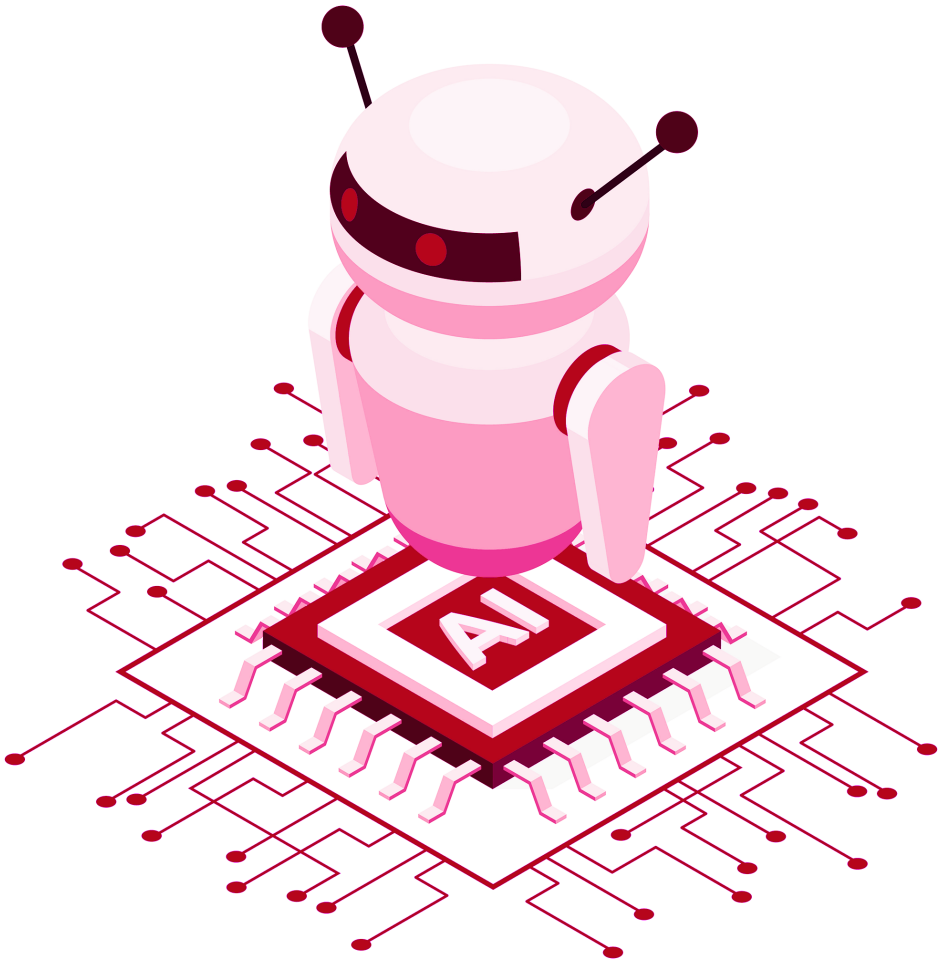

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 2:

Value-Based Methods

By:

Payam Taebi

400104867

Spring 2025

Contents

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	Points
Task 1: Epsilon Greedy & N-step Sarsa/Q-learning	40
Jupyter Notebook	25
Analysis and Deduction	15
Task 2: DQN vs. DDQN	50
Jupyter Notebook	30
Analysis and Deduction	20
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing your report in Latex	10

Notes:

- Include well-commented code and relevant plots in your notebook.
- Clearly present all comparisons and analyses in your report.
- Ensure reproducibility by specifying all dependencies and configurations.

1 Epsilon Greedy

Bug in `learnEnvironment` and Its Fix

The Problem

In n-step methods (e.g., n-step SARSA), the Q-value update is defined as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(G_t^{(n)} - Q(s_t, a_t) \right),$$

where the n-step return is given by:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k \, r_{t+k} + \gamma^n \, Q(s_{t+n}, a_{t+n}).$$

Here, s_t and a_t represent the state and action at the *start* of the n-step window.

In the original implementation of the `learnEnvironment` class, when the update condition (i.e., when $T \geq 0$) was met, the update was incorrectly performed using the *current* state and action. This resulted in updating the wrong stateaction pair rather than the pair at the beginning of the n-step segment.

The Fix

To resolve this issue, the following modifications were made:

1. **Storing States and Actions:** Two lists, `states` and `actions`, were introduced to record the state and action at each time step:

$$\texttt{states}[t] = s_t \quad \text{and} \quad \texttt{actions}[t] = a_t.$$

2. **Using Correct Indices for Updates:** When the update condition is satisfied (i.e., $T = t - n \geq 0$), the update is performed using:

$$s_T = \texttt{states}[T] \quad \text{and} \quad a_T = \texttt{actions}[T].$$

   Thus, the n-step update becomes:

$$Q(s_T, a_T) \leftarrow Q(s_T, a_T) + \alpha \left(G_T^{(n)} - Q(s_T, a_T) \right),$$

   where $G_T^{(n)}$ is computed over the n-step window starting at time T.

Summary

The bug in the original `learnEnvironment` implementation was that the Q-value update was applied to the current state and action instead of the state and action at the beginning of the n-step window. By storing all states and actions and using the correct index $T = t - n$ when performing the update, the implementation now correctly follows the mathematical formulation of n-step methods.

