Project 2 - CS7642: Reinforcement Learning and Decision Making

Syed Muhammad Husainie
shusainie3@gatech.edu
GitHub Hash: 64a5009
Georgia Institute of Technology
OMSCS – CS 7643: Deep Learning

Abstract

In this project, the reinforcement learning Deep Deterministic Policy Gradient (DDPG) algorithm is implemented for lunar landing in a simulated environment. The goal of the agent is to safely land the lander between two flags. DDPG incorporates a function approximation algorithm to explore the environment and learn a policy that successfully lands the lander over a 100-episode evaluation period. The LunarLanderContinuous-v3 environment from Gymnasium is used for this project; which has a predefined set of rules for rewards.

1 Lunar Landing Continuous Environment

1.1 State Space

The state vector in `LunarLanderContinuous-v3` is 8-dimensional: six continuous variables and two binary. The continuous variables are horizontal and vertical positions (x, y), velocities ($\dot{x}$, $\dot{y}$), and orientation (θ, $\dot{\theta}$). The binary variables, leg_L and leg_R, indicate whether the left or right leg is touching the ground (1 if touching, 0 otherwise).

1.2 Action Space

The agent controls two continuous actions: main engine and side boosters. The main engine throttle ranges from -1 to 1, with effective thrust only between 0 (50%) and 1 (100%). Side booster throttle also ranges from -1 to 1: values below -0.5 fire the left booster, above 0.5 fire the right, both scaled from 50% to 100% power. Intermediate values result in no side thrust.

1.3 Reward Function

Rewards encourage smooth, efficient landings. Positive reward is given for approaching the landing pad; moving away incurs penalties. Contact by each leg grants +10 points. A successful landing gives +100, crashing results in -100. Firing the main engine incurs a -0.3 penalty to promote fuel efficiency (fuel is unlimited). The environment is considered solved when the agent achieves an average reward of 200+ over 100 episodes.

1.4 Alogirthms for Implementation

To tackle the lunar landing problem, selecting the appropriate algorithm was a critical first step. Several algorithms were possible and would likely solve the environment if implemented effectively. Given that this environment implemented a continuous action space, precise control was required and so, different options were considered as follows:

Value-based methods: This included methods such as Q-Learning and Deep Q-Networks (DQN) which are powerful algorithms for discrete action spaces. However, they do not scale well with continuous action spaces which have high action dimensions — making them impractical for this project. For the lunar landing action space, although the action space is only 2 dimensions (main engine throttle and orientation thruster), discretizing it even into a moderate number of bins (e.g., 10 bins per action) would lead to a very high action space (up to $10^2 = 100$ discrete actions). This increased complexity not only increases training time and computational cost but also reduces the policy's precision and flexibility. Low amount of discretization bins can lead to suboptimal policies that struggle with fine-grained control, which is critical for stable lunar landing. Furthermore, adding discretization introduces an additional hyperparameter (number of bins) that requires tuning, further complicating the training process.

Policy gradient methods: Policy gradient algorithms, especially those with actor-critic architectures, naturally handle continuous action spaces by directly parameterizing the policy as a function approximator (usually a neural network) (3). . This approach removes the need for discretization and allows the agent to output continuous-valued actions, enabling precise control necessary for tasks like lunar landing. Among the modern policy gradient methods available (such as Twin Delayed Deep Deterministic Policy Gradient (TD3), Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), etc.), Deep Deterministic Policy Gradient (DDPG) was selected for several reasons:

1. **Algorithm simplicity and conceptual clarity:** DDPG is a straightforward extension of deterministic policy gradients combined with actor-critic methods. It combines value-based and policy gradient approaches by learning both a critic network (estimating the Q-value) and an actor network (learning a deterministic policy). This modular separation made it easier to implement and debug in a modular codebase, which was valuable given project time constraints.

