CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

Virginia Filippi - VR495315

Martina Toffoli - VR501056

Reinforcement Learning &
Advanced Programming course

2023/2024

Timothy P. Lillicrap∗, Jonathan J. Hunt∗, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra - Google Deepmind – ICLR 2016

INTRODUCTION

What is one of the primary goals of AI?

To solve complex tasks from unprocessed sensory input (high-dimensional)

Deep Learning + Reinforcement Learning = **DQN**

- Solves problems with high-dimensional observation spaces
- BUT can only handle discrete and low-dimensional action spaces

DQN cannot be applied to continuous domains!

2. INTRODUCTION

SOLUTION:

Simply discretize the action space!

PROBLEM: many limitations in the curse of dimensionality

↓

DQN-like networks is likely intractable!

What we will present today?

3. INTRODUCTION

We present an algorithm that has the characteristics as:

- Model-free
- Based on DPG
- Combination of actor-critic algorithm and DQN

DQN is able to learn value functions using function approximators.

In this work we make use of the same ideas which we call **Deep DPG (DDPG)**.

RL DEFINITIONS

We consider a standard RL setup.

Goal in RL: learn a policy which maximizes the expected return from the start distribution.

$$J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}\left[R_1\right]$$

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

Q-learning: off-policy algorithm that uses the greedy policy $\mu(s) = \arg\max_a Q(s, a)$

Function approximators parametrized by θ^Q, which we optimize by minimizing the loss

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right]$$

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

OUTLINE OF THE PRESENTATION

Policy gradient theorem »» Actor-critic algorithms »» Deterministic Policy Gradient (DPG) »» Deep DPG (DDPG)

POLICY GRADIENT THEOREM

$J(\boldsymbol{\theta})$ is a measure of policy performance depending on policy parameters $\boldsymbol{\theta}$.

$J(\boldsymbol{\theta})$ function in stochastic case:

$$J(\pi_\theta) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) r(s,a) \mathrm{d}a \mathrm{d}s$$

$$= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\left[r(s,a)\right]$$

POLICY GRADIENT THEOREM

The goal of policy gradient methods is to learn parameters $\boldsymbol{\theta}$ that maximize $J(\boldsymbol{\theta})$

Parameter updates approximate gradient ascent in J: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate of the gradient of $J(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}_t$

The **policy gradient theorem** for the episodic case establishes that:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta})\right]
\end{aligned}$$

where gradients are column vectors of partial derivatives w.r.t. the components of θ and $\mu(s)$ is the on-policy state distribution under policy π (parametrized by θ).

POLICY GRADIENT THEOREM

Deterministic policy = $\mu_\theta : \mathcal{S} \longrightarrow \mathcal{A}$

Parameter vector: $\theta \epsilon \mathbb{R}^n$

$J(\boldsymbol{\theta})$ function in deterministic case:

$$
\begin{aligned}
J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s)) \mathrm{d}s \\
&= \mathbb{E}_{s \sim \rho^\mu} \left[r(s, \mu_\theta(s)) \right]
\end{aligned}
$$

POLICY GRADIENT THEOREM

How to pass from discrete to continuous action space?

→ Move the policy in the direction of the gradient of the Q function

Parameter updated for continuous action space

$$\theta^{k+1} = \theta^k + \alpha \mathbb{E}_{s \sim \rho^{\mu^k}} \left[\nabla_\theta Q^{\mu^k}(s, \mu_\theta(s)) \right]$$

By applying the chain rule:

$$\theta^{k+1} = \theta^k + \alpha \mathbb{E}_{s \sim \rho^{\mu^k}} \left[\nabla_\theta \mu_\theta(s) \; \nabla_a Q^{\mu^k}(s, a) \Big|_{a=\mu_\theta(s)} \right]$$

Gradient of policy

Gradient of action value

DETERMINISTIC POLICY GRADIENT THEOREM

Theorem 1

Suppose that the MDP satisfies the regularity conditions; these imply that $\nabla_\theta \mu_\theta(s)$ and $\nabla_a Q^\mu(s,a)$ exist and the deterministic policy gradient exists.

$$
\begin{aligned}
\nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \; \nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)} \, \mathrm{d}s \\
&= \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_\theta \mu_\theta(s) \; \nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)}\right]
\end{aligned}
$$

POLICY GRADIENT THEOREM

Recall the policy gradient theorem:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s, \boldsymbol{\theta})$$

