Learning to Walk Using TATD3 an improved algorithm based on the TD3

Yifan Chen

Abstract

The aim of this experiment is to implement an improved algorithm, TATD3, based on the traditional TD3 algorithm and test its performance in the BipedalWalker environment. The implementation principle and advantages and disadvantages of this algorithm will also be analyzed.

1 Introduction

1.1 Environment

BipedalWalker-v3 is a deep reinforcement learning environment from OpenAI Gym. The main goal is to train a 4-joint walker robot to move through the environment as quickly as possible while using less energy. The normal environment is with slightly uneven terrain and the hardcore is with ladders, stumps, pitfalls. The state space is a vector composed of 24 continuous values, which represent the state information of the robot, including its position, velocity, and the angle and velocity of the joints. The action space has a size of four, corresponding to the two joints on each leg. Reward is given for moving forward, total 300+ points up to the far end and if the robot falls, it gets -100[1].

1.2 Algorithm

In the initial stage of the experiment, I used the traditional TD3 algorithm. I found that although the traditional TD3 algorithm could efficiently complete the training in the normal environment, it struggled in the hardcore environment. Therefore, I adopted an improved algorithm based on TD3 called TATD3[2]. Compared to the TD3 algorithm, TATD3 uses two actor networks, this can better improve exploratory behavior and avoid getting stuck in local optimum.

2 Methodology

Reinforcement learning is a machine learning method that learns optimal decisions through trial and error. It considers the paradigm of the interaction between an agent and its environment, and maximizes rewards by adjusting the agent's behavior. It should be noted that we should not only focus on immediate reward but also consider the cumulative future reward, more precisely, the cumulative discounted future reward. This is because the importance of immediate reward and future reward is different. The cumulative discounted reward can be represented as:

$$U_t = R_1 + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots \tag{1}$$

Where γ is discount factor. If we expand the equation and include states $s \in S$, actions $a \in A$, we can obtain:

$$U_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i) \tag{2}$$

U_t is a random variable that depends on all future actions and states. To evaluate the current situation more accurately, we can take the expectation of U_t to eliminate the random variable. Then we get a action-value function:

$$Q_\pi(s_t, a_t) = E[U_t | S_t = s_t, A_t = a_t] \tag{3}$$

This is the expected return obtained by taking action a_t at state s_t according to the policy π.
Q-learning algorithm is based on value function learning. The main idea is to learn a value function for a state-action pair and use an iterative approach to gradually update it, thereby continuously improving the agent's policy. In Q-learning, the value function can be learned based on the *Bellman equation* using the temporal difference learning algorithm.
According to Eq.(1) and Eq.(3):

$$U_t = R_1 + \gamma(R_{t+1} + \gamma R_{t+2} + \cdots) \tag{4}$$
$$Q_\pi(s_t, a_t) = r + \gamma E[U_t | S_t = s_t', A_t = a_t'] \tag{5}$$

Where the value a', s' presents the subsequent actions and states.
Eq.(5) represents that the value of the current state-action pair is equal to the current immediate reward plus the expected value of the values of all possible state-action pairs in the next time step. Therefore, we can find that the goal of Q-learning is to find the optimal policy by learning the optimal Q-value Q*:

$$Q^*(s_t, a_t) = E[r + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1} | S_t = s_{t+1}, A_t = a_{t+1})] \tag{6}$$

The value iteration formula is known as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s, a)) \tag{7}$$

For a large state space, the value can be estimated with a differentiable function approximator $Q_\theta(s, a)$, with parameters θ. The network is updated by using temporal difference learning with a secondary frozen target network $Q_{\theta'}(s, a)$ to maintain a fixed objective y over multiple updates[3]:

$$y = r + \gamma Q_{\theta'}(s', a') \tag{8}$$

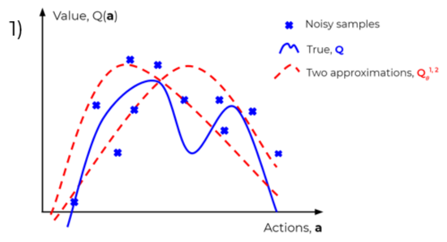
The weights of the target network are periodically updated through soft updates using the exponential moving average method. This ensures that the parameters of the target network θ' do not change too much and are only influenced slightly by the trained model θ. In our algorithm, both of the two target actor networks and two target critic networks are updated using this method:

