TD3 Supplemental Material

Momin Haider

Algorithm

In order to understand the Twin Delayed DDPG algorithm, we have to start by understanding DDPG itself, which stands for Deep Deterministic Policy Gradient.

Deep Deterministic Policy Gradient (DDPG)

The fundamental goal of DDPG is the same as that of Q-learning and Deep Q-learning. It is to iteratively construct a function, $Q(s, a|\phi)$ with parameter vector ϕ, that can accurately approximate $Q^*(s, a)$, the optimal action-value function in some environment governed by a transition probability distribution, $P(\cdot|s, a)$. The optimal action-value function is defined as the discounted sum of future rewards starting from state s, taking action a, and thereafter following the optimal policy for the rest of the episode. Interestingly, $Q^*(s, a)$ can be defined in terms of a recursive relationship known as the Bellman equation (Eq. 1), where $r(s, a)$ is the immediate reward received by taking action a in state s, γ is the discount factor, s' is the state that follows s, and a' is the action taken in state s'.

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s' \sim P}\Big[r(s, a) + \gamma \max_{a'} Q^*(s', a')\Big] \tag{1}$$

With respect to the function approximator $Q(s, a|\phi)$, ϕ is a vector of parameters that defines the function and changes with successive iterations in the environment. In other words, as more training time goes by, $Q(s, a|\phi)$ should iteratively become a better and better approximator for $Q^*(s, a)$ (Eq. 2).

$$Q(s, a|\phi) \approx Q^*(s, a) \tag{2}$$

It is well known that if the optimal action-value function, $Q^*(s, a)$, in some environment is known, then the optimal policy in that environment is also known. Specifically, the optimal policy, $\mu^*(s)$, is to deterministically choose the action that maximizes the optimal action-value function (Eq. 3).

$$\mu^*(s) = \arg\max_a Q^*(s, a) \tag{3}$$

If $Q(s, a|\phi)$ is used to approximate $Q^*(s, a)$, the *locally* optimal policy can be defined as the action that maximizes the approximate action-value function, given some state, s (Eq. 4). Here, θ is a vector of parameters that defines the policy.

$$\mu(s|\theta) = \arg\max_a Q(s, a|\phi) \tag{4}$$

When dealing with discrete action spaces, the locally optimal policy, $\mu(s|\theta)$ is trivial to evaluate at each state: simply evaluate $Q(s, a_i|\phi)$ for all a_i where $1 \leq i \leq N$ and N is the cardinality of the action space and then choose the action that results in the maximum value of Q. In fact, such a policy would not even need to be dependent on an external parameter vector, θ. However, evaluating the locally optimal policy at each state, s, when dealing with continuous action spaces is highly nontrivial in general. Even in the one-dimensional case where a single action, a, is a real number bounded between $[a_{\text{low}}, a_{\text{high}}]$, determining the locally optimal policy at each step requires solving an optimization problem over the bounded region of possible actions. Such an optimization problem becomes increasingly difficult as the action-value function approximator becomes increasingly complex. Since $Q(s, a_i|\phi)$ is usually represented by a deep artificial neural network in practice, the aforementioned optimization subroutine would be completely infeasible if it had to be performed on each step through the environment.

Alternatively, what is done in DDPG is to train a deterministic policy, $\mu(s|\theta)$, *separately* from the approximate action-value function, $Q(s, a_i|\phi)$. To train the Q function approximator, it is intuitively appealing to minimize the mean squared Bellman error among a random sample of (s, a, r, s') transitions from a replay buffer, R (Eq. 5). It is advantageous to store (s, a, r, s') transitions in a large enough replay buffer such that when they are randomly sampled, there is little correlation between them to keep the training process stable. By convention, it should also be noted that $Q(s', a'|\phi) = 0$ for all a' whenever s' is a terminal state.

$$L(R, \phi) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim R}\left[\left(Q(s, a|\phi) - \left(r + \gamma \max_{a'} Q(s', a'|\phi)\right)\right)^2\right] \tag{5}$$

However, computing $Q(s, a|\phi)$ and the targets given by $y = r + \gamma \max_{a'} Q(s', a'|\phi)$ using the same parameter vector, ϕ, often leads to instability in practice. Therefore, DDPG makes use of *target* action-value and policy function approximators with separate parameter vectors, ϕ_{target} and θ_{target}, respectively. The target parameters slowly converge toward the original parameters, ϕ and θ, via slow weighted averaging (Eqs. 6-7). The averaging hyperparameter, $\tau \ll 1$, is usually between 0.001 and 0.01 in practice.

$$\phi_{\text{target}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{target}} \tag{6}$$

$$\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}} \tag{7}$$

With all this in mind, the mean squared Bellman error loss function used to train the action-value function approximator in DDPG is given below (Eqs. 8-9).

$$y(r, s') = r + \gamma Q(s', \mu(s'|\theta_{\text{target}})|\phi_{\text{target}}) \tag{8}$$

$$L(R, \phi) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim R}\left[\left(Q(s, a|\phi) - y(r, s')\right)^2\right] \tag{9}$$

While $Q(s, a|\phi)$ is trained to minimize the mean squared Bellman error loss, the deterministic policy, $\mu(s|\theta)$, is trained to maximize the expected action-value of states randomly sampled from R, given the current parameter vector ϕ.