2. **Effective use of replay buffers and off-policy learning:** DDPG is off-policy and utilizes a replay buffer to store experiences, which improves sample efficiency by enabling multiple reuses of past transitions. This was beneficial for stabilizing learning in the stochastic and noisy lunar landing environment where data collection can be expensive and noisy.

3. **Good performance on continuous control benchmarks:** DDPG has demonstrated good performance on classical continuous control benchmarks such as MuJoCo and OpenAI Gym environments, including lunar lander variants. This established track record supports its suitability for the problem at hand (2; 1).

4. **Baseline for future improvements:** Using DDPG served as a solid baseline model. While newer algorithms like TD3 or SAC offer improvements such as better exploration and stability, DDPG's simpler architecture was sufficient for an initial exploration. It also leaves room to incrementally improve with more advanced methods later.

5. **Parameterized policy and critic networks:** By parameterizing both the actor and critic as neural networks, DDPG can approximate complex policies and value functions needed for the nuanced control in lunar landing without explicit discretization.

In summary, DDPG's ability to handle continuous action spaces directly, combined with its straightforward implementation, sample efficiency through replay buffers, and proven success on similar tasks, justified its selection. This approach balanced complexity and performance, making it a suitable choice for a methodical initial implementation and in-depth exploration before considering more sophisticated algorithms.

1.5 Lunar Lander Implementation

The lunar lander problem was implemented using several modular Python files:

- **networks.py**: This file implements the DDPG actor and critic networks. Both networks are parameterized using deep neural networks to represent the policy and the Q-value function.

- **noise.py**: Applying the Gaussian Noise function. A static variance model was implemented as a baseline to measure the effect of exploration.
- **replay_buffer.py**: The replay buffer is used to store tuples of (state, action, reward, next state, done) and support random sampling for training. Once the buffer reaches its maximum capacity, older experiences are replaced.
- **ddpg_agent.py**: The DDPG agent handles the implementation of the algorithm for DDPG by storing the online and target actor/ critic policies using neural networks and the Bellman equation.
- **lander_implementation.py**: The lander implementation file initializes the environment, process hyper parameters, and training loop. It then trains the target policy and evaluates the learned policy for 100 episodes.

1.6 Hyperparameters Setup for DDPG

DDPG Agent: Hidden dimensions = (400, 400) $\gamma = 0.99$, $\tau = 0.005$, value and policy learning rates = 0.001.

Gaussian Noise: Standard deviation = 0.1.

Replay Buffer: Max Size = 10000, Batch Size = 64.

Warm-up Steps: 800.

Number of Episodes: 2000.

These values served as baselines and some of them were later varied to study their impact on learned policies.

2 Implementation and Discussion of Results

2.1 Tuning Parameters

In order to successfully implement the DDPG environment, the following parameters were tuned:

Number of Episodes The number of training episodes is a key hyperparameter that balances exploration and exploitation. In this DDPG experiments, performance improved steadily up to around 700–1200 episodes but declined afterward, suggesting policy instability or overfitting. Too few episodes can lead to suboptimal policies, while too many may reinforce unstable behaviors—a known issue in off-policy methods like DDPG. Monitoring the moving average of rewards helps identify the optimal training duration for a stable and effective policy.

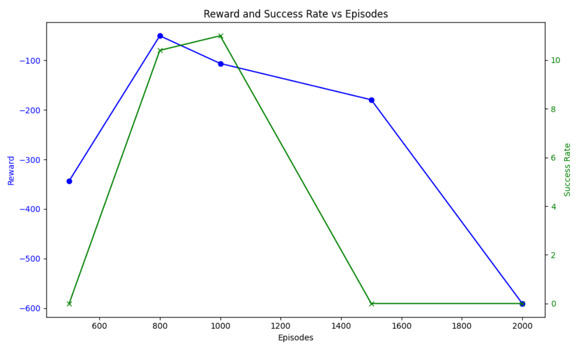


Figure 1: Plot of the average reward and success rate versus number of episodes during the 100 episode evaluation period in LunarLandingContinuous