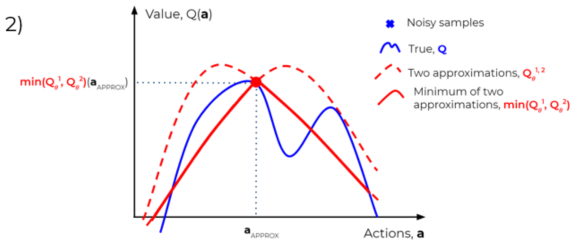
$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{9}$$

However, in Q-learning, the target y is easily affected by errors, such as randomness and noise, which may overestimate the value function of certain state-action pairs. This causes the agent to have a high estimate of these state-action pairs during the learning process, resulting in suboptimal decisions.
Denote the difference between approximation and the true Q-function as additive noise U and assume it has zero mean. By applying *Jensen inequality*, we can show that maximization of the approximate Q-function leads to overestimation with respect to the true Q-function[4].

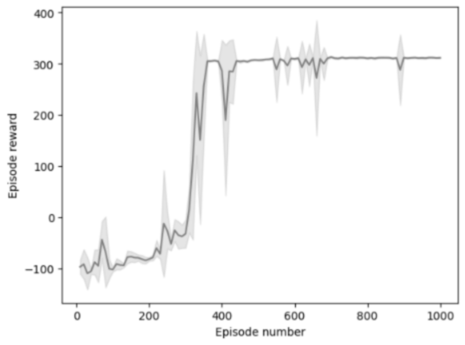
$$\underbrace{\mathop{\mathbb{E}}_{U}[\max_a\{Q_\theta(a)\}]}_{\text{predicted}} = \mathop{\mathbb{E}}_{U}[\max_a\{Q(a) + U(a)\}] \geq \max_a \mathop{\mathbb{E}}_{U}\{Q(a) + U(a)\} = \underbrace{\max_a Q(a)}_{\text{true}}$$

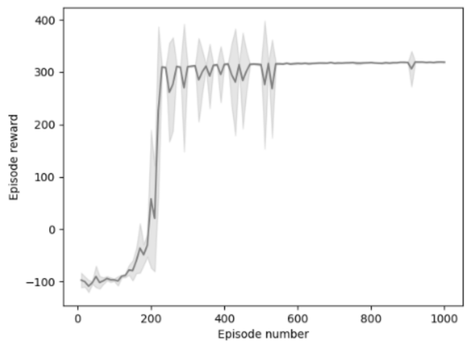
To address the overestimation bias problem, the TD3 algorithm uses a method called Clipped Double Q-learning[5]. This method trains two approximate value functions Q_θ^1 and Q_θ^2, and then uses the minimum of these approximations as the training target.




2)
Value, Q(a)
Noisy samples
True, Q
Two approximations, $Q_\theta^{1,2}$
Minimum of two approximations, $\min(Q_\theta^1, Q_\theta^2)$
$\min(Q_\theta^1, Q_\theta^2)(a_{APPROX})$
a_{APPROX}
Actions, a

Inspired by this method, we tried to introduce two actor networks as well and compared the improved algorithm with the single actor algorithm. The comparison showed that the TATD3 algorithm has a significant improvement compared to the TD3 algorithm.





From above figure, we can see that using two actor networks(**right**) performs better than using a single actor network(**left**), with a score surpassing 300 in fewer episodes and exhibiting better convergence. Our improved algorithm gradually stabilized from around 200 episodes and was close to fully converged in less than 600 episodes, showing excellent performance.

The detailed various steps of implementation of the algorithm are below:
Step 1. Initialize two actor networks π_{ϕ_1}, π_{ϕ_1}, and two critic networks Q_{θ_1}, Q_{θ_2}, initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi_1' \leftarrow \phi_1$ and $\phi_2' \leftarrow \phi_2$, initialize replay buffer β.
Step 2. Select action a_i from the actor network based on state s, and then add noise:

$$\begin{gathered} a_i = clip(\pi_\phi(s) + \epsilon, a_{max}, a_{min}) \quad i \in \{1,2\}, \\ \epsilon \sim clip(\mathcal{N}(0,\sigma), c, -c) \end{gathered} \tag{10}$$

Where a_{max} and a_{min} represent the upper and lower bounds of action, c is noise clip and σ is policy noise. Then select action a that maximizes the Q function:

$$a = \arg\max_{a_i} Q_{\theta_j}(s, a_i) \quad i,j \in \{1,2\} \tag{11}$$

Step 3. Execute action a, observe reward r and new state s' then store transition tuple (s,a,r,s') in β.
Step 4. Sample a batch of transitions from β.

