Learning to Walk with improved TD3

Hanna Foerster

Abstract

This paper proposes to learn to walk by using the Twin Delayed Deep Determinisitic policy gradient (TD3) algorithm and improving on it. Several approaches to improving TD3 are: trying more efficient sampling methods, regulating the amount of noise, changing the reward function, increasing training time and increasing the frequency of policy updates. The improvement TD3-FORK brings is found to work best for training a bipedal walker, but other methods are compared against this. Finally, the reason why FORK is the best is explained in an analytical way.

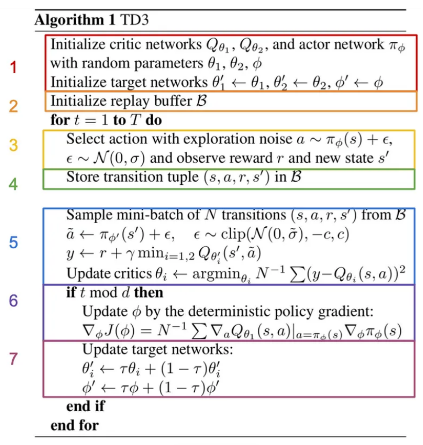
1 Methodology

1.1 Environment

This paper proposes training a bipedal robot to learn to walk by using the Twin Delayed Deep Determinisitic policy gradient (TD3) algorithm [3] and improving upon it. The environment[9] is made up of a simple 4-joint walker that must walk on a normal slightly uneven terrain in the basic environment and an environment with ladders, stumps and pitfalls in the hardcore version. The action space is continous between -1 and 1 for the motoric movement for each of the 2 leg joints that make up each leg. The goal of the environment is to get to the end of the environment in as small number of steps as possible without falling. 300+ points are awarded if this is achieved in a minimum number of steps.

1.2 TD3

TD3 [3] is a deep reinforcement algorithm that is adapted from the Deep Deterministic Policy Gradient (DDPG) [7] algorithm. DDPG is a model-free, off-policy RL algorithm that learns a Q-function (function that estimates reward from a given state and action) by using off-policy data and the Bellman equation to learn a policy. DDPG employs an actor-critic agent where the critic is used to estimate the Q-values and an actor is used to update the policy parameter δ in the direction suggested by the critic to maximize the expected cumulative long-term reward. TD3 improves DDPG by using a pair of critic networks instead of just one, delaying updates of the actor and adding action noise regularization, which addresses the overestimation bias in the critic and actor. The twin critic networks help in reducing bias for estimating the Q-value, by always taking the smaller value of the two critic networks, preventing overestimation of values which can cause destabilization by propagation through the algorithm. An overestimation in policy can similarly harm the agent's policy in a vicious cycle of erroneous updates between the actor and critic. To fix this, TD3 carries out less frequent updates (usually updates every 2 timesteps) for the actor compared to the critic, making it more resistant against individual outliers. The final improvement is to add noise to the target. Because of the deterministic nature of the actor the estimated actions for states have a high variance



```
Algorithm 1 TD3
1   Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$
    with random parameters $\theta_1, \theta_2, \phi$
    Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
2   Initialize replay buffer $\mathcal{B}$
    for $t = 1$ to $T$ do
3       Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
        $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
4       Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

5       Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
        $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
        $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
        Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
6       if $t$ mod $d$ then
            Update $\phi$ by the deterministic policy gradient:
            $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
7           Update target networks:
            $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
            $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
        end if
    end for
```

Figure 1: TD3 algorithm [2]

and a high tendence to overfit at spikes. Adding noise to the value estimates stabilizes the policy because actions that are more robust to noise survive.

The detailed algorithm can be found on the right side. After initializing critic, actor, target networks and the replay buffer, the algorithm consists of two parts. One is based on real experience, where for a given state an action with noise is carried out in the environment and the outcome of the action from the state is stored in a replay buffer. The other is the training part based on past experience stored in the replay buffer. A batch of past experiences is sampled, and the actor first estimates an action with its current policy. Then, the critic computes the reward for the state with the actually experienced action stored in the buffer and the reward for the state with the estimated action from the policy. These are then compared to each other with a loss function and the critic learns from this. The policy is then updated every two runs to update the policy according to the critic.