Figure 1 illustrates the effects of under-training and over-training across different episode counts. At 500 episodes, the policy evaluation resulted in consistently low performance, characterized by highly negative average rewards and zero successful landings, indicating that the agent had not yet sufficiently learned to control the lander. Similarly, when training extended beyond 1500 episodes, the policy performance again degraded, showing a high negative reward and no successful landings, suggesting overfitting or instability in the learned policy. Notably, the optimal performance was observed between 800 and 1000 episodes, where the average reward over 100 evaluation episodes increased significantly, accompanied by approximately 10% successful landings. This intermediate range reflects a balance where the agent had learned a sufficiently effective policy without being over-trained. It should be noted, however, that the average reward remained negative, indicating that further tuning of the model was necessary. Despite this, the experiments justified the implementation of a training constraint to prevent over-training. Specifically, for the next hyperparameters tuned as part of this project, the training was set to stop once the average reward over the most recent 40 episodes exceeded 200; signaling that an effective policy had been learned. Choosing a 40-episode rolling average, rather than a longer window such as 50 episodes, allowed for faster training turnaround while still ensuring a robust policy.

2. Hidden Layers: The number of hidden layers can be an important parameter for effective implementation of the neural networks in the actor-critic architecture and for extracting useful properties from the model. Having a low number of hidden layers may not extract high information from the model, especially in environments with complex dynamics where deeper representations can help capture subtle control features. On the other hand, using a very high number of layers can be computationally expensive and may lead to vanishing gradients or overfitting, especially when training data is sparse or noisy. Moreover, increasing the depth of the network can initially lead to significant improvements in learning, as more abstract features are modeled. However, there is a threshold beyond which additional layers may introduce redundancy or unnecessary complexity, making it harder for the agent to generalize.

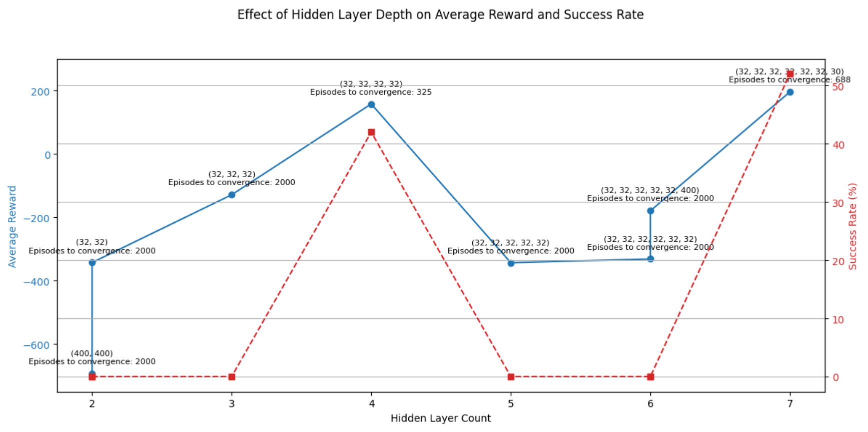


Figure 2: Average reward and success rate for 100 episode evaluations of the trained policy using different hidden sizes and number of layers in LunarLandingContinuous

Figure 2 shows that models with only 2 or 3 hidden layers consistently failed to learn effective policies, exhibiting low average rewards and 0% success rates even after extensive training. This poor performance is likely due to insufficient representational capacity, as these smaller networks, despite large layer sizes (such as (400, 400) versus (32, 32)), were unable to capture the complexity of the task. In contrast, the 4-layer model with 32 units per layer struck a balance between capacity and trainability, achieving a 42% success rate and converging rapidly within 325 episodes. However, increasing the depth to 5 and 6 layers led to a sharp decline in performance, with both models failing to converge to useful policies and maintaining 0% success rates. This drop is likely due to optimization difficulties common in deeper networks, such as vanishing or exploding gradients, coupled with potential overparameterization and the lack of architectural enhancements like normalization or skip connections, which were not added to preserve model simplicity. Consequently, these intermediate-depth networks appear to have been too deep to train easily but not sufficiently deep or well-structured to overcome the complexity challenges, resulting in unstable training and failure to learn. Interestingly, going from 6 to 7 layers increased the success rate to 52%. This highlights the critical need for a carefully balanced architecture that aligns depth, width, and training methodology with the task at hand. Ultimately, lower-dimensional networks with either 4 or 7 layers performed effectively: the policy converged in just 325 episodes with 4 layers, whereas the 7-layer model converged more slowly but achieved a higher success rate of 52%. It should be noted, however, that due to random weight initialization and the stochastic nature of the environment, even the 4- and 7-layer models can occasionally suffer from gradient explosion, affecting stability across runs.