$$\begin{gathered} \widetilde{a}_i = \pi_{\phi'}(s') + \epsilon \quad i \in \{1,2\}, \\ \epsilon \sim clip(\mathcal{N}(0,\sigma), c, -c) \end{gathered} \tag{12}$$

same as **step 2**, choose the $\widetilde{a}$ that maximizes $Q_\theta(s', a_i)$:

$$\widetilde{a} = \arg\max_{\widetilde{a}_i} Q_{\theta_j}(s', \widetilde{a}_i) \quad i,j \in \{1,2\} \tag{13}$$

Step 5. Use the minimum Q-value between $Q_{\theta_1'}(s', \widetilde{a})$ and $Q_{\theta_2'}(s', \widetilde{a})$ to calculates the target value and the *Loss*. The Q-values are updated through backtracking *Loss*. Update the critic network parameters θ_1 and θ_2 through stochastic gradient descent.

$$tv = r + \gamma \min Q_{\theta_j'}(s', \widetilde{a}), \quad j \in \{1,2\} \tag{14}$$

$$Loss = MSE(Q_{\theta_1}(s,a), tv) + MSE(Q_{\theta_2}(s,a), tv) \tag{15}$$

Step 6. Periodically update actor target networks by using gradient ascent on the Q-value of a critic network and the weights of the critic target network by using formula(9):

$$\phi'_i \leftarrow \tau\phi + (1-\tau)\phi'_i, \quad i \in \{1,2\} \tag{16}$$

$$\theta'_j \leftarrow \tau\theta + (1-\tau)\theta'_j, \quad j \in \{1,2\} \tag{17}$$

Step 7. Update state $s \to s'$.
Step 8. Repeat step 2-7 till $episode = episode_{max}$.

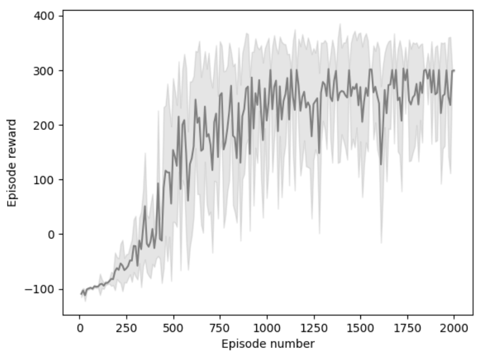
3 Parameter optimization and tuning

In this experiment, I not only tried and compared the TD3 and TATD3 algorithms multiple times but also attempted to optimize the hyperparameters in the algorithms. First, I adjusted the learning rate(lr). In the TD3 algorithm, we tried using 1e-3 as the default learning rate, but the performance of this parameter did not meet expectations in the TATD3 algorithm. Due to the higher exploratory behavior of TATD3 compared to TD3, this learning rate was too large for TATD3, which would cause the model to be unstable and difficult to converge in the later stages of training. Therefore, we finally set the learning rate to 5e-4.

We also made special adjustments for the BipedalWalker environment. From the documentation[1], we found that the reward for the robot falling is extremely high(-100), while other rewards may be less than 1. This would make the robot very afraid of falling, which is detrimental to future attempts and progress. For example, the robot may stop moving when the two legs are spread too far apart because it's afraid of falling, and it cannot try subsequent actions. Therefore, we adjusted the reward for falling from -100 to -5.

4 Limitation and Further work

Although the improved TATD3 algorithm performs much better than the TD3 algorithm, there are still some problems. For example, although we can maintain a score of around 300 in the hardcore environment, the model is still unstable and does not converge well.



Also, through experimentation, I found that the batch size has a greater impact on the model than I expected. We eventually chose 64 as the parameter for the normal environment and 128 as the parameter for the hardcore environment because the hardcore environment is more complex and requires richer transitions for training. However, the performance of 128 in the normal environment is not as good as 64, and this will be further explored in future work.

References

[1] Oleg Klimov. *Bipedal Walker*. `https://www.gymlibrary.dev/environments/box2d/bipedal_walker`.

[2] Tanuja Joshi et al. "Twin actor twin delayed deep deterministic policy gradient (TATD3) learning for batch process control". In: (Feb. 2021). arXiv: `2102.13012 [eess.SY]`.

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: `10.1038/nature14236`. URL: `https://doi.org/10.1038/nature14236`.

[4] Alexander Grishin Arsenii Kuznetsov. *Fixing overestimation bias in continuous reinforcement learning*. `https://research.samsung.com/blog/Fixing-overestimation-bias-in-continuous-reinforcement-learning`.

[5] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. eprint: `arXiv:1802.09477`.