Learning to Walk - Twin-Delayed DDPG

Anonymous author

Abstract

This paper proposes to train a bipedal robot to learn to walk in a 2D physics simulation via the implementation of a Twin-Delayed Deep Deterministic Policy Gradient (DDPG) algorithm, referred to as TD3. [1]. The methodology describes the underpinning theory of the DDPG algorithm, before outlining how TD3 extends upon this by addressing some of its possible flaws. The convergence results of the implemented method on the BipedalWalker-v3 environment are displayed. Some limitations of this method are also outlined, alongside some suggestions on what could be done in the future to improve the method and optimise results on this environment.

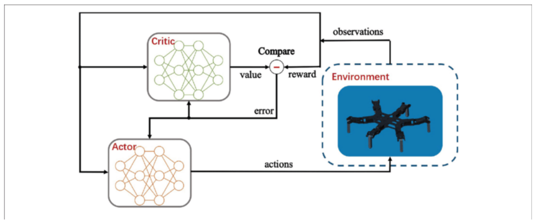
1 Methodology

This paper proposes training a bipedal robot to learn to walk in the BipedalWalker-v3 environment, which has 24 observations. The agent can take 4 actions but the action space is continuous: (referenced here):

	Name	**Min**	**Max**
0	Hip 1 (Torque/velocity)	-1	$+1$
1	Knee 1 (Torque/velocity)	-1	$+1$
2	Hip 2 (Torque/velocity)	-1	$+1$
3	Knee 2 (Torque/velocity)	-1	$+1$

Table 1: The continuous action space is between -1 and $+1$.

The baseline DDPG algorithm uses 4 neural networks: a state-action value network, a deterministic policy function, a target state-action value network, and a target policy network [4]. The state-action network and policy network are based on an actor-critic model [5]. The actor is a neural network that aims to learn the optimal policy, so that it can take a state as input and output the exact best action for that state (via function approximation) [3]. The critic (which is a separate, independent model) has a similar architecture; its role is to evaluate this action and output the advantage value given an environment and an action provided by the actor as input. The advantage value is a component of the state-action value, which determines how much better a particular action is compared to the other possible actions at a given state. It is this value that is learned by the critic as opposed to just the state-action value - the algorithm knowing the degree to which a particular action outperforms the others should reduce any high variance in the actor and elicit model stability. Both the actor and the critic play a game where they both improve at their roles as time goes by. Gradient ascent is used to update both models' weights at each step. The following diagram displays an architectural overview of this structure [9]:



DDPG is designed for use in environments with continuous action spaces such as BipedalWalker-v3, whereby it is not feasible for a traditional optimal action-value function ($Q^*(s,a)$) to be used to determine the optimal action in a given state; the computational cost of evaluating the entire action space would be too large when the number of actions are not finite and discrete [8]. However, $Q^*(s,a)$ is differentiable with respect to the action argument in continuous action spaces, so a gradient-based learning rule can be established. This will enable the optimal action in a given state, $max_a Q(s,a)$, to be approximated. Let the approximator to $Q^*(s,a)$ be the neural network $Q_\phi(s,a)$ with parameters ϕ. The algorithm uses a replay buffer: a record of previous experiences, which will be used within $Q^*(s,a)$ to update neural network parameters. This is denoted as the set D of transitions (s,a,r,s',d), whereby d takes a value of 1 if the next state s' is a terminal state, and 0 otherwise. Random batches of experience from this replay buffer are then sampled whenever the value and policy networks are updated. This ensures that data is independently distributed, which is necessary for optimisation problems; the data would not be independent when trying to optimise a sequential decision process in an on-policy way [10]. The expression $r + \gamma(1-d)Q_{\phi targ}(s', \mu_{\theta targ}(s'))$ denotes the target network, which $Q^*(s,a)$ is attempting to adapt to. If the target were to use (and thus be interdependent on) the same parameters, ϕ, that the method is trying to train on, the minimisation of the error loss would be unstable and prone to divergence. The target network is therefore established with its own parameters ϕ_{targ}, which are similar to ϕ but with a time delay, given by:

