SAIDL Spring Assignment Submission Report

Tanmay Joshi

March 2025

1 *Reinforcement Learning Task 1*

Statement - Implement TD3 (Twin Delayed DDPG) algorithm on Mujoco's Hopper-v4 environment.
Techniques to be included - Polyak Averaging, Dual Critic Networks, Prioritized Experience Replay, and n-step returns.
Techniques to be excluded - Target Policy Smoothing Component of TD3.
Relevant Paper-https://arxiv.org/abs/1802.09477

Background- The paper *"Addressing Function Approximation Error in Actor-Critic Methods"* explores how function approximation errors contribute to overestimated value estimates and suboptimal policies. It demonstrates that this issue persists in actor-critic frameworks and introduces novel mechanisms to mitigate its impact on both the actor and the critic. Noise is inevitable in a function approximation setting. Overestimation bias is a property of Q-learning in which maximization of a noisy value estimate induces a consistent overestimation. Due to overestimation bias, this accumulated error can cause arbitrarily bad states to be estimated as high values, resulting in suboptimal policy updates and divergent behavior. Double DQN estimates the value of the current policy with a separate target value function, which makes it inefficient in an actor-critic setting.

To address the above concern, the paper proposes a clipped Double Q-learning variant which leverages that a value estimate suffering from overestimation bias can be used as an approximate upper bound to the true value estimate. This favors underestimations, which do not tend to be propagated during learning, as actions with low-value estimates are avoided by the policy. This paper explores techniques to reduce variance in deep Q-learning to mitigate overestimation bias caused by noise. Key components include:

1. **Target Networks**: Shown to be crucial in reducing error accumulation and thereby lowering variance.
2. **Delayed Policy Updates**: Proposes postponing policy updates until the value estimates stabilize to decouple value and policy learning.

3. **SARSA-style Regularization**: Introduces a new strategy that bootstraps from similar action estimates, further helping to reduce variance.

So, the paper proposes the TD3 algorithm or Twin Delayed Deep Deterministic Algorithm, an actor-critic algorithm which considers the interplay between function approximation error in both policy and value updates.

Overestimation Bias-In Q-learning with discrete actions, the value estimate is updated with a greedy target y = r + maxa0 Q(s 0 , a0), however, if the target is susceptible to error , then the maximum over the value along with its error will generally be greater than the true E[maxa0 (Q(s 0 , a0)+)] maxa0 Q(s 0 , a0) .As a result, even initially zero-mean error can cause value updates to result in a consistent overestimation bias, which is then propagated through the Bellman equation. This is problematic as errors induced by function approximation are unavoidable.

In actor-critic methods, the policy is updated with respect to the value estimates of an approximate critic. , let approx define the parameters from the actor update induced by the maximization of the approximate critic Q(s, a) and true the parameters from the hypothetical actor update with respect to the true underlying value function Q (s, a) (which is not known during learning):

$$\phi_{approx} = \phi + \frac{\alpha}{Z_1} E_{s\sim p_\pi}\left[\nabla_\phi \pi_\phi(s) \nabla_a Q_\theta(s,a)|_{a=\pi_\phi(s)}\right]$$

$$\phi_{true} = \phi + \frac{\alpha}{Z_2} E_{s\sim p_\pi}\left[\nabla_\phi \pi_\phi(s) \nabla_a Q^\pi(s,a)|_{a=\pi_\phi(s)}\right]$$

where we assume that Z1 and Z2 are chosen to normalize the gradient, that is, such that

$$Z^{-1}\left\|E[\cdot]\right\| = 1$$

.

As the gradient direction is a local maximizer, there exists ϵ_1 sufficiently small such that if $\alpha \leq \epsilon_1$ then the *approximate* value of π_{approx} will be bounded below by the approximate value of π_{true}:

$$E\left[Q_\theta(s, \pi_{approx}(s))\right] \geq E\left[Q_\theta(s, \pi_{true}(s))\right]. \tag{1}$$

Conversely, there exists ϵ_2 sufficiently small such that if $\alpha \leq \epsilon_2$ then the *true* value of π_{approx} will be bounded above by the true value of π_{true}:

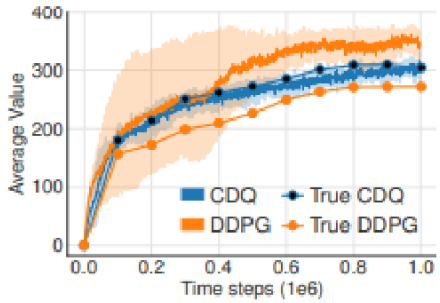
$$E\left[Q^\pi(s, \pi_{true}(s))\right] \geq E\left[Q^\pi(s, \pi_{approx}(s))\right]. \tag{2}$$

