-dimensionality: the number of actions increases exponentially with the number of degrees of freedom.

- For example, a 7 degree of freedom system (as in the human arm) with the coarsest discretization $a_i \in \{-k, 0, k\}$ for each joint leads to an action space with dimensionality: $3^7 = 2187$.

Continuous reinforcement learning

This work will present a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces.
It is heavily based on 2 previous works:

- DPG - Deterministic policy gradient (Silver 2014) algorithm (itself similar to NFQCA from 2001)
- DQN - Deep Q Network (Mnih 2013, 2015)

Continuous reinforcement learning

We consider a standard reinforcement learning setup consisting of an agent interacting with an environment E in discrete timesteps, while its actions are continuous.

- At each timestep t the agent receives an observation x_t, takes an action $a_t \in R^N$ and receives a scalar reward r_t.
- The environment is assumed here to be fully-observed so $s_t = x_t$.
- An agent's behavior is defined by a policy, π, which maps states to a probability distribution over the actions $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$.
- The return from a state is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$ with a discounting factor $\gamma \in [0, 1]$.

Action-value function

The action-value function is used in many reinforcement learning algorithms. It describes the expected return after taking an action a_t in state s_t and thereafter following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}\left[R_t | s_t, a_t\right] \tag{1}$$

The target policy is deterministic, so we can describe it as a function $\mu : \mathcal{S} \leftarrow \mathcal{A}$

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))\right] \tag{2}$$

The expectation depends only on the environment. This means that it is possible to learn Q^{μ} off-policy, using transitions which are generated from a different behavior policy μ'.

Deep Q-learning

Following the earlier success on Atari games, they wish to use Q-learning, together with a neural-network based value estimator (parameterized by θ^Q) optimized by minimizing the loss:

$$L(\theta^Q) = \mathbb{E}_{\mu'}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right] \tag{3}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q). \tag{4}$$

Deep Q-learning

Prior to DQN, it was generally believed that learning value functions using large, non-linear function approximators was difficult and unstable.
DQN is able to learn value functions using such function approximators in a stable and robust way due to two innovations:

- *replay buffer* - the network is trained off-policy with samples from a replay buffer to minimize correlations between samples;
- *target network* - the network is trained with a “target” Q network to give consistent targets during temporal difference backups.

The DPG algorithm

It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of $a_t = \arg\max_a Q(s, a)$ at every timestep;

- this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces.

Instead, an actor-critic approach based on the DPG algorithm is used.

The DPG algorithm

DPG maintains a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action.

- The critic $Q(s, a)$ is learned using the Bellman equation as in Q-learning.
- The actor is updated by applying the chain rule with respect to the actor parameters:

$$\begin{aligned}\nabla_{\theta^{\mu}}\mu &\approx \mathbb{E}_{\mu'}\left[\nabla_{\theta^{\mu}} Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}\right] \\ &= \mathbb{E}_{\mu'}\left[\nabla_{a} Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_{\mu}}\mu(s|\theta^{\mu})|_{s=s_t}\right]\end{aligned} \tag{5}$$

David Silver proved (2014) that this is the *policy gradient*, the gradient of the policy's performance.

Target network modification

The target network is similar to the one used previously by deepMind, but modified for actor-critic and using “soft” target updates, rather than directly copying the weights:

- Create a copy of the actor and critic networks, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, that are used for calculating the target values.
- The weights of these target networks are then updated by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$.
- Target values are constrained to change slowly, greatly improving the stability of learning.

Other techniques used

- Batch-normalization (Ioffe, 2015) - was used on the state input and on all layers of the μ network and all layers of the Q network prior to the action input.
  This effectively allows learning across many different tasks with differing types of units, without needing to manually ensure the units were within a set range.
- Exploration noise process - added to the actor (using the fact that this is an off-policy algorithm).

DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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 gradient:

$$\nabla_{\theta^\mu} \mu|_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}, \qquad \theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

Environments

A wide variety of domains was tested.

- Classic reinforcement learning environments such as cartpole.
- Difficult, high dimensional tasks such as gripper
- Tasks involving contacts such as puck striking (*canada*)
- Locomotion tasks such as *cheetah*.

In all domains but *cheetah* the actions were torques applied to the actuated joints.