1.3 TD3 improvements

There are several points where TD3 can start being improved, such as sampling, the amount of noise, the reward function, training time and frequency of policy updates. The first is sampling. This can be improved either by determining which kind of experiences are unhelpful or harmful to training and disincluding this data in the buffer or by storing all experiences in the buffer, but sampling only the ones with highest training benefit. There are several methods that I have explored that include the Prioritized Experience Replay (PER) [10], the Likelihood Free Importance Weights (LFIW) [11], DisCor [5] + Regret Minimization Experience Replay in Off-Policy RL [8] and FORK [12]. First, PER samples the experiences with priority that present the highest temporal-difference error in order to down-weigh experiences that are encountered often to give a more uniform distribution over the experience sampling. This error is made up of the discounted estimated value for the next time step subtracted by the current estimate of the value and the reward gained from the transition between this and the next step: $r_{\mathrm{t}+1} + \gamma * V(s_{\mathrm{t}+1}) - V(s_{\mathrm{t}})$. However, in [11], Sinha et al. argue that this method might not work well on actor-critic methods where the goal is to learn the q-function attained from the current policy, so following off-policy experiences might not give the best results. They argue that it is more beneficial to perform importance sampling that reflects on-policy experiences instead. In LFIW they propose a method that reweighs the importance of experiences based on their likelihood from the present policy. The importance weights are calculated from the likelihood-free density ratio estimator between on-policy and off-policy experiences.

Further, Kumar et al [5] argue that in Q-learning algorithms which learn on a memory buffer, on-policy data collection which provides corrective feedback might not be sufficient to correct errors in the Q-function. Training on collected experience can give noisy and poor results depending on the distribution of it. DisCor (Corrective Feedback in RL via distribution correction) is proposed to prevent this from happening by approximating the optimal distribution and using it to reweigh the samples used in training. Liu et al. [8] note that data with higher hindsight TD error, more on-policiness and higher accuracy in Q-value should be prioritized when sampling. ReMERN which learns an error network and ReMERT which exploits the temporal ordering of states are introduced as solutions to regret minimization. (Here regret minimization refers to the difference between the cumulative reward obtained by following a learned policy and the cumulative reward that would have been obtained by following the optimal policy.)

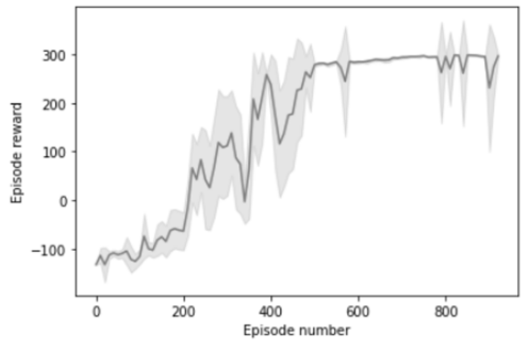
Lastly, in td3-fork [12], where fork stands for "a forward-looking actor for model-free RL" the policy predicts the expected reward for a given state-action pair by looking ahead for more than just one step. This is done by training a "system network" which estimates the next state given the current state and action: $\tilde{s}_{\mathrm{t}+1} = F_\theta(s_{\mathrm{t}}, a_{\mathrm{t}})$. The system network is used when updating the policy network by calculating the actor loss not only for the current timestep, but for two timesteps ahead and adding this with decay to the current actor loss and training: $L(\phi) = E[-Q_\psi(s_{\mathrm{t}}, A_\phi(s_{\mathrm{t}})) - \gamma * Q_\psi(\tilde{s}_{\mathrm{t}+1}, A_\phi(\tilde{s}_{\mathrm{t}+1})) - 0.5 * \gamma * Q_\psi(\tilde{s}_{\mathrm{t}+2}, A_\phi(\tilde{s}_{\mathrm{t}+2}))]$ (This formula is different from the one in the paper, but taken from how the code for bipedal walker is written.) TD3-FORK does well on the bipedal walker environment because of its specific adjustments for the environment. For example, the reward is adjusted before the experiences are saved into the replay buffer. Instead of adding -100 when the walker falls down, only -5 is added to the cumulative reward and for all other rewards 5*reward is added. This skews the experiences, such that gaining a small reward is not much better than falling down. Thus, we can see in training time that the agent keeps falling until finding an optimal first step. After finding this the walker starts walking optimally in a very small timestep interval. Furthermore, the number of training steps after every real run in the environment is adjusted on the reward attained in that run. There is more training when the walker falls down, when the episode reward is less than an acceptable value and sometimes at random. For the episodes, where more training is done, their experience is added to the replay buffer. However, for all others they are not added. This gives us more samples where the walker falls down and is not performing that well yet. It does not store samples where the walker did not finish in the environment in a maximum number of steps, thus disregarding the walker if it just stands around.

