# BipedalWalker with different algorithms

Yernar Akhmetbek

November 2024

## 1 Abstract

The implementation involved integrating three reinforcement learning algorithms: Q-learning, Deep Q-Network, and Twin Delayed DDPG. The goal was to begin with a simple algorithm and gradually increase complexity to demonstrate the differences in performance. Each method was tested on the "BipedalWalker-v3" environment from OpenAI Gym, where the agent's objective was to move forward without falling. The Twin Delayed DDPG algorithm emerged as the most successful in accomplishing the task compared to the others.

## 2 Introduction

Reinforcement learning is a fascinating area of machine learning where agents learn to perform a task by maximizing rewards. The goal of this project was to implement several learning algorithms with the BipedalWalker-v3 agent and compare their performance. The agent's task was to devise a strategy that maximized the reward, given four possible actions at each time step.

The motivation behind this work was to explore reinforcement learning techniques and apply them to real-world problems. For instance, teaching a pair of mechanical legs to walk could help individuals with disabilities, while reinforcement learning could also be used in autonomous vehicles to enable cars to park without driver assistance. These examples highlight the importance of analyzing reinforcement learning algorithms and understanding both their potential and limitations.

The algorithms implemented include Q-learning, Deep Q-Network (DQN), and Twin Delayed Deep Deterministic Policy Gradient (TD3). The results showed that more complex algorithms performed better than simpler ones. It was found that Q-learning is better suited for smaller learning problems, as it requires no model and operates using the epsilon-greedy policy. While DQN performed better than Q-learning, it still struggled with this task due to the continuous action domain. The implementation of TD3 aimed to address the limitations of DQN and showed considerably better performance. However, TD3 also introduced its own challenges.

# 3 Details

All the algorithms are grounded in the core principle of the Q-function, which is employed as an optimal action-value function. This function follows the Bellman equation, where the action selected in each update aims to maximize the estimated Q-value for the next time step. The equation is expressed as follows:

Q ( s , a ) = Q ( s , a ) + ( r + max Q ( s , a ) Q ( s , a ) ) (1) Q(s,a)=Q(s,a)+(r+maxQ(s ,a )Q(s,a))(1) In this equation, s s denotes the current state, a a the current action, s s the next state, a a the next action, r r the reward received, the learning rate, and the discount factor for future rewards.

While each algorithm applies a variant of the Q-function and introduces various modifications to improve performance, they all face unique challenges that prevent them from completely mastering the task.

## 3.1 Q-Learning

Q-Learning is a straightforward reinforcement learning algorithm. Similar to other algorithms that use the Q-function to implement the Bellman equation, its goal is to select actions that maximize the Q-value. The actions are chosen using an -greedy policy, meaning that there is a probability of for selecting a random action instead of the one that would maximize the Q-value. This randomness prevents the algorithm from getting stuck in local minima or maxima. In our implementation, the value of changes dynamically over time, decaying as episodes progress. Early on, is close to 1, meaning actions are chosen randomly most of the time. After 500 episodes, the probability of selecting a random action reduces to about 0.5. The algorithm does not rely on a model; instead, it stores the possible actions and their corresponding Q-values in a Q-table.

## 3.2 Deep Q-Network

Deep Q-Network (DQN) retains the core features of Q-learning, such as the -greedy policy, but enhances the algorithm with a deep neural network, experience replay, and stochastic gradient descent for weight updates. This approach has been successfully applied in tasks like playing Atari 2600 games (Mnih et al., 2015). The authors introduce experience replay, where the last N experiences are stored, with each experience represented as a tuple ( s , a , r , s , d ) (s,a,r,s ,d), where d d indicates whether the experience is terminal. To generalize the Q-value estimation, they modify the Bellman equation, enabling the Q-network Q ( s , a ; ) Q(s,a;) to approximate the Q-value. During training, the weights are updated using stochastic gradient descent based on a loss function that calculates the mean squared error between the target y y and the Q-value approximation Q ( s , a ; ) Q(s,a;). Targets are generated by a second Q-network Q Q that is periodically cloned from Q Q to ensure stability, preventing divergence caused by frequent network updates. Stochastic gradient

descent is preferred over gradient descent due to its efficiency, as it processes a mini-batch of experiences, although it sacrifices some accuracy due to noise.