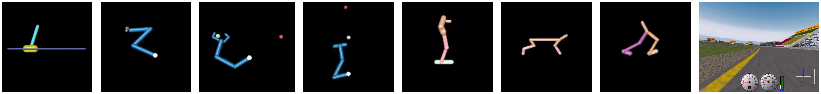


Figure : Environments

Evaluation

Experiments were ran using 2 modes:

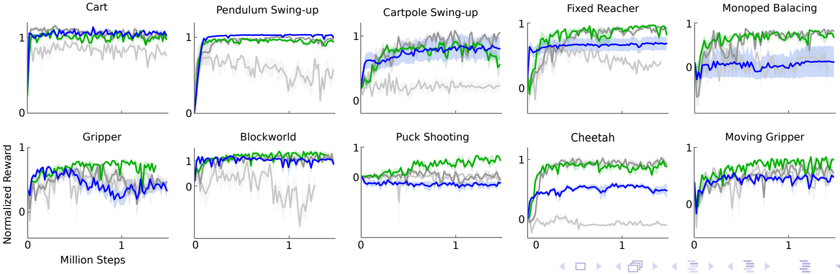
- Low-dimensional state description (such as joint angles and positions)
- High-dimensional renderings of the environment - 64x64 rgb images fed into a convolutional network similar to DQN.

As in DQN, and in order to make the problems approximately fully observable in the high dimensional environment, they used *action repeats*.

Videos Link

Performance

- The results are compared to a planner (iLQG), with full access to the underlying physical model and its derivatives.
- Surprisingly, in some simpler tasks, learning policies from pixels is just as fast as learning using the low-dimensional state descriptor.
- In order to perform well across all tasks, both target network and batch normalization are necessary.



Q-Value estimates

- Q-learning is prone to over-estimating values.
- DDPG's estimates were examined empirically by comparing the values estimated by Q after training with the true returns seen on test episodes.
- DDPG estimates returns accurately without systematic biases. For harder tasks the Q estimates are worse, but DDPG is still able learn good policies.

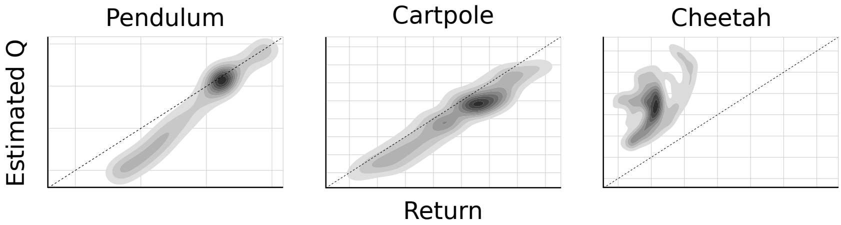


Figure : returns

Table : Performance after training across all environments

environment	$R_{av,lowd}$	$R_{best,lowd}$	$R_{av,pix}$	$R_{best,pix}$
blockworld1	1.156	1.511	0.466	1.299
blockworld3da	0.340	0.705	0.889	2.225
canada	0.303	1.735	0.176	0.688
canada2d	0.400	0.978	-0.285	0.119
cart	0.938	1.336	1.096	1.258
cartpole	0.844	1.115	0.482	1.138
cartpoleBalance	0.951	1.000	0.335	0.996
cartpoleParallelDouble	0.549	0.900	0.188	0.323
cartpoleSerialDouble	0.272	0.719	0.195	0.642
cartpoleSerialTriple	0.736	0.946	0.412	0.427
cheetah	0.903	1.206	0.457	0.792
fixedReacher	0.849	1.021	0.693	0.981
fixedReacherDouble	0.924	0.996	0.872	0.943
fixedReacherSingle	0.954	1.000	0.827	0.995
gripper	0.655	0.972	0.406	0.790
gripperRandom	0.618	0.937	0.082	0.791
hardCheetah	1.311	1.990	1.204	1.431
hopper	0.676	0.936	0.112	0.924
hyq	0.416	0.722	0.234	0.672
movingGripper	0.474	0.936	0.480	0.644
pendulum	0.946	1.021	0.663	1.055
reacher	0.720	0.987	0.194	0.878
reacher3daFixedTarget	0.585	0.943	0.453	0.922
reacher3daRandomTarget	0.467	0.739	0.374	0.735
reacherSingle	0.981	1.102	1.000	1.083
walker2d	0.705	1.573	0.944	1.476

Summary

In this work we've seen

- Using DQN and DPG to learn policies on continuous domains from pixel-level information "end-to-end".
- These learned policies can, in some cases, compete with even a planning algorithm with full access to the dynamics of the domain and its derivatives.
- It can be very slow, and poor in some cases, so there is definitely a lot of room to improve ;)