If in expectation the value estimate is at least as large as the true value with respect to ϕ_{true}, $E\left[Q_\theta(s, \pi_{true}(s))\right] \geq E\left[Q^\pi(s, \pi_{true}(s))\right]$, then Equations (1) and (2) imply that if $\alpha < \min(\epsilon_1, \epsilon_2)$, then the value estimate will be overestimated:

$$E\left[Q_\theta(s, \pi_{approx}(s))\right] \geq E\left[Q^\pi(s, \pi_{approx}(s))\right]. \tag{3}$$

Although this overestimation may be minimal with each update, the presence of error raises two concerns. Firstly, the overestimation may develop into a more significant bias over many updates if left unchecked. Secondly, an inaccurate value estimate may lead to poor policy updates. This is particularly problematic because a feedback loop is created, in which suboptimal actions might be highly rated by the suboptimal critic, reinforcing the suboptimal action in the next policy update .

Measuring overestimation bias in the value estimates of actor critic variants of Double DQN (DDQN-AC) and Double Qlearning (DQ-AC) on MuJoCo environments over 1 million time steps -



(a) Hopper-v1

A very clear overestimation bias occurs from the learning procedure, which contrasts with the novel method that we describe in the following section, Clipped Double Q-learning, which greatly reduces overestimation by the critic.

The authors explore overestimation bias in actor-critic algorithms such as DDPG and Double DQN. They find that:

- Using target networks in slow-changing policies is ineffective because they are too similar to the current networks.
- Instead, the original Double Q-learning setup is adapted using two actors $(\pi_{\phi_1}, \pi_{\phi_2})$ and two critics $(Q_{\theta_1}, Q_{\theta_2})$, where each actor is optimized with respect to its paired critic.
- Despite this setup, overestimation bias persists, as the critics are not fully independent due to shared replay data and the interdependent learning targets.

To mitigate this, the authors introduce **Clipped Double Q-learning**, which uses the minimum of the two critic estimates to compute the value target:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_{\phi_1}(s')) \tag{4}$$

This approach reduces overestimation bias, although it may introduce some underestimation. However, this is considered acceptable, as underestimated action values are not reinforced or propagated during policy updates.

Addressing Variance: Besides the impact on overestimation Addressing Function Approximation Error in Actor-Critic Methods bias, high variance estimates provide a noisy gradient for the policy update. This is known to reduce learning speed as well as hurt performance in practice.

Due to TD update, there is a built up of error in each step. This can be given as -

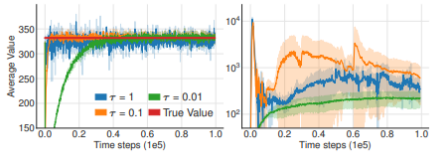
$$Q_\theta(s,a) = r + \gamma E\left[Q_\theta(s',a')\right] - \delta(s,a) \tag{5}$$

which can be further shown as -

$$Q_\theta(s_t,a_t) = r_t + \gamma E\left[Q_\theta(s_{t+1},a_{t+1})\right] - \delta_t = r_t + \gamma E\left[r_{t+1} + \gamma E\left[Q_\theta(s_{t+2},a_{t+2})\right] - \delta_{t+1}\right] - \delta_t$$

$$= \mathrm{E}_{s_i\sim p_\pi, a_i\sim\pi}\left[\sum_{i=t}^{T}\gamma^{i-t}(r_i - \delta_i)\right] \tag{6}$$

When the discount factor γ is large, the variance in value estimates can increase rapidly with each update if the associated errors are not effectively controlled. Moreover, each gradient update is performed on a small mini-batch, which only reduces error locally and offers no guarantees regarding the accuracy of value estimates outside the sampled mini-batch.



(a) Fixed Policy (b) Learned Policy

The above results represent the average estimated value of a randomly selected state on the `Hopper-v1` environment, evaluated under different target network settings. Specifically, they compare the cases without target networks ($\tau = 1$) and with slow-updating target networks ($\tau = 0.1, 0.01$), using both a fixed policy and a learned policy.