From stochastic to deterministic case

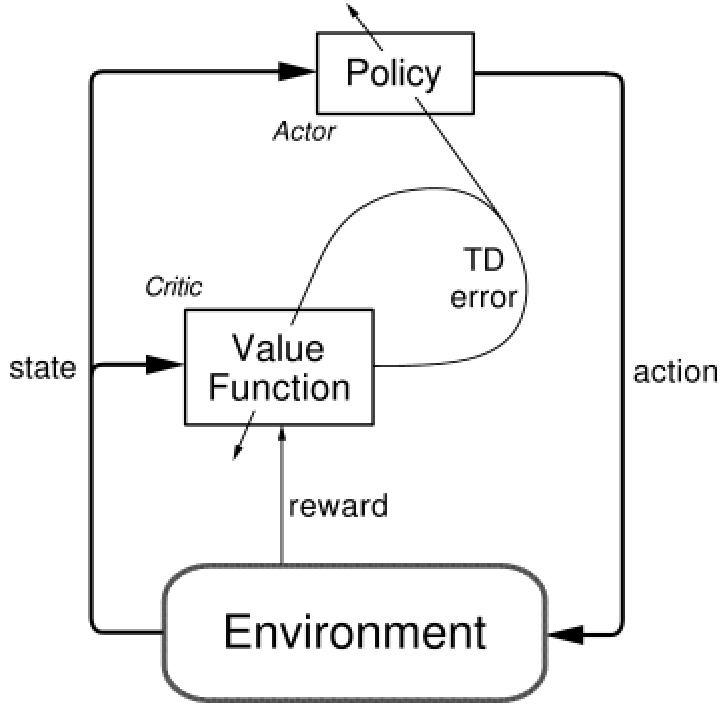
Theorem 2. Consider a stochastic policy $\pi_{\mu_\theta,\sigma}$ such that $\pi_{\mu_\theta,\sigma}(a|s) = \upsilon_\sigma(\mu_\theta(s), a)$, where σ is a parameter controlling the variance and υ_σ satisfy conditions B.1 and the MDP satisfies conditions A.1 and A.2. Then,

$$\lim_{\sigma \downarrow 0} \nabla_\theta J(\pi_{\mu_\theta,\sigma}) = \nabla_\theta J(\mu_\theta)$$

ACTOR-CRITIC ALGORITHM

Actor-critic algorithms combine 2 principal components:

- Actor that learns a policy $\pi(s;\ \theta)$
- Critic that evaluate the taken action from the actor by computing the Q function $Q(s, a)$



How they works?

1. Actor takes an action a in the state s based on the current policy $\pi_\theta(a|s)$
2. Critic estimates the Q value by computing the TD error
3. Update the actor based on the critic feedback
4. Update the critic

DPG ALGORITHM

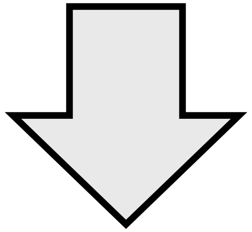
DPG algorithm uses:

- Parameterized deterministic actor function $\mu(s|\theta^{\mu})$
- The critic $Q(s,a)$ is learned using the Bellman equation
- The actor is updated by following:

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

DDPG ALGORITHM

Convergence is no longer guaranteed with non-linear function approximators.
DPG uses mini-batch but the policy is not updated at each timestep.



DDPG: modifications to original DPG to use NN function approximators in a good way by trying to solve some problems connected to neural networks

DDPG ALGORITHM

Problem	Proposed solution in DDPG
1. Most optimization algorithms assume that the samples are independently and identically distributed → this assumption no longer holds in our setting	Learn in mini-batches, to remove the dependency between sequential correlated samples → Replay buffer containing (s_t, a_t, r_t, s_{t+1}) → Used by sampling the mini-batch samples randomly from the buffer
2. The Q update is prone to divergence due to the fact that the network Q being updated is also used in computing the target value	Target networks using «soft» updates: 1. Create a copy of the actor and the critic networks $Q'(s,a\|\theta^{Q'})$ and $\mu'(s\|\theta^{\mu'})$ respectively 2. Weights are updated by having them slowly track the learned networks $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$

DDPG ALGORITHM

Problem	Proposed solution in DDPG
3 Learning across many different tasks/environments with different types of physical units in the different components of an observation → Difficult for the network to learn effectively generalizing across environments	The use of batch normalization • normalize each dimension across the samples in a minibatch to have unit mean and variance • maintain a running average of the mean and the variance to use for normalization during testing Learn effectively without needing to manually ensure the units were within a set range
4 Problem of exploration in learning in continuous action spaces	Treat the problem of exploration independently from the learning algorithm: construct an exploration policy μ' by adding noise sampled from a noise process $\mathcal{N}$ to the actor policy $\mu'(s_t) = \mu(s_t \vert \theta_t^{\mu}) + \mathcal{N}$