3. Warm-up Steps: Before training starts for DDPG, the buffer is populated with a set of random experiences which count as warm-up steps. This phase is critical because it influences the initial learning trajectory of both the actor and critic networks. If learning starts too early, the buffer contains poor quality and sparse samples, leading to unstable or suboptimal updates. On the other hand, if learning starts too late, the buffer is filled with noisy and uninformative data, limiting learning efficiency. This shows there is a sweet spot range of warm-up steps that provides just enough diversity for stable learning without overwhelming the networks with randomness.

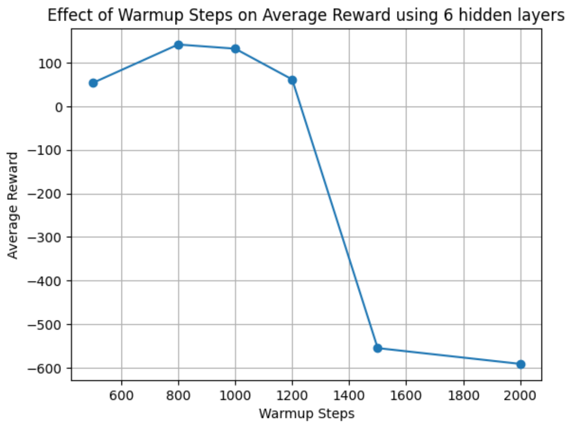


Figure 3: Plot of Warmup Steps on Average Reward using 6 hidden layers in LunarLandingContinuous

Results from figure 3 confirm this as the performance peaked between 800-1000 warm-up steps. As the warmup steps fell significantly below 800 or above 1000, there was noticeable drops in the average reward.

4. Noise Standard Deviation: The standard deviation of the Gaussian noise added to the actions plays a major role in balancing exploration and exploitation in DDPG. If the noise is too low, the agent does not explore enough and can get stuck in suboptimal behaviors. On the other hand, high noise leads to unstable policies and excessive randomness in action selection.

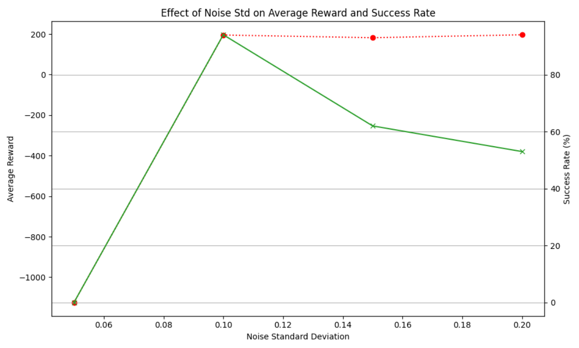


Figure 4: Average reward and success % over the noise standard deviation

From the results, a noise standard deviation of 0.1 gave the best overall performance, achieving a high average reward of 195.53 and a success rate of 94%. Too little noise (0.05) led to extremely poor rewards and no success, while larger noise values like 0.15 and 0.2 maintained moderate rewards but saw drops in success rate. This clearly shows the importance of carefully tuning the noise parameter, since it directly impacts how the agent learns effective policies over time.