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

This is carried out every time the main network updates. ρ is a hyper-parameter which takes a value between 0 and 1 and requires tuning; the implemented method uses a value of 0.995. All this put together, the learning is ultimately accomplished by minimising the following mean-squared Bellman error (MSBE) loss with stochastic gradient descent:

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

where $\mu_{\theta targ}$ denotes the target policy network. $Q^*(s,a)$ is then used to learn the optimal policy $\mu_\theta(s)$ - i.e. a deterministic policy function which outputs whichever action maximises $Q_\phi(s,a)$ [6]. This can be achieved by executing gradient ascent with respect to the policy parameters to solve

$$\underset{\theta}{max} \underset{s\sim D}{E}[Q_\phi(s, \mu_\theta(s))]$$

based on the assumption that $Q^*(s,a)$ is differentiable with respect to the action argument, since the action space is continuous.

The implemented method, TD3 (code based from [7]), is based on this algorithm, but adjusted slightly in order to address some of the possible issues that can occur in baseline DDPG. One such specific error occurs when the approximator to $Q^*(s,a)$ establishes (incorrectly) a sharp peak for some actions; the policy will then swiftly exploit this peak, leading to it having incorrect and brittle behaviour down the line. To mitigate this, TD3 uses randomly sampled noise to 'smooth out' $Q^*(s,a)$ over similar actions. Rather than just using the mere output from the target policy $\mu_{\theta targ}$ to produce an action, clipped noise (the bounds for which is given by a hyper-parameter c that requires tuning; it is set to 0.5 in the implemented method) is additionally added on each action dimension. Once this has been added, the resulting action is then clipped such that it lies in the range of valid actions. This ultimately acts to regularise the algorithm. The final target actions are given by:

$$a'(s') = clip(\mu_{\theta_{targ}}(s') + clip(\varepsilon, -c, c), a_{Low}, a_{High}), \epsilon \sim N(0, \sigma)$$

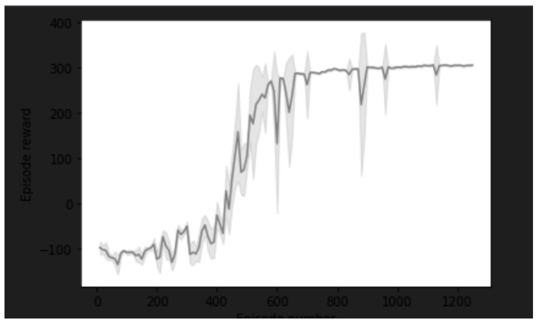
Furthermore, two optimal action value functions Q_{ϕ_1} and Q_{ϕ_2} are simultaneously learned rather than just one. The smaller of the respective target values returned by each function is then used as the $Q_{\theta targ}$ value within the MSBE loss function described before (for both functions), to perform the functions' learning. Ensuring regression in learning is aimed towards the smaller target value will help mitigate possible over-estimation in $Q^*(s,a)$, which would be very harmful, as it would cause the policy to start exploiting errors in $Q^*(s,a)$ and consequently stop working as intended.

The optimal policy learning is the same as described in baseline DDPG (with the intention of maximising one of the optimal action value functions, say, Q_{ϕ_1}). But in TD3, the policy is updated less frequently than Q_{ϕ_1} and Q_{ϕ_2} are. The rationale behind this is that it should result in a mitigation of the volatility that can occur when a policy update changes the target, so it makes sense to ensure the former is less frequent relatively speaking. The less frequent, the larger the benefit in terms of error accumulation. But this would then have implications for the network's learning, so in the implemented method, the policy is updated every other update of Q_{ϕ_1} and Q_{ϕ_2}.

The implemented method has a few further specific configurations aimed at optimising results. A 'soft start' is implemented, whereby during the first 18000 timesteps (measured cumulatively across episodes), the agent simply selects actions at random and does not train on the corresponding transition data. This helps improve exploration at the start of training by improving outreach at a time where the learning would be too undeveloped to be effective. After some experimentation, the value of 18000 was selected; this seemed to trigger natural convergence to begin at a fast rate after 18000 timesteps had elapsed. The replay buffer is configured such that it can hold 1500000 sets of state transition data. It is necessary for it to be very large so that it contains a diverse set of experiences - if it only contained data of very similar nature, it might overfit to this. To this regard, Fujimoto et. al. use the replay buffer to store the entirety of the agent's experiences. However, when attempting to replicate this in the implemented method and using very large replay buffer sizes, this then caused learning to slow down, so 1500000 was the selected middle ground. Although Fujimoto et. al. recommend a learning rate of 0.001 to be used for the Adam optimiser to update the network parameters, this did not work well in the method. There were heavy signs of divergence behaviour in the function suggesting that a smaller learning rate would be required; the method uses 0.0003 instead.