DDPG PSEUDO-CODE

Algorithm 1 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 policy gradient:

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

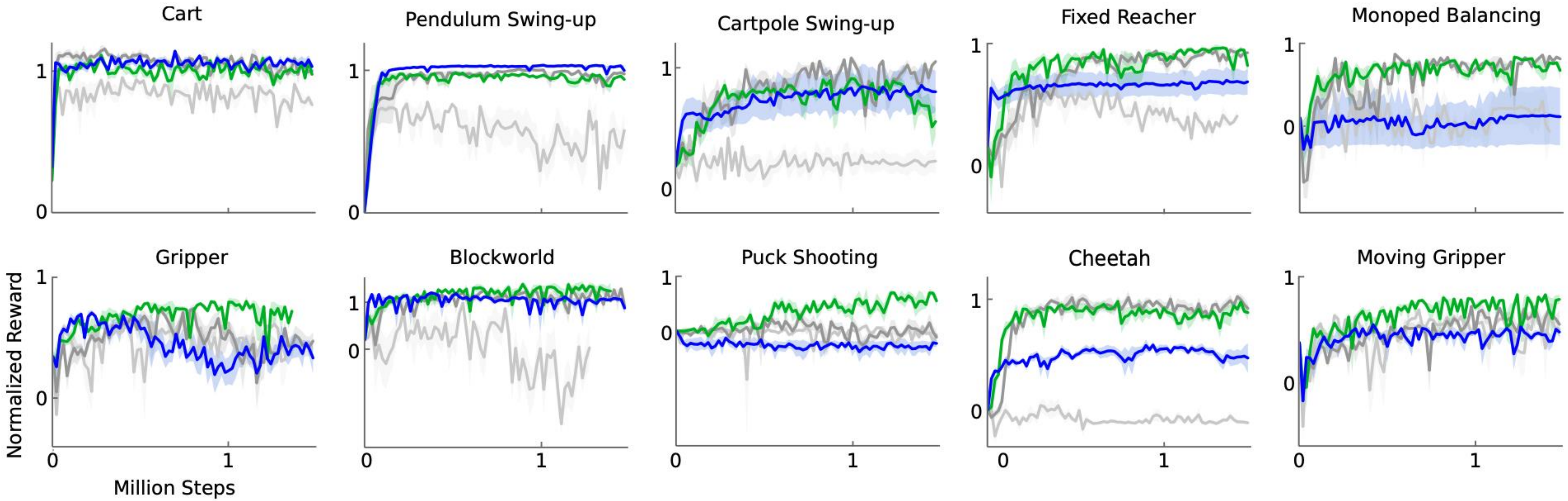
    Update the target networks:

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

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

  end for
end for

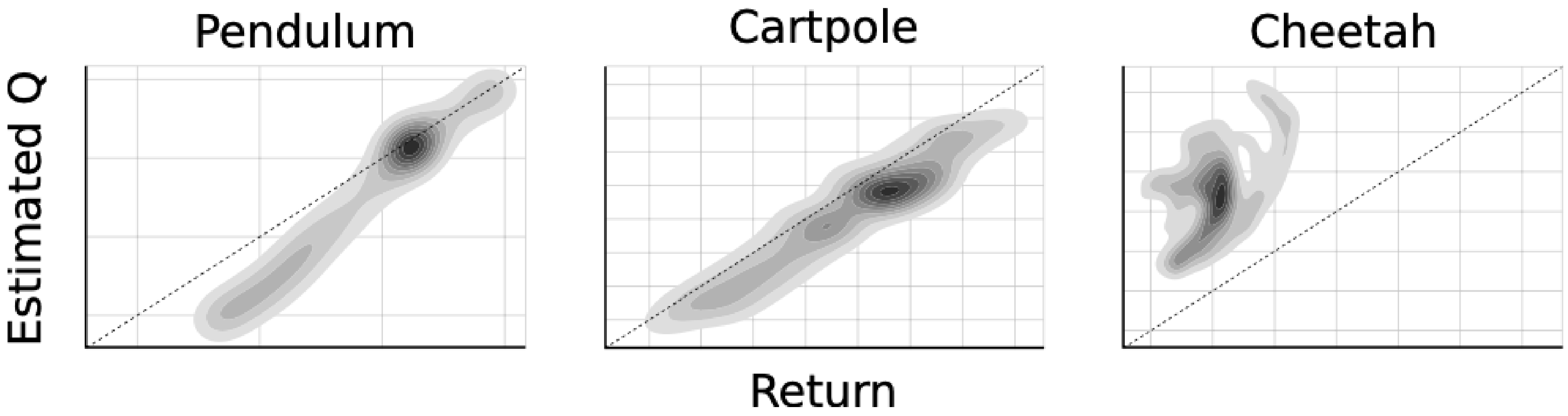
RESULTS OF THE PAPER



Legend of variants DPG:

- With batch normalization
- With target networks
- With target networks and batch normalization (DDPG)
- With target networks from pixel only inputs

RESULTS OF THE PAPER



Density plot showing estimated Q values versus observed returns sampled from test episodes on 5 replicas.