2 Convergence Results

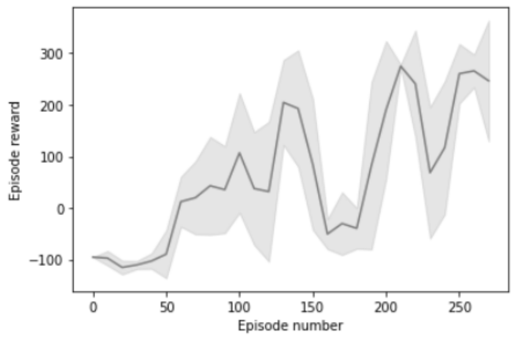
I have used following code bases for my code: [6] [13] [4] [1]

DisCor + regret minimization experience replay in off-policy reinforcement learning was implemented on the SAC algorithm in [13]. I rebuilt this to test PER, LFIW, and DisCor + regret minimization experience replay and a combination of them on TD3, but in the end the method td3-fork that I found won against all of them in performance. I have still attached the other code below my td3-fork code. I will now go in order through all methods and present some results for comparison.

Compared to vanilla td3, td3-PER indeed sped up the training process, with my best results after tuning the function that adds noise in terms of steps done in the environment being convergence after about 500 episodes in the basic environment. After experimenting with combinations of DisCor, lfiw, regret minimization and PER, I found that DisCor with lfiw and regret minimization gave me the best results. Here I started getting scores of over 100 at episode 67. However, then I found that after achieving a score of about 260 in episode 130 the reward did not get better that fast. They slowly snailed to 290 in another 170 episodes, but even after tweaking never reached 300 or above.



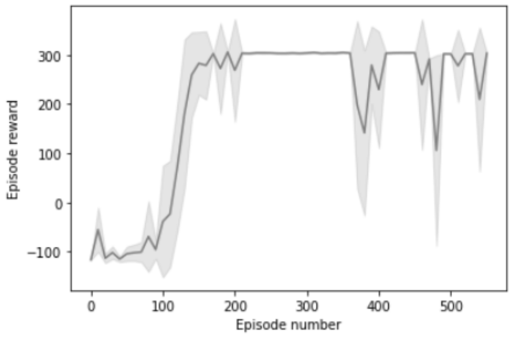
(a) td3-PER



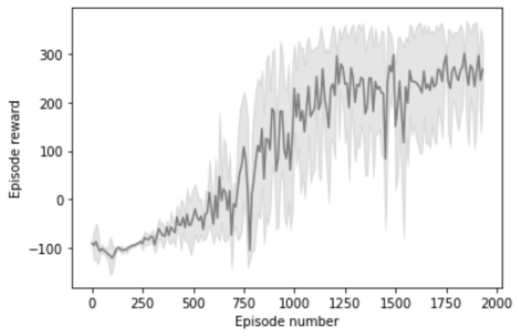
(b) td3-DisCor-lfiw-regretMin

After finally trying td3-fork instead, I analysed the reason for the above failure to converge in an optimal score. There is a fundamental difference in the learning approach of fork and the other methods. This is that in fork the algorithm first looks for the optimal first step that is both far-reaching and stable before starting to walk further. This is why in td3-fork we can see the reward suddenly going from -70 to near 300. In the other algorithms a stable first step is found and soon after the agent starts trying to walk as far as possible, not trying to optimize the step until reaching the goal. In this way the agent has a hard time correcting the stable but slow stepping method to a faster stable stepping method. I also tried adjusting the other methods to incorporate some of the adjustments I found in the td3-fork code, such as changing the rewards for the replay buffer or changing training time according to above mentioned criteria, however this did not work well for the other methods.

