Deep Deterministic Policy Gradient

Table of Contents

Background

(Previously: Introduction to RL Part 1: The Optimal Q-Function and the Optimal Action)

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s, a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s, a).$$

DDPG interleaves learning an approximator to $Q^*(s, a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s, a)$.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s, a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s, a)$, we can approximate it with $\max_a Q(s, a) \approx Q(s, \mu(s))$. See the Key Equations section details.

Quick Facts

- DDPG is an off-policy algorithm.
- DDPG can only be used for environments with continuous action spaces.
- DDPG can be thought of as being deep Q-learning for continuous action spaces.
- The Spinning Up implementation of DDPG does not support parallelization.

Key Equations

Here, we'll explain the math behind the two parts of DDPG: learning a Q function, and learning a policy.

The Q-Learning Side of DDPG

First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right]$$

Here, in evaluating $(1-d)$, we've used a Python convention of evaluating `True` to 1 and `False` to zero. Thus, when `d==True` —which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state. (This choice of notation corresponds to what we later implement in code.)

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

Trick One: Replay Buffers. All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

We've mentioned that DDPG is an off-policy algorithm: this is as good a point as any to highlight why and how. Observe that the replay buffer *should* contain old experiences, even though they might have been obtained using an outdated policy. Why are we able to use these at all? The reason is that the Bellman equation *doesn't care* which transition tuples are used, or how the actions were selected, or what happens after a given transition, because the optimal Q-function should satisfy the Bellman equation for *all* possible transitions. So any transitions that we've ever experienced are fair game when trying to fit a Q-function approximator via MSBE minimization.

Trick Two: Target Networks. Q-learning algorithms make use of **target networks**. The term

$$r + \gamma(1-d)\max_{a'} Q_\phi(s',a')$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

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

where ρ is a hyperparameter between 0 and 1 (usually close to 1). (This hyperparameter is called `polyak` in our code).

DDPG Detail: Calculating the Max Over Actions in the Target. As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\text{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\right) \right)^2 \right],$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

The Policy Learning Side of DDPG

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s,\mu_\theta(s))\right].$$

Note that the Q-function parameters are treated as constants here.

Exploration vs. Exploitation

DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. Since the latter is simpler, it is preferred. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training. (We do not do this in our implementation, and keep noise scale fixed throughout.)

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions.

You Should Know

Our DDPG implementation uses a trick to improve exploration at the start of training. For a fixed number of steps at the beginning (set with the `start_steps` keyword argument), the agent takes actions which are sampled from a uniform random distribution over valid actions. After that, it returns to normal DDPG exploration.

Pseudocode

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

Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of DDPG in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However, we include both full docstrings for completeness.

Documentation: PyTorch Version

spinup.ddpg_pytorch(*env_fn, actor_critic=<MagicMock spec='str' id='140554320181456'>, ac_kwargs={}, seed=0, steps_per_epoch=4000, epochs=100, replay_size=1000000, gamma=0.99, polyak=0.995, pi_lr=0.001, q_lr=0.001, 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={}, save_freq=1*)

Deep Deterministic Policy Gradient (DDPG)

Parameters:

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** –
  The constructor method for a PyTorch Module with an `act` method, a `pi` module, and a `q` module. The `act` method and `pi` module should accept batches of observations as inputs, and `q` should accept a batch of observations and a batch of actions as inputs. When called, these should return:

Call	Output Shape	Description
`act`	(batch, act_dim)	Numpy array of actions for each observation.
`pi`	(batch, act_dim)	Tensor containing actions from policy given observations.

Call	Output Shape	Description
`q`	(batch,)	Tensor containing the current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to DDPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) –
  Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:
  $$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$
  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.

- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `ddpg_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

Documentation: Tensorflow Version

spinup.ddpg_tf1(*env_fn, actor_critic=<function mlp_actor_critic>, ac_kwargs={}, seed=0, steps_per_epoch=4000, epochs=100, replay_size=1000000, gamma=0.99, polyak=0.995, pi_lr=0.001, q_lr=0.001, 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={}, save_freq=1*)

Deep Deterministic Policy Gradient (DDPG)

Parameters:

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** –

A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Deterministically computes actions from policy given states.
`q`	(batch,)	Gives the current estimate of Q* for states in `x_ph` and actions in `a_ph`.
`q_pi`	(batch,)	Gives the composition of `q` and `pi` for states in `x_ph`: q(x, pi(x)).

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to DDPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) –
  Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:
  $$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$
  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.

- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`a`	Tensorflow placeholder for action input.
`pi`	Deterministically computes an action from the agent, conditioned on states in `x`.
`q`	Gives action-value estimate for states in `x` and actions in `a`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

References

Relevant Papers

- Deterministic Policy Gradient Algorithms, Silver et al. 2014
- Continuous Control With Deep Reinforcement Learning, Lillicrap et al. 2016

Why These Papers?

Silver 2014 is included because it establishes the theory underlying deterministic policy gradients (DPG). Lillicrap 2016 is included because it adapts the theoretically-grounded DPG algorithm to the deep RL setting, giving DDPG.

Other Public Implementations

- Baselines
- rllab
- rllib (Ray)
- TD3 release repo