RESULTS OF THE PAPER

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

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

1: Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: **repeat**
4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** however many updates **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

13: Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_\phi(s,a) - y(r,s',d)\right)^2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

15: Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

16: **end for**
17: **end if**
18: **until** convergence

```python
class ReplayBuffer:
    """
    A simple FIFO experience replay buffer for DDPG agents.
    """

    def __init__(self, obs_dim, act_dim, size):
        self.obs_buf = np.zeros(core.combined_shape(size, obs_dim), dtype=np.float32)
        self.obs2_buf = np.zeros(core.combined_shape(size, obs_dim), dtype=np.float32)
        self.act_buf = np.zeros(core.combined_shape(size, act_dim), dtype=np.float32)
        self.rew_buf = np.zeros(size, dtype=np.float32)
        self.done_buf = np.zeros(size, dtype=np.float32)
        self.ptr, self.size, self.max_size = 0, 0, size

    def store(self, obs, act, rew, next_obs, done):
        self.obs_buf[self.ptr] = obs
        self.obs2_buf[self.ptr] = next_obs
        self.act_buf[self.ptr] = act
        self.rew_buf[self.ptr] = rew
        self.done_buf[self.ptr] = done
        self.ptr = (self.ptr+1) % self.max_size
        self.size = min(self.size+1, self.max_size)

    def sample_batch(self, batch_size=32):
        idxs = np.random.randint(0, self.size, size=batch_size)
        batch = dict(obs=self.obs_buf[idxs],
                     obs2=self.obs2_buf[idxs],
                     act=self.act_buf[idxs],
                     rew=self.rew_buf[idxs],
                     done=self.done_buf[idxs])
        return {k: torch.as_tensor(v, dtype=torch.float32) for k,v in batch.items()}
```

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

```python
def ddpg(env_fn, actor_critic=core.MLPActorCritic, ac_kwargs=dict(), seed=0,
         steps_per_epoch=4000, epochs=100, replay_size=int(1e6), gamma=0.99,
         polyak=0.995, pi_lr=1e-3, q_lr=1e-3, batch_size=100, start_steps=10000,
         update_after=1000, update_every=50, act_noise=0.1, num_test_episodes=10,
         max_ep_len=1000, logger_kwargs=dict(), save_freq=1):
```

- ac_kwargs
- seed
- steps_per_epoch
- epochs
- replay_size
- gamma
- polyak
- pi_lr
- q_lr
- batch_size
- start_steps
- update_after
- update_every
- act_noise
- num_test_episodes
- max_ep_len
- logger_kwargs
- save_freq

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for

  end if

until convergence

```python
# Set up function for computing DDPG Q-loss
def compute_loss_q(data):
    o, a, r, o2, d = data['obs'], data['act'], data['rew'], data['obs2'], data['done']

    q = ac.q(o,a)

    # Bellman backup for Q function
    with torch.no_grad():
        q_pi_targ = ac_targ.q(o2, ac_targ.pi(o2))
        backup = r + gamma * (1 - d) * q_pi_targ

    # MSE loss against Bellman backup
    loss_q = ((q - backup)**2).mean()

    # Useful info for logging
    loss_info = dict(QVals=q.detach().numpy())

    return loss_q, loss_info

# Set up function for computing DDPG pi loss
def compute_loss_pi(data):
    o = data['obs']
    q_pi = ac.q(o, ac.pi(o))
    return -q_pi.mean()
```