While updating the value estimate without target networks ($= 1$) increases the volatility, all update rates result in similar convergent behaviors when considering a fixed policy. However, when the policy is trained with the current value estimate, the use of fast-updating target networks results in highly divergent behavior. These results suggest that the divergence that occurs without target networks is the result of policy updates with a high variance value estimate. If target networks can be used to reduce the error over multiple updates, and policy updates on high-error states cause divergent behavior, then the policy network should be updated at a lower frequency than the value network, to first minimize error before introducing a policy update. We propose delaying policy updates until the value error is as small as possible. The modification is to only update the policy and target networks after a fixed number of updates

d to the critic. To ensure the TD-error remains small, we update the target networks slowly $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$.

Algorithm For TD3–

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
$t = 1$ to T
Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$
Observe reward r and new state s'
Store transition tuple (s, a, r, s') in $\mathcal{B}$
Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
$\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim clip(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
$y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
Update critics $\theta_i \leftarrow \arg\min_{\theta_i} \frac{1}{N} \sum (y - Q_{\theta_i}(s, a))^2$
$td = 0$
Update ϕ by the deterministic policy gradient:

$$\nabla_\phi J(\phi) = \frac{1}{N} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$$

Update target networks:

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i' \qquad \phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

Why TD3? TD3 increases the stability and performance with consideration of function approximation error. TD3 maintains a pair of critics along with a single actor.

My Implementation of TD3- As mentioned in the algorithm given above we have a dual critic network with a prioritized experience replay.

1)Actor-

The network receives an input state as a tensor with a size defined by `input_size`, representing all the features that describe the current situation in the environment. This state is first processed by a fully connected linear layer that transforms it into 400 features; the resulting output is then passed through a ReLU activation function to introduce non-linearity and eliminate negative values. The output from this layer is intended to be further processed by a second linear layer that ideally reduces the dimensionality to 300 features, again followed by a ReLU activation to enhance non-linear representation (note that the provided code indicates a potential inconsistency in this layer's dimensions that may need correction). Next, these 300 features are passed through

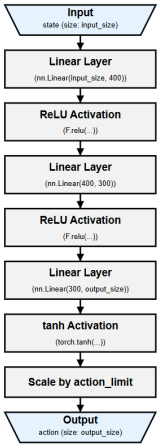
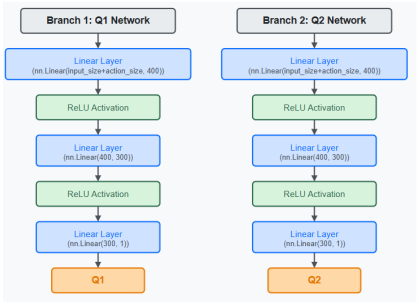


Figure 1: Actor Network Architecture

an output linear layer that maps them to a vector of size `output_size`, corresponding to the number of action dimensions the network must produce. This vector is then squashed using a tanh activation function to limit the values within the range [-1, 1]. Finally, the resulting output is scaled by a factor called `action_limit`, ensuring that the action values fall within the acceptable range required by the environment. The final output is thus an action vector of size `output_size`, ready for use by the agent in its decision-making process.

Why take 400 and 300 neurons in the subsequent layers?
The choice of 400 and 300 neurons in the layers is largely based on empirical evidence and is inspired by similar architectures in reinforcement learning methods like DDPG and TD3. In my experiments, using a configuration of 512 and 256 neurons resulted in higher computational costs and longer execution times, while using 256 and 128 neurons proved insufficient to handle the complexity of the environment and risked overfitting. The first layer with 400 neurons is robust enough to capture a diverse range of features from the input, while reducing the dimensionality to 300 neurons in the subsequent layer strikes a good balance between preserving a rich representation of the data and managing computational efficiency, ultimately improving the network's ability to generalize without incurring excessive overhead.

2)Critic-
Each "branch" (Q1 Network and Q2 Network) is designed to estimate the Q-value of a given state-action pair, thereby providing two independent estimates to help mitigate overestimation bias. Both Q1 and Q2 networks receive the same input: a concatenation of the **state** and the **action**. The first linear layer transforms the combined input into 400 features. A ReLU is applied to introduce non-linearity and help learn more complex representations.The second layer reduces 400 features to 300 features, which strikers a balance between



model capacity and computational efficiency.Another ReLU is applied here.The final output layer scales the 300 features to 1 output feature.**The algorithm typically uses the minimum of these two Q-values as the target, reducing the risk of overly optimistic value estimates.**

