Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3) Implementation and Comparative Analysis

Pau Hidalgo Pujol
APRNS @GIA UPC

November 24, 2024

Abstract

Value based algorithms, like DQN, are able to learn with sample efficiency but only work on environments with a discrete set of actions. Methods like VPG and A3C are't as efficient, but can deal with continuous control tasks. The end goal, then, is to be able to have sample-efficient and stable algorithms that can be applied to environments with continuous actions. This work compares Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3), evaluating their performance in the LunarLander-v2 (continuous) environment. DDPG extends value-based methods to continuous spaces, while TD3 introduces key improvements to mitigate DDPG's instability and overestimation issues. The results show TD3 significantly outperforms DDPG in terms of stability, learning efficiency, and final learned policy. A discussion on the implementation challenges, hyperparameter sensitivity, and practical observations is included. The report concludes with suggestions for future improvements, other possible comparisons and and broader applicability of the algorithms.

1 Introduction

Continuous control tasks present unique challenges for reinforcement learning (RL) algorithms, as traditional value-based methods like Deep Q-Networks (DQN) [7] are limited to discrete action spaces. Policy-based methods such as Vanilla Policy Gradient (VPG) and Advantage Actor-Critic (A3C) are able to extend RL to continuous spaces but are sample-inefficient (in part, due to their on-policy nature).

Deep Deterministic Policy Gradient (DDPG) [6] combines deterministic policies with off-policy learning to address these issues. However, DDPG is very sensitive to hyperparameters, prone to overestimation errors, and can be unstable during training. Twin Delayed DDPG (TD3) [3] improves upon DDPG with innovations aimed at stabilizing learning and improving performance.

This report documents the implementation of DDPG and TD3, evaluates their performance on the LunarLander-v2 (continuous) environment, and provides a detailed comparison of their strengths, limitations, and practical considerations.[1]

2 Algorithms and Implementation

2.1 Deep Deterministic Policy Gradient (DDPG)

DDPG extends the DQN framework to continuous action spaces by using an actor-critic architecture. The critic estimates Q-values for state-action pairs, while the actor learns a deterministic policy to maximize these Q-values. Key components include:

- **Replay Buffer:** Facilitates off-policy learning by storing experience tuples (s, a, r, s').

[1] Note: In the latest Gym API, the environment is updated to LunarLander-v3.

- **Actor-Critic Networks:** Custom architecture with batch normalization and weight initialization as per [6].
- **Exploration via Ornstein-Uhlenbeck Noise:** Adds temporally correlated noise to actions for efficient exploration.
- **Soft Target Updates:** Stabilizes training by slowly updating target networks.

Programming the Replay Buffer was easy, since we could reuse the DQN one. Since the ActorCritic networks architectures where also already defined, initiating the networks was also straighforward. Note that, since DDPG uses target networks, we had to use the copy target function so both the network and the taraget start with the same parameters.

Choosing the action consisted simply in using the Actor network to predict it given a state. Computing the critic loss was a bit more difficult, but we could also reuse most of the DQN code. The main difference was that we didn't need to unsqueeze the actions (since in our case they aren't discrete). The computations where done following the algorithm, and taking into account that the target networks shouldn't be added to the gradient graph. Our implementation reference was OpenAI spinning up Github, which was also done in Pytorch [8].

Computing the Actor loss was done using -torch.mean, with the Critic and Actor networks, and to apply the gradients we simply used the optimizers defined before (with backward() and step()). To do the soft updates we also had a helper function so we used that one.

2.2 Twin Delayed DDPG (TD3)

TD3 introduces three key improvements over DDPG:

- **Twin Critics:** Reduces overestimation bias by using two Q-value networks and selecting the smaller value during target computation.
- **Clipped Gaussian Noise:** Adds bounded noise to actions for better exploration.
- **Delayed Actor Updates:** Stabilizes learning by updating the actor less frequently than the critic.