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

```python
def update(data):
    # First run one gradient descent step for Q.
    q_optimizer.zero_grad()
    loss_q, loss_info = compute_loss_q(data)
    loss_q.backward()
    q_optimizer.step()

    # Freeze Q-network so you don't waste computational effort
    # computing gradients for it during the policy learning step.
    for p in ac.q.parameters():
        p.requires_grad = False

    # Next run one gradient descent step for pi.
    pi_optimizer.zero_grad()
    loss_pi = compute_loss_pi(data)
    loss_pi.backward()
    pi_optimizer.step()

    # Unfreeze Q-network so you can optimize it at next DDPG step.
    for p in ac.q.parameters():
        p.requires_grad = True

    # Record things
    logger.store(LossQ=loss_q.item(), LossPi=loss_pi.item(), **loss_info)

    # Finally, update target networks by polyak averaging.
    with torch.no_grad():
        for p, p_targ in zip(ac.parameters(), ac_targ.parameters()):
            # NB: We use an in-place operations "mul_", "add_" to update target
            # params, as opposed to "mul" and "add", which would make new tensors.
            p_targ.data.mul_(polyak)
            p_targ.data.add_((1 - polyak) * p.data)
```

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi \\ \theta_{\text{targ}} &\leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta \end{aligned}$$

    end for

  end if

until convergence

```python
def get_action(o, noise_scale):
    a = ac.act(torch.as_tensor(o, dtype=torch.float32))
    a += noise_scale * np.random.randn(act_dim)
    return np.clip(a, -act_limit, act_limit)

def test_agent():
    for j in range(num_test_episodes):
        o, d, ep_ret, ep_len = test_env.reset(), False, 0, 0
        while not(d or (ep_len == max_ep_len)):
            # Take deterministic actions at test time (noise_scale=0)
            o, r, d, _ = test_env.step(get_action(o, 0))
            ep_ret += r
            ep_len += 1
        logger.store(TestEpRet=ep_ret, TestEpLen=ep_len)
```

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

1: Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: **repeat**
4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** however many updates **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

13: Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

15: Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

16: **end for**
17: **end if**
18: **until** convergence

```python
# Main loop: collect experience in env and update/log each epoch
for t in range(total_steps):

    # Until start_steps have elapsed, randomly sample actions
    # from a uniform distribution for better exploration. Afterwards,
    # use the learned policy (with some noise, via act_noise).
    if t > start_steps:
        a = get_action(o, act_noise)
    else:
        a = env.action_space.sample()

    # Step the env
    o2, r, d, _ = env.step(a)
    ep_ret += r
    ep_len += 1

    # Ignore the "done" signal if it comes from hitting the time
    # horizon (that is, when it's an artificial terminal signal
    # that isn't based on the agent's state)
    d = False if ep_len==max_ep_len else d

    # Store experience to replay buffer
    replay_buffer.store(o, a, r, o2, d)

    # Super critical, easy to overlook step: make sure to update
    # most recent observation!
    o = o2
```