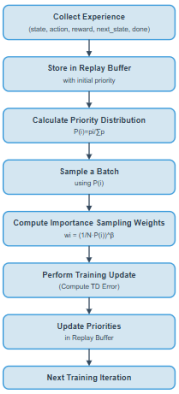
What is `def first_q()` function?
The **actor** (policy network) is updated by maximizing the Q-value estimated by only one branch of the critic—commonly the first branch (Q1). The `first_q` method provides a direct way to compute just that Q-value without running the second branch, saving computational effort. Conceptually, while the critic uses both Q1 and Q2 (taking the minimum) to reduce overestimation bias in target calculations, the policy only needs a single Q-value estimate for its gradient update. By focusing on one branch (Q1), the actor learns to select actions that maximize that Q-value. In practice, this is sufficient because the second branch (Q2) still contributes to stable learning through the target calculation, and the first branch's gradient provides a consistent learning signal for improving the policy.

Why Two Q-Networks?
Using two separate Q-value estimators (Q1 and Q2) helps mitigate the **overestimation bias** that can occur when a single network is used. By taking the minimum of the two estimates during target calculation, the agent is less likely to become overly optimistic about the value of certain actions.

3)Prioritized Replay Buffer-

A prioritized replay buffer improves sample efficiency by storing experiences with associated priority values, which are typically based on their temporal-difference (TD) error. Rather than sampling experiences uniformly, the buffer samples non-uniformly according to these priorities. In this process, the agent first collects an experience tuple, consisting of the state, action, reward, next state, and a done flag, as it interacts with the environment. Each collected experience is then stored in the replay buffer along with an initial priority, usually set to a default value such as 1.0, ensuring that all experiences start with a baseline level of importance.

Once the experiences are stored, the buffer calculates a probability distri-

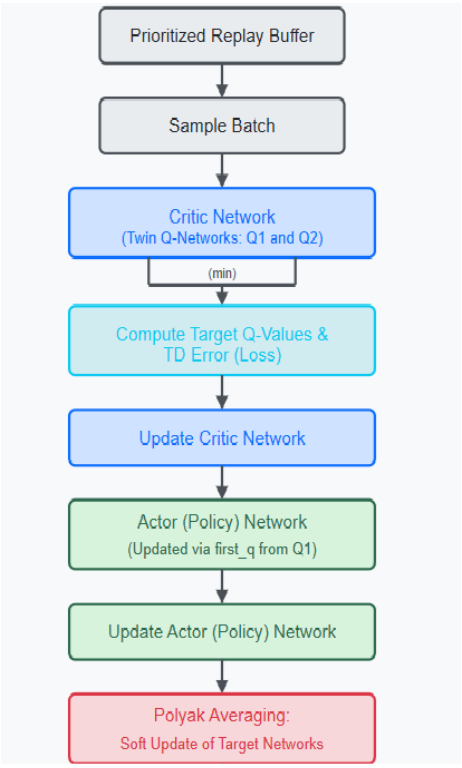


bution by normalizing the priority values. This normalization ensures that experiences with higher TD errors, and therefore greater learning potential, have a higher probability of being sampled. When a batch is required for training, the experiences are sampled based on this probability distribution, allowing the learning algorithm to focus on the more informative experiences.

To correct for the bias introduced by this non-uniform sampling, each sampled experience is assigned an importance sampling weight. These weights are used to adjust the loss during training so that the learning updates remain unbiased. During the training update, the network is updated using the sampled experiences, and the TD error is computed as part of the loss calculation. After the update, the priorities in the replay buffer are revised based on the new TD errors, ensuring that the buffer continuously reflects the most useful experiences for learning.

In addition to these steps, it is important to note that there is a bias-variance trade-off inherent in this approach. While prioritizing experiences helps to focus on more important transitions, the use of importance sampling weights mitigates the bias that this non-uniform sampling introduces. The replay buffer dynamically adapts over time as the agent's performance improves; experiences that were once surprising may become less significant as their TD errors decrease. Empirical evidence has shown that this method can accelerate learning and improve performance in complex environments. Moreover, the choice of hyperparameters, such as the priority exponent and the importance sampling exponent (β), plays a crucial role and often requires careful tuning. Although the prioritized replay buffer introduces extra computational steps compared to uniform sampling, the efficiency gains in sample quality can significantly enhance overall training performance.

4)TD3 - The `TwinDelayedAgent` class implements a TD3 agent that combines an actor (policy network) and a critic (value network) with their corresponding target networks to achieve stable learning in continuous control tasks.