### 3.3 Twin Delayed DDPG

TD3 builds upon the concepts introduced by DQN, including deep neural networks, experience replay, and stochastic gradient descent. However, TD3 addresses issues present in DQN by incorporating an actor-critic architecture. In this setup, the "actor" is responsible for selecting actions based on the policy, while the "critic" evaluates the state-action pairs to estimate their value. A known issue with value-based methods like Q-learning is the tendency to overestimate Q-values, leading to suboptimal policies (Fujimoto et al.). TD3 aims to tackle this problem by improving upon Double DQN using the Deep Deterministic Policy Gradient (DDPG) algorithm.

TD3 consists of six distinct networks: an actor, two critics, a target actor, and two target critics. It introduces three key modifications to Double DQN: Target Policy Smoothing, Clipped Double-Q Learning, and Delayed Policy Updates. In Clipped Double-Q Learning, TD3 uses two Q-functions, Q 1 Q 1
and Q 2 Q 2
, and takes the minimum value between the two to reduce the overestimation bias. This helps in preventing the algorithm from retaining and propagating suboptimal states.

Delayed Policy Updates involve updating the policy network less frequently than the value network. This is important because the policy network can behave erratically when exposed to high-error states, and infrequent updates from the target network help stabilize the Q-values (Fujimoto et al.).

Lastly, Target Policy Smoothing introduces small amounts of random noise to the target policy, sampled from a normal distribution N ( 0 , 0.2 ) N(0,0.2), and clips the noise to the range [ 0.5 , 0.5 ] [0.5,0.5] to keep the modified action close to the original. This helps in improving the stability of the policy update.

## 4 Reward System

In reinforcement learning (RL), the reward system is essential for guiding an agent toward its objective. In our implementation of Q-learning, Deep Q-Networks (DQN), and Twin Delayed DDPG (TD3), the reward system in the BipedalWalker-v3 environment plays a key role in shaping the agent's learning process.

For the BipedalWalker-v3 environment, the agent is tasked with maintaining balance and progressing forward without falling. The reward system is designed to provide feedback based on the agent's movements. At each time step, the agent receives a reward depending on its actions, which can either encourage continued progress or penalize failure.

In the case of Q-learning, the agent is rewarded or penalized based on its actions, and the Q-values are updated according to the Bellman equation. The

Q-learning algorithm chooses actions with an -greedy policy, balancing exploration and exploitation. Early in the episodes, the value of  is high, meaning the agent explores the environment more randomly. As  decays, the agent increasingly exploits its learned knowledge, selecting actions that maximize the Q-value. The reward system in Q-learning directly affects how the agent refines its Q-values, with the ultimate goal of learning to select actions that maximize cumulative rewards, like avoiding falls and achieving forward progress.

For DQN, which extends Q-learning with deep neural networks, experience replay, and stochastic gradient descent, the reward system similarly influences the network's learning process. DQN experiences a more complex reward structure due to the continuous action space of BipedalWalker-v3. The reward is calculated after each action, and the Q-values are updated based on the observed reward. The experience replay mechanism allows the agent to revisit previous experiences, enhancing its ability to generalize from the rewards it has received over time. The reward function in DQN encourages the agent to move forward and maintain balance, but the challenges of continuous action space and overestimation of Q-values, as seen in our results, prevent the algorithm from reaching optimal performance.

In the TD3 algorithm, the reward system plays a crucial role in the training process, similar to DQN. However, TD3 addresses the overestimation issue present in DQN by using two critic networks and incorporating techniques such as target policy smoothing, clipped Double-Q learning, and delayed policy updates. These modifications allow TD3 to handle the reward system more effectively, leading to better performance compared to Q-learning and DQN. The reward system in TD3 guides the agent to explore more stable and efficient actions, especially in continuous action spaces. The algorithm's actor-critic architecture enables the agent to balance exploration and exploitation while minimizing the effect of catastrophic forgetting and maximizing cumulative rewards over time.