SPINNINGUP CODE

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for

  end if

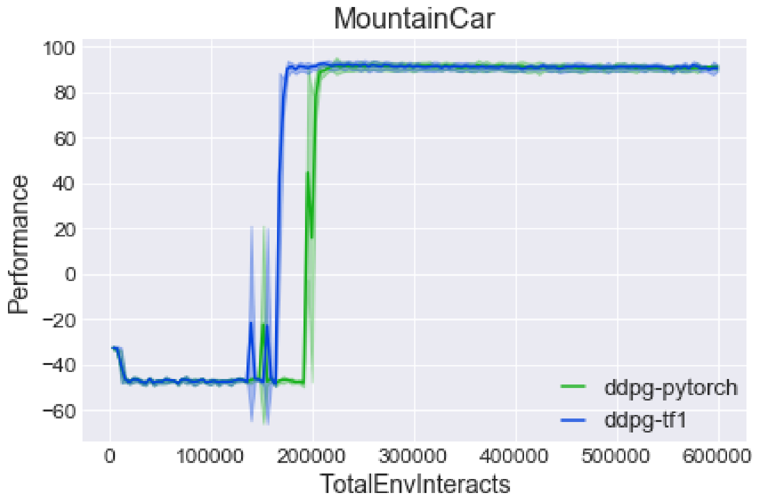
until convergence

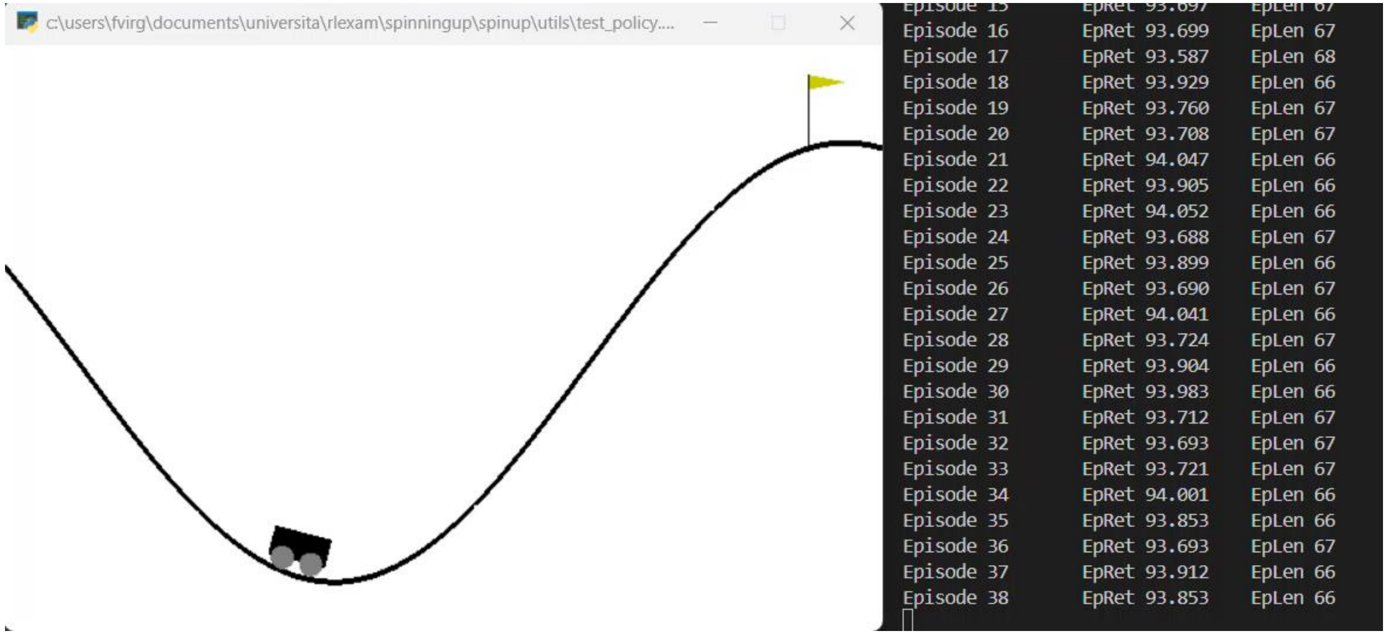
```python
# End of trajectory handling
if d or (ep_len == max_ep_len):
    logger.store(EpRet=ep_ret, EpLen=ep_len)
    o, ep_ret, ep_len = env.reset(), 0, 0

# Update handling
if t >= update_after and t % update_every == 0:
    for _ in range(update_every):
        batch = replay_buffer.sample_batch(batch_size)
        update(data=batch)

# End of epoch handling
if (t+1) % steps_per_epoch == 0:
    epoch = (t+1) // steps_per_epoch

    # Save model
    if (epoch % save_freq == 0) or (epoch == epochs):
        logger.save_state({'env': env}, None)

    # Test the performance of the deterministic version of the agent.
    test_agent()
```

RESULTS OF SPINNINGUP CODE

MountainCar Continuous

Action Space	Box(-1.0, 1.0, (1,), float32)
Observation Shape	(2,)
Observation High	[0.6 0.07]
Observation Low	[-1.2 -0.07]
Import	`gym.make("MountainCarContinuous-v0")`