Before talking about the implementation, we would like make a clarification about the Clipped Gaussian Noise. In the Lab instructions, it seemed that the noise they Clipped was the one used for exploration, during training, in the same place where DDPG used noise. However, looking at the TD3 paper [3], in Section 5.3, we saw that in fact TD3 added noise in the Target Policy (it acts as a regularization to reduce variance) when updating the Critic. This noise, also Normal, was the one clipped: the exploration noise wasn't.

Having considered that, we implemented the code: we created another critic and target (which also required modifying the optimizer), and changed the exploration noise from Ornstein-Uhlenbeck to Gaussian, since the TD3 paper found that it offered no performance benefits. We also added the clipped noise in the compute critic loss method, and modified it to use the minimum of two critics to reduce overestimation (the loss was still computed for both Critics, but the target was the same). The delayed Actor updating was also easy to implement, using simply a modulus to reduce the frequency.

An important thing to note is that our implementation still wasn't an exact match of the official one, since they used a purely exploratory policy for the first time steps to remove the dependency on the initial parameters. This can be confirmed looking at [1, 9] (the author's Pytorch implementation, and spinning up, our references). However, we felt that this little change wasn't really that important and that probably it's effect in our environment wasn't as important as the other modifications.

3 Experimental Setup

3.1 Environment

The LunarLander-v2 (continuous) environment was used for the experiments, featuring continuous control over the main and lateral engines with actions in $[-1, 1]$.

Hyperparameter	Algorithm		
	DDPG	**TD3**	**DDPG-v2**
Actor Learning Rate	10^{-4}	10^{-3}	10^{-3}
Critic Learning Rate	10^{-3}	10^{-3}	10^{-3}
Batch Size	64	100	100
Replay Buffer Size	10^6	10^6	10^6
Target Network Update Rate (τ)	0.001	0.005	0.005
Exploration Noise Std. Dev.	0.2 (OU)	0.1 (Gaussian)	0.1 (Gaussian)
Target Noise Std. Dev.	N/A	0.2 (Gaussian)	N/A
Target Noise Clip	N/A	-0.5 to 0.5	N/A

Figure 1: Hyperparameters for DDPG, TD3, and DDPG-v2.

3.2 Hyperparameters

Both algorithms used recommended hyperparameters from [6], but specially from [3]:

In the TD3 paper, they also used their own hyperparameters on the DDPG baseline, and that showed improvements in some environments. We decided to also try those hyperparameters, under the name DDPG-v2. The table of hyperparameters can be found at 1.

If we look at the author's TD3 implementation ([1]) we'll see different hyperparameters and network architectures, but those ones where modified so the algorithm could be used in the humanoid environment. It's important to note, however, that the original hyperparameters in theory aren't optimized, but rather chosen to be similar to DDPG [2].

We didn't try to search new hyperparameters, since we were strongly warned against doing it, and instead we followed the ones from the paper.

3.3 Training

Both algorithms were trained for $500,000$ timesteps (but finalized before, once the mean reward of the last 5 was ¿ 220), with rewards recorded for each episode. Performance metrics included episode reward, mean reward, and training stability (observed via the reward curves).