Thus, in all three algorithms, the reward system in BipedalWalker-v3 serves as the backbone for agent learning, influencing how actions are selected and how knowledge is updated. The algorithms adapt to the continuous and dynamic nature of the reward system, with the ultimate goal of improving performance and achieving efficient movement without falling.

# 5 Comparison and Evaluation

To assess the performance of each algorithm, the "score" for the agent in each episode was recorded. The score was defined as the total cumulative reward collected during an episode. An episode would end under one of three conditions: the agent falls and touches the ground, the agent reaches a terminal state, or the agent completes the environment. Rewards were calculated based on the agent's movements. For example, if the agent fell, it would receive a score of -100. Therefore, the agent's score improves the farther it progresses without falling. We compared the scores of the Q-learning, DQN, and TD3 algorithms

to evaluate their effectiveness. This comparison was done by plotting the scores for a particular run of each algorithm and analyzing the results. Based on these observations, we developed hypotheses to explain the performance of each algorithm.

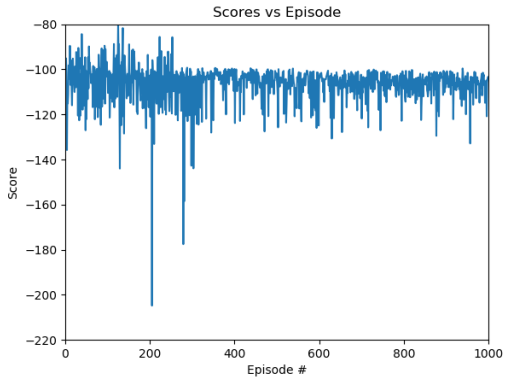


Figure 1: Q-Learning

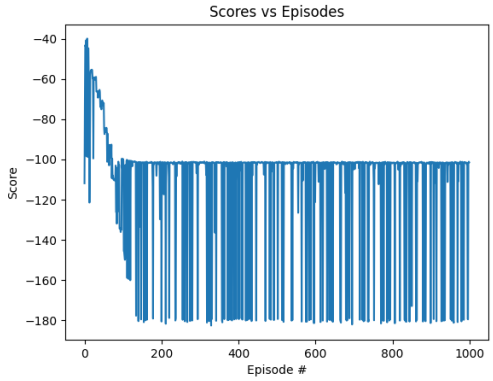


Figure 2: DQ

As illustrated in Figure 1, Q-learning struggles to effectively train the BipedalWalker-v3 agent. In the beginning, the agent made some progress, reaching a peak score of -79.6. However, as training continued, the agent consistently performed poorly. This is due to the decaying epsilon value. At the start of training, epsilon is relatively high, causing the agent to choose random actions more frequently than those that would maximize the Q-score. Over time, epsilon decreases, leading the agent to prioritize actions that maximize the Q-score. Unfortunately, this strategy does not work well in this case, as the action space

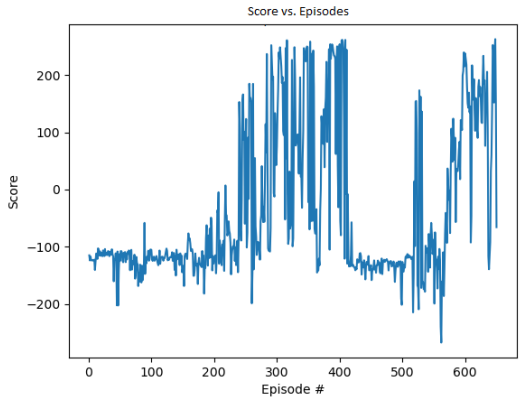


Figure 3: TD3

in the BipedalWalker-v3 environment is continuous, while Q-learning is better suited for discrete, low-dimensional action spaces. As a result, Q-learning is more effective for simpler, smaller-scale problems and struggles with more complex, continuous action spaces like the one in this environment.