5. Replay Buffer Size: The size of the replay buffer determines how much past experience the agent can access when sampling for training. A small buffer may limit diversity in training samples and lead to overfitting on recent transitions, while an excessively large buffer can introduce outdated or irrelevant transitions, especially in non-stationary settings.

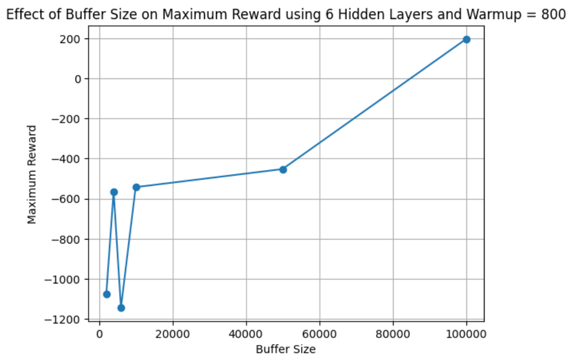


Figure 5: Maximum reward over varying buffer sizes in LunarLandingContinuous

The results show that increasing the buffer size from 2000 to 100000 had a high positive effect. While moderate improvements were seen at 4000, the agent still performed poorly with maximum reward only reaching −564.98. The most dramatic leap was observed at a buffer size of 10000, where the maximum reward shot up to 195.53. This suggests that a larger buffer provided richer and more varied experience, which allowed the critic to better estimate Q-values and the actor to stabilize its policy. However, the performance at 6000 dropped again to −1142.46, implying that simply increasing buffer size is not always better and it must align with the agent's capacity, the warm-up steps, and learning dynamics. Moreover, since the other hyper-parameters were left constant during this study, these results also speak to the random nature of the lunar landing environment which can produce exploding gradients even with good parameters, if the initial weights are not optimal.

6. Soft Update Coefficient (τ): The soft update coefficient τ determines how gradually the target networks track the main networks. This also has a fine balance since slow updating can cause the policy to not learn effectively, while fast updates can cause the policy to learn random or noisy data.

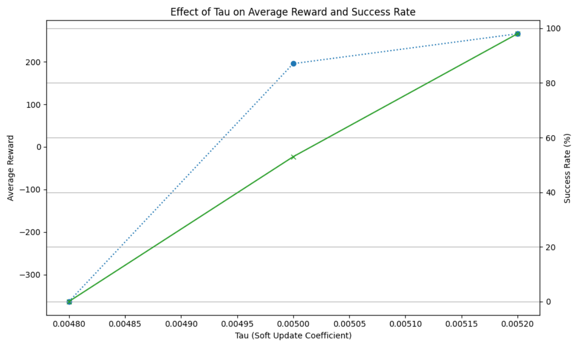
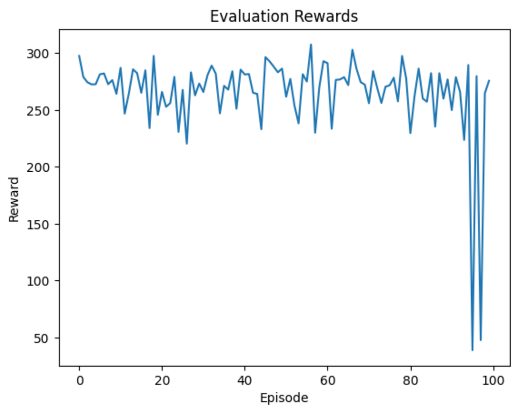


Figure 6: Effect of soft update coefficient (τ) on average reward and success rate in LunarLander-Continuous.