4 Results

4.1 Performance Metrics

4.1.1 DDPG Performance

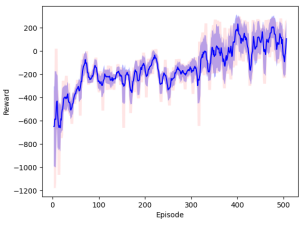

Figure 2

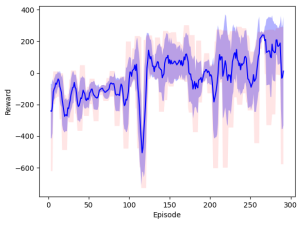

Figure 3

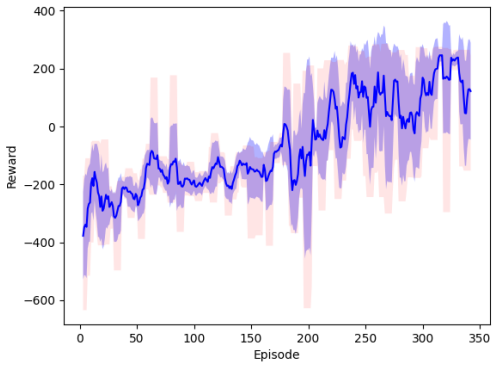

Figure 4: DDPG Learning Curves

The DDPG with stock hyperparameters has learning curves that are very unstable. As we can see in 2, the training takes, in some cases, over 500 episodes (which we'll see is a lot compared to the other algorithms). It sometimes has extreme drops in reward, causing it to learn more slowly.

4.1.2 DDPG v2 Performance

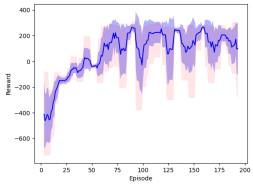

Figure 5

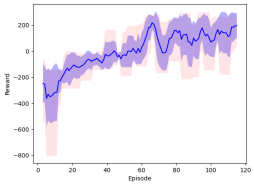

Figure 6

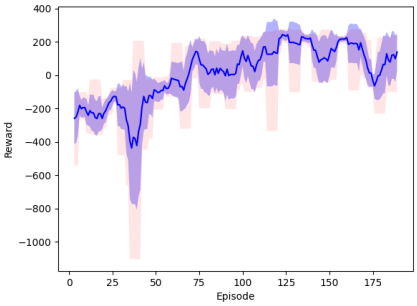

Figure 7: DDPG v2 Learning Curves

With the improved hyperparameters, DDPG is a lot better, taking almost half of the episodes to reach the same reward 6. It still is a bit unstable, although not as much, and is able to learn better the policy.

From the curves, it also seems that in some cases the algorithm was pretty close to ending its training, but it suddenly had a drop in reward. This isn't ideal, and it shows the instability of DDPG.

4.1.3 TD3 Performance

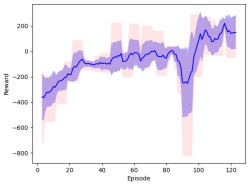

Figure 8

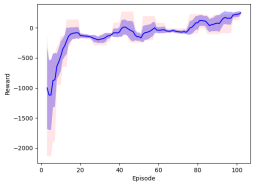

Figure 9

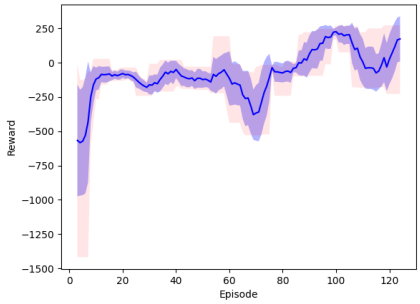

Figure 10: TD3 Learning Curves

Finally, TD3 seems to be the most stable and the fastest: in some cases, in 100 episodes it is able to finish training. It still has some drops sometimes, but in general the rewards curves are smoother. It significantly outperforms the original DDPG, and it also outperforms a bit the improved hyperparameters one.

4.2 Comparison

Seeing the reward evolutions and in general the behaviors of the algorithms, we can say that TD3 exhibited faster convergence, greater stability, and better reward consistency compared to DDPG. That was expected, since TD3 improves over DDPG. In fact, in many cases TD3 finalized it's training in half the episodes compared to DDPG (100-120 episodes vs 200-500).

We also saw the great effect that modifying the hyperparameters had: DDPG v2 was a lot better than the stock one, and it was pretty close to TD3. In fact,

in the TD3 paper they found that in some environments (like HalfCheetah) their DDPG hyperparameters where a lot better than the stock ones (section 6 of [3]).

5 Discussion and Observations

5.1 Challenges in Implementation

- **DDPG:** High sensitivity to hyperparameters and noise parameters led to frequent divergence during experiments (with slightly different hyperparameters, sometimes it took forever to converge). Also, very difficult to adjust those hyperparameters. Implementation in Pytorch wasn't difficult, some methods were very similar to DQN.