Below is a summary of its key functionality, emphasizing how n-step returns and Polyak averaging are integrated into the learning process.

Initialization:

The agent initializes by creating both a policy network and a corresponding target policy network. These networks are identical at the start, with the target network's parameters set to match those of the main policy. Similarly, it creates a value network (the critic) that consists of twin Q-networks (to estimate the value of state-action pairs) and a target critic network, which is synchronized with the main critic. Two Adam optimizers are set up—one for the actor and one for the critic—to manage parameter updates during training.

Experience Sampling and Pre-processing:

In the `learn()` function, the agent samples a batch of experiences from a prioritized replay buffer. Each experience includes the state, action, reward, next state, and a terminal flag. These are converted into tensors and sent to the appropriate compute device (CPU or GPU). This batch sampling is key for efficient training and is weighted to focus on more informative experiences.

Target Q-Value Calculation with n-Step Returns:

n-steps=3

discount=0.99

The agent computes the target Q-values by first obtaining the next actions us-

ing the target policy network, then passing the next states and these actions through the target critic network. Since the critic uses twin Q-networks, the minimum of the two Q-value estimates is taken to reduce overestimation bias. The agent implements an n-step return by applying a multi-step discount factor. Instead of only considering the immediate reward, it computes the return over several steps by raising the discount factor, γ, to the power of the number of lookahead steps (i.e., `steps_lookahead`). This effectively accumulates future rewards over a short horizon, making the learning target more informative, especially in environments where rewards may be delayed. Critic Update:
With the target Q-values computed (including the multi-step return), the main critic network calculates its current Q-value estimates for the sampled state-action pairs. A loss is computed using the mean squared error between the current Q-values and the target Q-values, and gradients are back propagated to update the critic's parameters. This process ensures that the critic learns to predict more accurate state-action values. Actor Update Using `first_q`:
The actor (policy network) is updated to maximize the estimated Q-value. Instead of relying on both Q-networks, the agent uses a helper function (named `first_q`) to compute the Q-value from the first branch of the critic. The actor's loss is defined as the negative of this Q-value, so that minimizing the loss corresponds to maximizing the expected return. This provides a clear gradient signal for the actor to improve its policy.
Polyak Averaging (Soft Updates) for Target Networks**:**
After the updates of the main networks, the target networks are updated using Polyak averaging. Instead of an abrupt copy, each target network parameter is gradually adjusted using the formula:

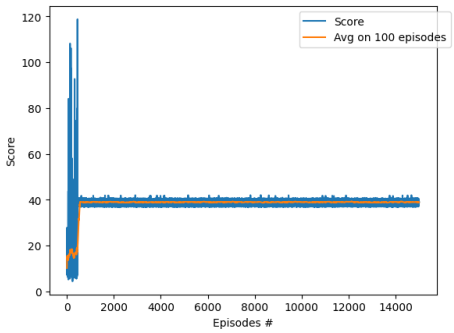
$$\theta_{target} \leftarrow \tau\, \theta_{current} + (1 - \tau)\, \theta_{target},$$

where τ is a small constant (e.g., 0.005). This soft update strategy ensures that the target networks change slowly, providing more stable targets for training the main networks. The stability afforded by Polyak averaging is crucial for preventing sudden shifts in the target Q-values, which can destabilize learning.
Updating Priorities in Replay Buffer:
Finally, after each training iteration, the agent updates the priorities of the sampled experiences in the replay buffer. This update is based on the absolute TD error between the current Q-value estimate and the computed target Q-value. Experiences with larger errors are assigned higher priorities, making them more likely to be sampled in future training iterations.
In Summary:
The `TwinDelayedAgent` class leverages n-step returns to incorporate a short-term look ahead in reward estimation and uses Polyak averaging to smoothly update target networks, thereby enhancing the stability and efficiency of the training process. The integration of prioritized experience replay ensures that the agent focuses on the most informative transitions, which together create a robust framework for learning continuous control policies in complex environments.
5)Training Loop: The `train_agent` function orchestrates the entire training

```
start_dim:  11 , action_dim:  3
n max_action:  1.0 , threshold:  3800.0 , std_noise:  0.02
```