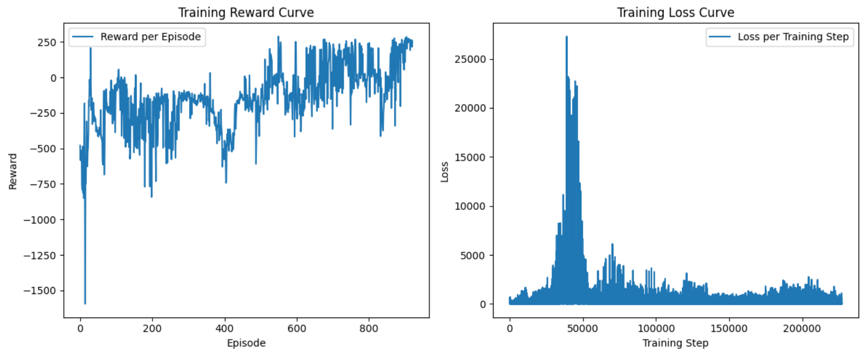
From the results, it can be seen that at $\tau = 0.0048$, the agent struggles with an average reward of -363.34 and no successful landings. Increasing τ to 0.005 leads to a significant jump in performance with an average reward of 195.53 and a 53% success rate. Further increasing it to 0.0052 yields an even better reward of 265.7 and a 98% success rate. These results suggest that a slightly faster tracking of the target networks can accelerate convergence and stability in the LunarLanderContinuous environment. However, this only works up to a point and if the updates are too large, it can destabilize learning. Nevertheless, for this problem the slightly faster updates were highly beneficial and also hints that a very small τ may overly delay policy correction, especially in environments where early mistakes compound quickly.

2.2 Final Remarks on DDPG Model Performance

Using a warm-up step size of 800, a buffer size of 100,000, a training cutoff criterion based on a rolling average reward over 40 episodes exceeding 200, a batch size of 64, a discount factor (γ) of 0.99, and a target network update parameter (τ) of 0.0055, the model with seven hidden layers (sizes: 32, 32, 32, 32, 32, 32, 30) and policy/value learning rates of 0.001 achieved an optimal evaluation average reward of 265.7 with a success rate of 98%. However, as shown in evaluation figure 7a below, the stochastic nature of the `LunarLanderContinuous` environment leads to noticeable fluctuations in episode rewards despite the policy effectively learning. Figure 7b indicates that the curve starts with highly negative rewards (around -1500), but as the episodes increase, there's a clear upwards trend and by episode 600 the reward ranges in positive values, reaching 250+ reward in some cases. Although some fluctuations do exist, this is expected due to exploration and environment stochasticity. The training loss curve shows that it starts small but increases sharply at 40000 steps followed by a rapid decrease and stabilization. This stabilization is a good sign (going in hand with the high training reward), and is indicative of the value network learning good estimates. It was also encouraging to observe that a simple noise model performed well, and noise decay, when tested, did not improve performance (results not shown in this report). The final model converged after 797 episodes, requiring approximately one hour to complete training.



(a) Policy evaluation over 100 episodes



(b) Training reward and loss plots over episode duration

Figure 7: Policy metrics for the final policy

2.3 Conclusions

The DDPG model demonstrated strong performance on the LunarLanderContinuous environment, achieving a high average reward and success rate with a relatively simple noise model. Despite the inherent randomness of the environment causing occasional fluctuations in episode rewards, the learned policy was effective and stable. The convergence within 797 episodes and the total training time of approximately one hour show the model's efficiency given the chosen hyperparameters and network architecture. It is important to note however that running the same DDPG model three times resulted in success rates of 98%, 68%, and 63%, highlighting the high variance often observed in reinforcement learning, especially with off-policy algorithms like DDPG. This variability stems from factors such as sensitivity to initialization, stochastic exploration, and instability in continuous action spaces. These results emphasize the need for implementing an even more robust model.

For future work, it would be worthwhile to implement the TD3 model to assess whether it can achieve faster convergence while preserving or improving the average reward and stability. Additionally, experimenting with alternative noise strategies or adaptive noise scaling could potentially enhance exploration without sacrificing performance. Further hyperparameter tuning, such as adjusting the network size or learning rates, may also yield improvements, although 98% success rate seems hard to beat. Finally, testing the models on other continuous control tasks would help generalize the findings and validate the robustness of the approach.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. https://arxiv.org/abs/1606.01540, 2016.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tim Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, MA, 1998.