As shown in Figure 2, the DQN agent initially performed well, achieving a maximum score of -40. However, as training progressed, the agent's performance plateaued, and it could only achieve scores ranging from -100 to -180 after a short time. Additionally, the agent took 1 hour, 26 minutes, and 17 seconds to complete the task. We ran multiple trials to observe the model's consistency, and the results showed that the agent's behavior was consistently similar across runs. The main reason for DQN's poor performance seems to be the continuous action space of BipedalWalker. As discussed by Lillicrap et al. (2015), while DQN is effective in high-dimensional observation spaces, it performs best with discrete, low-dimensional action spaces, which explains its success in environments like Atari 2600. BipedalWalker, on the other hand, has four continuous action dimensions (Hip1, Knee1, Hip2, Knee2), with each dimension ranging from -1 to 1. This results in the agent needing to compute gradients at each step to maximize the Q-function in a continuous space, which is inefficient and impractical.

The agent initially performed better because the -greedy policy led it to take random actions, which occasionally resulted in better performance than actions selected by the Q-value maximization. One possible improvement to DQN could involve discretizing the action space, but this would be problematic for tasks requiring precise control, as reducing the action domain could hinder the agent's ability to perform crucial actions. To address this, we implemented TD3, an algorithm that uses the actor-critic architecture and has shown promise in environments with continuous or high-dimensional action spaces.

Initially, the agent learned quickly to stand and crawl forward by dragging its legs, reaching this stage at around 200 episodes, as shown in Figure 3. By

episode 300 ( 2 hours), the agent began to "skip," alternating between its right and left legs, with rewards peaking around 250. This trend continued to improve until episode 400, when, after several tests, I observed a drastic decline in performance. The agent lost its ability to walk entirely. Upon further investigation, I concluded that this issue was due to catastrophic forgetting—where the agent prioritizes newly learned information over previously learned knowledge (Imre, B. 2015). Essentially, as the agent learns new things, the replay buffer overwrites old experiences, causing the agent to forget what it had previously learned.

By episode 600 ( 6 hours), the agent started walking again, with a performance trend similar to episodes 200-300. Catastrophic forgetting, or interference, is a well-known issue in reinforcement learning, and it is often mitigated by using more memory and a smaller learning rate. For instance, training another BipedalWalker model from OpenAI's Gym can require up to 3GB of memory (Imre, B. 2015). To improve my TD3 implementation, one possible solution is to allocate more memory and fine-tune the hyperparameters, such as reducing the noise applied to actions. However, this creates a trade-off, especially when considering real-world applications: reinforcement learning algorithms, like DQN and TD3, are currently limited by the memory available to store and recall past experiences.

# 6 Conclusion

In this project, I implemented Q-learning, DQN, and TD3 algorithms for reinforcement learning. Q-learning provided the foundation with the Bellman equation, while DQN improved upon it with deep neural networks, experience replay, and stochastic gradient descent. However, DQN struggled due to the continuous action space in the BipedalWalker environment.

To address this, I used the TD3 algorithm, which fixes overestimation through target policy smoothing, clipped Double-Q learning, and delayed policy updates. TD3 outperformed the previous algorithms, but faced memory issues with experience replay, leading to catastrophic interference.

Despite this, TD3 shows great potential and could perform well with further development in memory handling, making it a promising area for future work.

# References


[1] Fujimoto, S.; Hoof, H.V.; Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *arXiv*, abs/1802.09477.

[2] Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; et al. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971.*

[3] Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518, 529–533. https://doi.org/10.1038/nature14236.

[4] Imre, B. (2021). An investigation of generative replay in deep reinforcement learning. (Bachelor's thesis, University of Twente).

[5] Sutton, Richard S. and Barto, Andrew G. (Year). *Reinforcement Learning: An Introduction*, MIT Press.