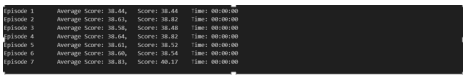
process for the TD3-based agent over a specified number of episodes. Initially, it sets up tracking variables, including a deque for recent episode rewards, arrays to store all rewards and their running average, and counters for total timesteps and save intervals. It then initializes a prioritized replay buffer and extracts action space boundaries to ensure that actions remain within valid limits. For each episode, the environment is reset to obtain the starting state, and the agent interacts with the environment by selecting actions either randomly during an exploration phase or based on the current policy (with optional noise for exploration) once enough timesteps have passed. Each action taken leads to an interaction with the environment where the next state, reward, and termination status are recorded; the resulting experience tuple is stored in the replay buffer. After the episode concludes, the function updates performance statistics and logs progress periodically. When sufficient experiences are available, the agent's networks are trained using the sampled batch: the critic network is updated by computing target Q-values with n-step returns and minimizing the mean squared error between these targets and the current estimates, while the actor is updated to maximize the Q-value computed from the critic's first branch. Polyak averaging is applied to softly update the target networks, ensuring stable training by gradually blending the main network parameters into the target networks. Finally, the priorities in the replay buffer are updated based on the new temporal-difference errors, and the model is saved periodically, with the training loop terminating early if the average reward exceeds a predefined threshold.

Results:
The following code was run for 15000 episodes, discount factor=0.99 and n-steps=3 and it gave the following results- This plot shows us that when run



for 15000 episodes the average scores/rewards converge to nearly 39/40 whereas maximum scores being 120 recorded in earlier episodes.

When the trained model was used to run Hopper-V4 to generate results what we found out was- We can see that all the scores converge to 38-39 which are similar to the average scores in each episode.



Conclusion- The results observed in the training graphs and logs can be explained through two key design choices in the agent's learning process: **n-step returns** and **Polyak averaging for target network updates**. These mechanisms significantly influence how the agent learns from its environment, explaining both the initial rapid improvement and the eventual stabilization of performance.

1. n-Step Returns and Its Impact on Learning

The agent employs n-step returns (with `steps_lookahead = 3`), meaning that it accumulates rewards over the next three steps rather than considering only the immediate reward. Mathematically, this can be expressed as:

$$R = r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \gamma^3\, Q(s_{t+3}, a_{t+3}),$$

where γ is the discount factor. This approach provides a richer, more informative learning signal early in training, as it allows the agent to capture the effect of its actions over a short horizon. Consequently, the agent quickly shifts from random exploration to adopting a strategy that yields higher rewards, which is reflected in the rapid increase in scores during the initial episodes.

2. Polyak Averaging and Stable Convergence

While n-step returns boost early learning, Polyak averaging ensures stability in later stages. Instead of replacing the target network's parameters abruptly, Polyak averaging updates them gradually using the formula:

$$\theta_{target} \leftarrow \tau\, \theta_{current} + (1 - \tau)\, \theta_{target},$$

where τ is a small constant (typically 0.005). This soft update strategy prevents sudden shifts in target Q-values, which could destabilize training. By slowly blending the current network parameters into the target networks, the method produces a stable reference for computing loss functions. This controlled, gradual change is key to the observed plateau in performance, as it prevents large fluctuations once the agent converges to a reliable policy.

3. How TD3 Enhances Performance

The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm incorporates several improvements over traditional actor-critic methods, which further enhance the performance of the agent. First, TD3 uses **twin Q-networks** in the critic, which provide two separate Q-value estimates. By taking the minimum of these two estimates during target computation, TD3 effectively reduces overestimation bias—a common problem in value-based methods. Additionally, TD3 delays the policy (actor) updates relative to the critic updates, which helps stabilize learning by ensuring that the critic's estimates are sufficiently accurate before they are used to update the actor. These enhancements allow the agent to learn more robust policies and achieve stable, high-performing behavior in complex continuous control tasks.

Interpreting the Results

The training graphs initially show high variance in rewards, with some episodes achieving very high scores. However, as training progresses, the rewards settle around a consistent value (approximately 40 in this case). This behavior is explained by the rapid improvement driven by n-step returns and the subsequent stabilization provided by Polyak averaging. TD3's use of twin Q-networks and delayed updates further refines this process, reducing overoptimistic estimates and preventing oscillations in policy performance.

In summary, the combination of n-step returns and Polyak averaging allows the TwinDelayedAgent to quickly identify a high-reward strategy and then maintain it with minimal fluctuations. The additional architectural improvements from TD3—namely the twin Q-networks and delayed policy updates—further enhance the agent's performance by ensuring more accurate value estimates and smoother learning dynamics. Together, these techniques create a robust framework for learning continuous control policies in complex environments, as evidenced by the stable, reproducible performance observed in the training logs and graphs.