The final DDPG algorithm is given below:

1. Initialize action-value function parameters ϕ, policy parameters θ, additive action noise σ, and an empty replay buffer R capable of storing a large amount of (s, a, r, s') transitions (say, one-million transitions).
2. Initialize $\theta_{\text{target}} \leftarrow \theta$ and $\phi_{\text{target}} \leftarrow \phi$.
3. For each episode:
   (a) Initialize the environment with state s.
   (b) For each step until episode termination:
      i. Given state s, choose action $a = \text{clip}(\mu(s|\theta) + \epsilon, a_{\text{low}}, a_{\text{high}})$ where $\epsilon \sim N(0, \sigma)$.
      ii. Execute a in the environment and observe the immediate reward r and next state s'.
      iii. Store the transition (s, a, r, s') in R.
      iv. Randomly sample a minibatch of N transitions from R.
      v. Compute the targets $y(r, s') = r + \gamma Q(s', \mu(s'|\theta_{\text{target}})|\phi_{\text{target}})$.
      vi. Update action-value function parameters ϕ via a step of minibatch gradient descent with respect to the loss $L(\phi) = \frac{1}{N}\sum_{N}(Q(s, a|\phi) - y)^2$.
      vii. Update policy parameters θ via a step of minibatch gradient descent with respect to the "loss" $L(\theta) = -\frac{1}{N}\sum_{N} Q(s, \mu(s|\theta)|\phi)$.
      viii. Update the target parameters with $\phi_{\text{target}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{target}}$ and $\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$.

To learn more about the DDPG algorithm, the reader is directed to the original DDPG paper [1].

Twin Delayed DDPG (TD3)

Given how DDPG works, TD3 is not much more difficult to understand, since the two algorithms are almost the same. TD3 just makes a few "tweaks" to improve performance on top of DDPG.

The first change that TD3 makes to the DDPG algorithm is with respect to the action-value function. In order to prevent overestimation biases from being too large, TD3 makes use of *two* action-value function approximators instead of one and takes the element-wise minimum between the two when evaluating targets. Since both Q-function approximators are optimizing for the same objective, they are referred to as "twins," as in the name, Twin Delayed DDPG.

The second change that TD3 makes to DDPG is that it delays updating the policy and target parameters: θ, $\phi_{\text{target-1}}$, $\phi_{\text{target-2}}$, and θ_{target}. This gives the Q-function parameters time to stabilize before being treated as constant when improving the policy parameters and updating the target parameters. This is why "delay" shows up in the name, Twin Delayed DDPG.

The last major change to DDPG is the addition of noise to the target actions before computing the targets in order to prevent overfitting to narrow peaks in the value estimate. The justification behind this is that similar (s, a) pairs should have similar action-value estimates.

The final TD3 algorithm is given below. Wherever the algorithm differs from DDPG, the text is **bolded**:

1. Initialize **action-value function parameters $\boldsymbol{\phi_1}$ and $\boldsymbol{\phi_2}$**, policy parameters θ, additive action noise σ_1, **additive target noise $\boldsymbol{\sigma_2}$**, **target clipping parameter $\boldsymbol{c}$**, and an empty replay buffer R capable of storing a large amount of (s, a, r, s') transitions (say, one-million transitions).
2. Initialize $\theta_{\text{target}} \leftarrow \theta$, $\boldsymbol{\phi_{\text{target-1}} \leftarrow \phi_1}$, **and** $\boldsymbol{\phi_{\text{target-2}} \leftarrow \phi_2}$.
3. For each episode:
   (a) Initialize the environment with state s.
   (b) For each step until episode termination:
      i. Given state s, choose action $a = \text{clip}(\mu(s|\theta)+\epsilon, a_{\text{low}}, a_{\text{high}})$ where $\epsilon \sim N(0, \sigma_1)$.
      ii. Execute a in the environment and observe the immediate reward r and next state s'.
      iii. Store the transition (s, a, r, s') in R.
      iv. Randomly sample a minibatch of N transitions from R.
      v. **Compute the target actions $\boldsymbol{a'(s') = \text{clip}(\mu(s'|\theta_{\text{target}}) + \text{clip}(\epsilon, -c, +c), a_{\text{low}}, a_{\text{high}})}$ where $\boldsymbol{\epsilon \sim N(0, \sigma_2)}$.**
      vi. Compute the targets $y(r, s') = r + \gamma \, \boldsymbol{\min}\,[\boldsymbol{Q(s', a'(s')|\phi_{\text{target-1}}), Q(s', a'(s')|\phi_{\text{target-2}})}]$.
      vii. **For both action-value functions**, update action-value function parameters

ϕ_i via a step of minibatch gradient descent with respect to the loss $L(\phi_i) = \frac{1}{N}\sum_N (Q(s,a|\phi_i) - y)^2$.

viii. **If enough steps have passed before the policy should be updated:**

A. Update policy parameters θ via a step of minibatch gradient descent with respect to the "loss" $L(\theta) = -\frac{1}{N}\sum_N Q(s, \mu(s|\theta)|\phi)$.

B. Update the target parameters with
$\boldsymbol{\phi_{\text{target-1}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{target-1}}}$,
$\boldsymbol{\phi_{\text{target-2}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{target-2}}}$, and
$\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$.

To learn more about the TD3 algorithm, the reader is directed to the original TD3 paper [2].

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[2] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.