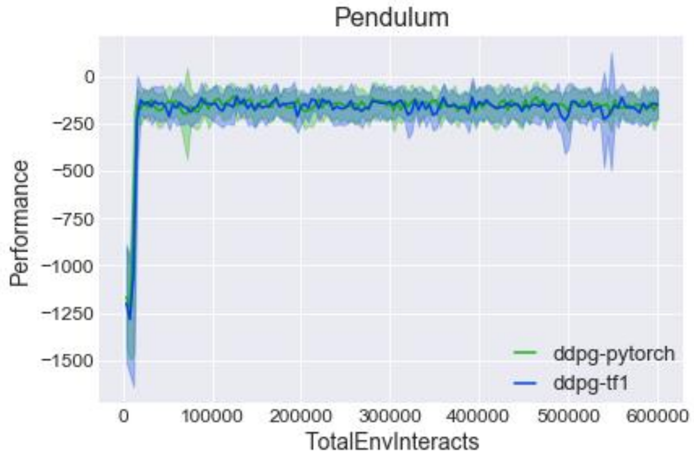


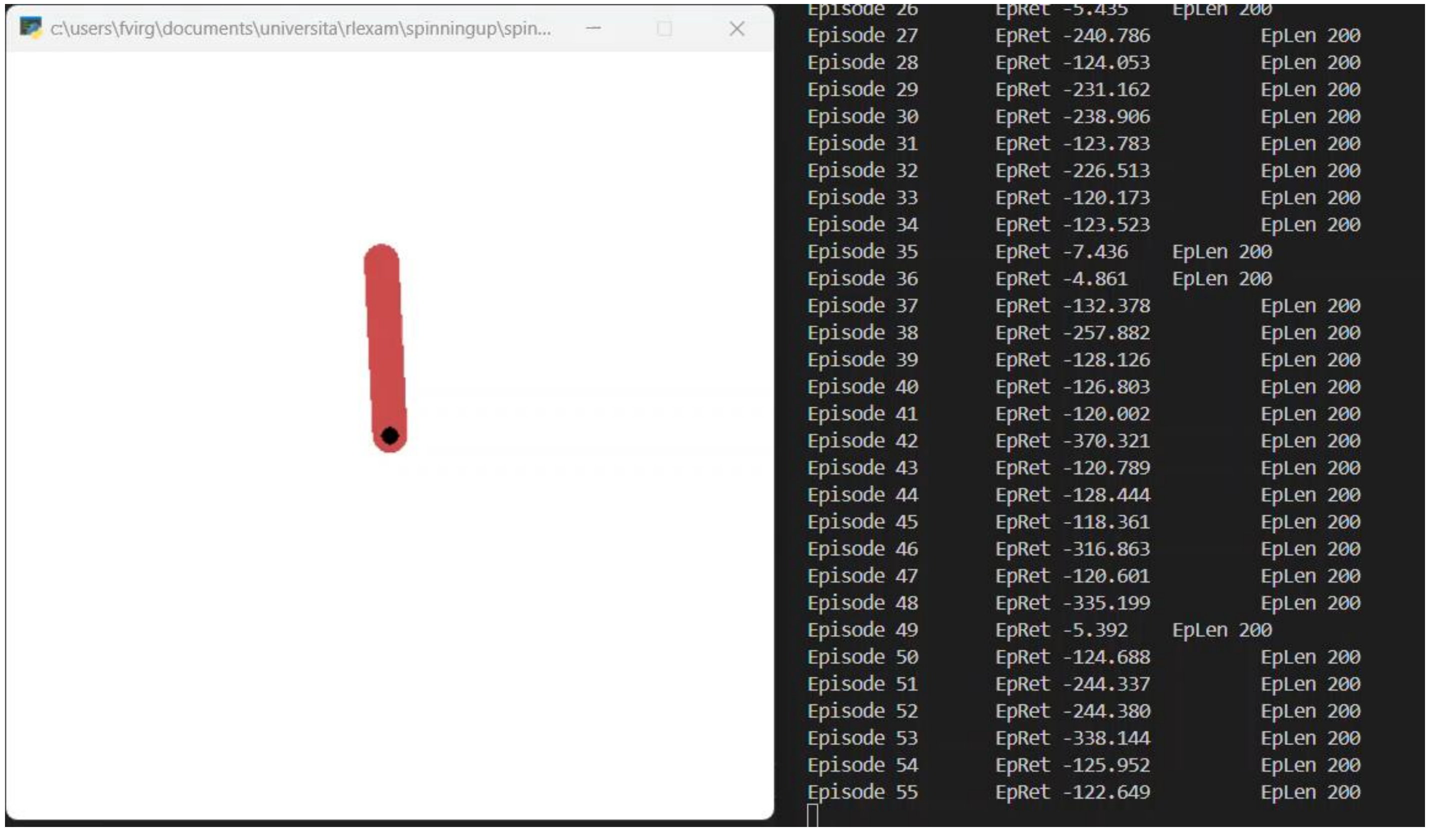


RESULTS OF SPINNINGUP CODE

Pendulum

Action Space	Box(-2.0, 2.0, (1,), float32)
Observation Shape	(3,)
Observation High	[1. 1. 8.]
Observation Low	[-1. -1. -8.]
Import	`gym.make("Pendulum-v1")`