- **TD3:** While more robust, TD3's twin critic networks and delayed updates introduced some additional computational complexity. However, this complexity was compensated with a faster and more stable learning, and it ended up taking less time. Pytorch implementation was a bit more complex, but it still is a pretty simple algorithm and it didn't took too long.

5.2 Observations

- TD3 consistently outperformed DDPG in terms of stability and convergence rate.

- Delayed actor updates in TD3 prevented the actor from overfitting early inaccurate critic estimates.

- The addition of twin critics mitigated Q-value overestimation, which was evident in DDPG's erratic reward patterns.

- The use of different hyperparameters improved a lot DDPG. Those hyperparameters, however, weren't chosen specifically for the LunarLander environment, suggesting that they could also be further improved.

6 Conclusion

This work highlights the strengths of TD3 over DDPG in addressing the challenges of continuous action spaces. While DDPG laid the foundation for deterministic off-policy methods, TD3's enhancements make it the superior choice for stable and efficient learning.

We saw how TD3 was in fact smoother during learning, and was able to learn better policies. This was what we expected, since TD3 improvements tailored known problems from DQN (like the overestimation bias). Looking at the videos, it seems that the policies learned by DDPG where also a bit more chaotic, while TD3 is able to control the spaceship more and had a policy that seems more cautious.

However, several other advanced algorithms could potentially offer further improvements:

- **Soft Actor-Critic (SAC):** By incorporating entropy maximization and stochastic policies, SAC could potentially improve upon TD3's performance. Its temperature parameter automatically adjusts exploration, which could lead to more robust policies and better sample efficiency in our continuous control tasks [4].

- **Proximal Policy Optimization (PPO):** While being an on-policy algorithm, PPO's trust region approach and clipped objective function could provide more stable learning compared to TD3. Its simplicity in implementation and strong performance across various environments make it an attractive alternative worth exploring [10].

- **Distributed Distributional DDPG (D4PG):** By extending DDPG with distributional value functions and distributed training, D4PG could potentially achieve better performance. Its ability to model the full distribution of returns, rather than just the mean, could lead to more nuanced and robust policies [5].

Future work could focus on implementing these algorithms and comparing their performance against

TD3 and DDPG. Particularly interesting would be an analysis of their sample efficiency, computational requirements, and robustness to hyperparameter choices. Also, to completely validate the claims from this work, we should have done more experiments (more than 3 ideally), in more environments and with better hyperparameters.

References

[1] Scott Fujimoto. Td3 implementation. https://github.com/sfujim/TD3/blob/master/TD3.py, 2018.

[2] Scott Fujimoto. Td3 hyperparameters not optimized, but chosen to be similar to ddpg. https://github.com/sfujim/TD3/issues/21#issuecomment-579332676, 2019.

[3] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018. URL https://arxiv.org/pdf/1802.09477.

[4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018. URL https://arxiv.org/pdf/1801.01290.

[5] Matteo Hessel, Hado van Hasselt, John Aslanides, Deeparnab Chakrabarty, and David Silver. Distributional reinforcement learning with quantile regression. *arXiv preprint arXiv:1804.08617*, 2018. URL https://arxiv.org/pdf/1804.08617.

[6] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. URL https://arxiv.org/pdf/1509.02971.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. URL https://arxiv.org/pdf/1312.5602.

[8] OpenAI. Spinning up in deep reinforcement learning: Ddpg implementation. https://github.com/openai/spinningup/blob/master/spinup/algos/pytorch/ddpg/ddpg.py, 2018.

[9] OpenAI. Spinning up in deep reinforcement learning: Td3 implementation. https://github.com/openai/spinningup/blob/master/spinup/algos/pytorch/td3/td3.py, 2018.

[10] John Schulman, Filip Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/pdf/1707.06347.