A similar tradeoff can also be found in td3. Adjusting parameters such as noise and the frequency of policy updates we can get the agent to start having positive rewards way earlier, but with the tradeoff that the agent starts walking without having found an optimal first step. Since the agent can achieve rewards of a lot higher than 300 in the basic environment (my highest was 318), there is a tradeoff to make between sampling efficiency and best results. If the agent starts walking faster then the agent might have not learned the optimal step, but a good enough step to reach 300. If noise is left high later on, the training continues for longer, with rewards dropping when the agent is experimenting, thus taking longer for the agent to converge to a score of over 300, but in the end achieving a score of much higher than 300. I am submitting results (episode text file and video) which were unseeded and achieved high rewards by the end of training but a slower convergence than if they converged on a lower 300 value. In the basic environment I got results with convergence on about 315 after episode 200 with the first episode over 300 at episode 131. However, since the results of runs differ greatly between runs, for reproducibility I am submitting the code and graph with a seed that performed a little better in terms of sample efficiency (first episode above 300 is at 118) but converged on a lower value of about 303. For the hardcore environment the seeded environment converged way better at about episode 1000, but only on a value of exactly 300 or 301. The unseeded environment did not converge that well, but produced episodes of above 310 after about the same number of episodes, but with a better gait, so I am submitting the unseeded results. The first episode above 300 was at episode 676 for the hardcore unseeded run. I also adjusted the training time in td3-fork to 10 times its orginally proposed training time, making the algorithm about twice as sample efficient, but a lot slower. Please lower this training time if you have time constraints for training.



(a) td3-FORK-basic-env-seeded



(b) td3-FORK-hardcore-env-seeded

3 Limitations

For sample efficiency purposes I increased training time by 10, but this makes especially the hardcore environment very slow to train. Moreover, the hardcore environment only starts achieving scores above 300 at about episode 1000 and only converges much later.

Future Work

It would be interesting to play around more with the different methods and different parameters to try and achieve better results. Manipulating randomness for better results would also be interesting, since I saw very different results with just small changes to the randomness.

References

[1] Robotic AI Learning Lab Berkeley. *rlkit-gaussian strategy*. URL: https://github.com/rail-berkeley/rlkit/blob/master/rlkit/exploration_strategies/gaussian_strategy.py. (accessed: 04.02.2023).

[2] Donal Byrne. *TD3: Learning To Run With AI*. URL: https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93. (accessed: 04.02.2023).

[3] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1587–1596.

[4] MetaRL HRL Memory Graphs. *RL-adventure*. URL: https://github.com/higgsfield/RL-Adventure-2/blob/master/6.td3.ipynb. (accessed: 04.02.2023).

[5] Aviral Kumar, Abhishek Gupta, and Sergey Levine. *DisCor: Corrective Feedback in Reinforcement Learning via Distribution Correction*. 2020. DOI: 10.48550/ARXIV.2003.07305. URL: https://arxiv.org/abs/2003.07305.

[6] leiying honghaow leiying. *TD3-FORK*. URL: https://github.com/honghaow/FORK. (accessed: 04.02.2023).

[7] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[8] Xu-Hui Liu et al. *Regret Minimization Experience Replay in Off-Policy Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2105.07253. URL: https://arxiv.org/abs/2105.07253.

[9] OpenAIGym. *Bipedal Walker*. URL: https://www.gymlibrary.dev/environments/box2d/bipedal_walker/. (accessed: 04.02.2023).

[10] Tom Schaul et al. *Prioritized Experience Replay*. 2015. DOI: 10.48550/ARXIV.1511.05952. URL: https://arxiv.org/abs/1511.05952.

[11] Samarth Sinha et al. *Experience Replay with Likelihood-free Importance Weights*. 2021. URL: https://openreview.net/forum?id=ioXEbG_Sf-a.

[12] Honghao Wei and Lei Ying. *FORK: A Forward-Looking Actor For Model-Free Reinforcement Learning*. 2020. DOI: 10.48550/ARXIV.2010.01652. URL: https://arxiv.org/abs/2010.01652.

[13] Toshiki Watanabe Zhenghai Xue. *Experiments on Discor*. URL: https://github.com/AIDefender/MyDiscor. (accessed: 04.02.2023).