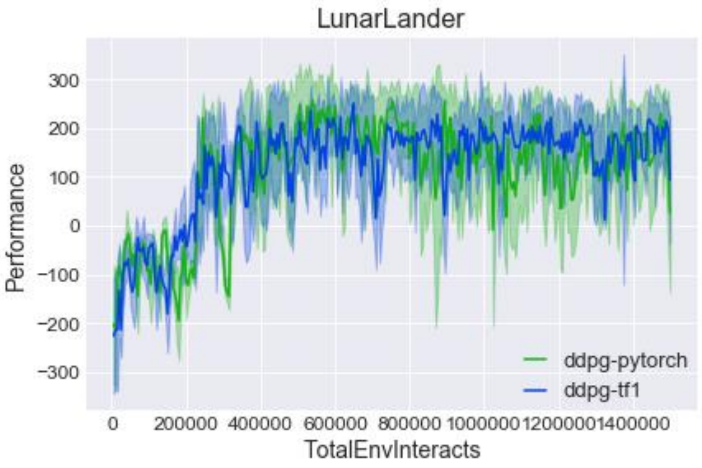




RESULTS OF SPINNINGUP CODE

Lunar Lander

Action Space	Discrete(4)
Observation Shape	(8,)
Observation High	[1.5 1.5 5. 5. 3.14 5. 1. 1.]
Observation Low	[-1.5 -1.5 -5. -5. -3.14 -5. -0. -0.]
Import	`gym.make("LunarLander-v2")`



CONCLUSIONS

What we have seen:

- The policy gradient theorem in stochastic and deterministic versions and how they are related
- How to insert the deterministic policy gradient theorem in DPG algorithms
- The DDPG algorithm based on DPG
- How DDPG algorithm performs on different environments and tasks
- The RỳŢộsçḥ implementation of DDPG from SpinningUp
- Our results using RỳŢộsçḥ and Ţêŋṣộsǧľộx models

REFERENCES

Paper:

[1] Timothy P. Lillicrap, et al. "Continuous control with deep reinforcement learning", ICLR (Poster) 2016

[2] Silver, David, et al. "Deterministic policy gradient algorithms", International conference on machine learning. Pmlr, 2014.

[3] https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Code:

[4] https://github.com/openai/spinningup/tree/master/spinup/algos/pytorch

APPENDIX

COMPARISON BETWEEN DDPG PSEUDOCODES

Algorithm 1 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 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 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

    end for
  end if
until convergence

MDP (REGULARITY CONDITIONS)

Regularity conditions for Theorem I of Deterministic Policy Gradient Theorem

A. Regularity Conditions

Within the text we have referred to regularity conditions on the MDP:

Regularity conditions A.1: $p(s'|s,a)$, $\nabla_a p(s'|s,a)$, $\mu_\theta(s)$, $\nabla_\theta \mu_\theta(s)$, $r(s,a)$, $\nabla_a r(s,a)$, $p_1(s)$ are continuous in all parameters and variables s, a, s' and x.

Regularity conditions A.2: there exists a b and L such that $\sup_s p_1(s) < b$, $\sup_{a,s,s'} p(s'|s,a) < b$, $\sup_{a,s} r(s,a) < b$, $\sup_{a,s,s'} ||\nabla_a p(s'|s,a)|| < L$, and $\sup_{a,s} ||\nabla_a r(s,a)|| < L$.

MDP (REGULARITY CONDITIONS)

Regularity conditions for Theorem 2 of Deterministic Policy Gradient Theorem

Conditions B1: Functions ν_σ parametrized by σ are said to be a *regular delta-approximation* on $\mathcal{R} \subseteq \mathcal{A}$ if they satisfy the following conditions:

1. The distributions ν_σ converge to a delta distribution: $\lim_{\sigma \downarrow 0} \int_{\mathcal{A}} \nu_\sigma(a', a) f(a) da = f(a')$ for $a' \in \mathcal{R}$ and suitably smooth f. Specifically we require that this convergence is uniform in a' and over any class $\mathcal{F}$ of L-Lipschitz and bounded functions, $||\nabla_a f(a)|| < L < \infty$, $\sup_a f(a) < b < \infty$, i.e.:

$$\lim_{\sigma \downarrow 0} \sup_{f \in \mathcal{F}, a' \in \mathcal{A}} \left| \int_{\mathcal{A}} \nu_\sigma(a', a) f(a) da - f(a') \right| = 0$$

2. For each $a' \in \mathcal{R}$, $\nu_\sigma(a', \cdot)$ is supported on some compact $\mathcal{C}_{a'} \subseteq \mathcal{A}$ with Lipschitz boundary $\mathrm{bd}(\mathcal{C}_{a'})$, vanishes on the boundary and is continuously differentiable on $\mathcal{C}_{a'}$.
3. For each $a' \in \mathcal{R}$, for each $a \in \mathcal{A}$, the gradient $\nabla_{a'} \nu_\sigma(a', a)$ exists.
4. Translation invariance: For all $a \in \mathcal{A}$, $a' \in \mathcal{R}$, and any $\delta \in \mathbb{R}^n$ such that $a + \delta \in \mathcal{A}$, $a' + \delta \in \mathcal{A}$, $\nu(a', a) = \nu(a' + \delta, a + \delta)$.

DQN ALGORITHM

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

DQN

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

DDPG

Algorithm 1 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 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