2 Convergence Results

The diagram below shows the episode reward vs episode number graph, when the method has been run for 1250 episodes (note the axis are labelled, they are just difficult to see). Convergence behaviour is exhibited by the method; after around 580 episodes, rewards of just under 300 are consistently hit; after around 780 episodes, rewards in and around the 300 mark are consistently hit. The method appears to be very stable because when running it several times, it always converges and never gets stuck in local minima. When inspecting the videos of the trained agent corresponding to the episodes where the reward is high, the agent is running quickly and navigating the terrain effectively. It is not showing any signs of undesirable behaviour. However, there do appear to be intermittent fluctuations in episode rewards (such as the one around episode 900) which are analysed in the next section.



3 Limitations And Future Work

By its nature, the TD3 algorithm is quite brittle with respect to its hyper-parameters and other aspects which require tuning. This, along with the fact that training time is so long (and it is therefore extremely expensive from a time standpoint to trial too many changes to hyper-parameters and examine the results), meant that it felt like the method was restricted to some degree in terms of experimentation of hyper-parameter tuning beyond what has already been documented to work well in the cited code and research literature. Thus, it was difficult to properly experiment and get the most out of the algorithm - there might have been potential improvements that were missed out on as a result. Future work would involve investing more time into experimenting with different hyper-parameter values.

In the convergence graph above there are intermittent drops in episode reward value. One potential cause of this could be the Gaussian noise that is currently added to actions during training in order to help the policy explore better. This is only really beneficial in the earlier stages when it might not attempt a wide enough variety of actions, given that it is exploring on a deterministic policy. Once the agent has been trained to a strong degree, it might be worth gradually scaling down this noise as time goes on so that it is no longer interfering when the actions given by the policy are accurate enough. This might reduce the levels of fluctuation in episode rewards, especially after many episodes have elapsed when this is undesirable.

Future work could involve the implementation of a different algorithm, such as a model based method, to get an even faster convergence speed and ultimately better results [2]. Such a method would learn a model (trained easily using supervised learning) from agent experience and use the model to plan the value function and/or policy from the model. The model would act as a simpler and more useful representation of the environment which would otherwise have to be accessed via direct interaction. This would allow the agent to make more intelligent decisions within its action choices, because it will have been able to predict future states and rewards.

References

[1] 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.

[2] Lukasz Kaiser et al. *Model-Based Reinforcement Learning for Atari*. 2020. arXiv: `1903.00374 [cs.LG]`.

[3] Sergios Karagiannakos. *The idea behind Actor-Critics and how A2C and A3C improve them*. 2018. URL: `https://theaisummer.com/Actor_critics/`.

[4] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[5] Chien-Liang Liu, Chuan-Chin Chang, and Chun-Jan Tseng. "Actor-Critic Deep Reinforcement Learning for Solving Job Shop Scheduling Problems". In: *IEEE Access* 8 (2020), pp. 71752–71762. DOI: `10.1109/ACCESS.2020.2987820`.

[6] David Silver et al. "Deterministic Policy Gradient Algorithms". In: ICML'14. Beijing, China: JMLR.org, 2014, I–387–I–395.

[7] soumik12345. *Twin Delayed DDPG*. 2020. URL: `https://github.com/soumik12345/Twin-Delayed-DDPG`.

[8] OpenAI Spinning Up. *Deep Deterministic Policy Gradient*. 2018. URL: `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[9] Ouyang W et al. "Adaptive Locomotion Control of a Hexapod Robot via Bio-Inspired Learning". In: *Front Neurorobot.* 2021.

[10] Chris Yoon. *Deep Deterministic Policy Gradients Explained*. 2019. URL: `https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b`.