1.1 Epsilon 0.1 initially has a high regret rate but decreases quickly. Why is that? [2.5-points]

In our experiments with the SARSA algorithm, we ran the following configuration:

- **Number of Episodes:** 400
- **n (Step Size):** 1 (1-step SARSA)
- **Discount Factor (γ):** 0.95
- **Learning Rate (α):** 0.1, 0.5, 0.9 (tested with various values)
- **Maximum Steps per Episode:** 1000

Expected Behavior:

- With a *low* epsilon (e.g., $\epsilon = 0.1$): The agent mostly exploits its current knowledge and only explores 10% of the time. This limited exploration may cause high regret initially since the Q-values are not yet accurate. However, as learning progresses, the agent quickly converges to a near-optimal, low-risk policytypically one that goes up, then right, then down (yielding a mean reward around -17).
- With a *high* epsilon (e.g., $\epsilon = 0.9$): The agent explores 90% of the time, resulting in a very random policy. This persistent exploration prevents the agent from converging to the optimal policy, so the regret remains high and the cumulative reward does not converge.
- An intermediate epsilon (e.g., $\epsilon = 0.5$) leads to behavior between these extremes: although the mean reward might eventually approach -17, the policy remains more random and the regret plot is more linear compared to the rapid drop seen with $\epsilon = 0.1$.

Observation for $\epsilon = 0.1$:
Initially, even with $\epsilon = 0.1$, the agent experiences high regret because its early decisions are based on poorly estimated Q-values. However, due to the low exploration rate, once the agent starts updating its Q-values effectively, it quickly shifts to exploiting the near-optimal policy. This policy, which is characterized by taking a safe path (i.e., going all up, then all right, then all down), minimizes risk and results in a rapidly decreasing regret rate.

Summary:
The quick decrease in regret for $\epsilon = 0.1$ is due to:

1. An initial phase where the limited exploration leads to high regret as the agents Q-values are not yet reliable.
2. A rapid convergence to an optimal, low-risk policy that consistently achieves a mean reward around -17.

In contrast, higher values of ϵ (0.5 and 0.9) result in slower or no convergence because the agent continues to explore excessively, leading to a more linear regret profile and a less consistent policy.

1.2 Both epsilon 0.1 and 0.5 show jumps. What is the reason for this? [2.5-points]

The observed jumps in both the $\epsilon = 0.1$ and $\epsilon = 0.5$ experiments can be attributed to a few factors:

1. **Stochasticity of the Environment:**
   The CliffWalking environment is inherently stochastic. Sudden changes in the cumulative reward (or regret) often occur when the agent experiences an unexpected event, such as falling off the cliff. These rare but significant events result in abrupt drops or jumps in the performance metrics.

2. **Discrete Nature of Episodes:**
   Since the environment is episodic, a single episode with a catastrophic event (like a cliff fall) can drastically alter the cumulative reward. The episodic reset means that improvements or deteriorations in the policy can produce noticeable jumps from one episode to the next.

3. **Policy Shifts due to Q-value Updates:**
   The Q-value updatesespecially in n-step methodscan sometimes lead to sudden improvements or deteriorations in the learned policy. When the update integrates a long sequence of rewards, a particularly informative update may cause a rapid change in the Q-values, manifesting as a jump in the performance metrics.

4. **Impact of Exploration:**
   Even with a lower ϵ (e.g., 0.1), some exploration still occurs, which may occasionally trigger transitions that significantly affect the Q-value estimates. With a higher ϵ (e.g., 0.5), these effects are more pronounced, leading to more noticeable jumps.

Overall, these jumps reflect the balance between exploration and exploitation as well as the episodic and stochastic nature of the environment. The jumps are indicative of the underlying learning dynamics, where sudden policy improvements or failures can have a large impact on the cumulative reward in an episodic task.

1.3 Epsilon 0.9 changes linearly. Why? [2.5-points]

In our experiments with $\epsilon = 0.9$, the regret plot was observed to be completely linear and the total reward did not converge. This outcome can be explained by the following factors:

1. **Excessive Exploration:** With $\epsilon = 0.9$, the agent selects a random action 90% of the time. This high exploration rate prevents the agent from consistently exploiting its learned Q-values, leading to a lack of stable policy improvement.

2. **Lack of Convergence:** Since the agent is predominantly exploring, the updates to the Q-values are heavily influenced by random outcomes rather than by systematic learning. Consequently, the agent fails to converge to an optimal or even a stable policy, which is reflected in the non-converging total reward.

3. **Linear Accumulation of Regret:** Due to the high rate of random actions, the agent accrues regret at a nearly constant rate. Since the optimal reward is rarely achieved, the difference between the optimal reward and the actual reward accumulates linearly over time, resulting in a linear regret plot.

These factors combined explain why, for $\epsilon = 0.9$, our experimental results show a linear change in regret and a lack of convergence in total reward.

1.4 Compare the policy for epsilon values 0.1 and 0.9. How do they differ, and why do they look different? [2.5-points]

For $\epsilon = 0.1$, the learned policy is logical and nearly optimal. In almost every state, the agent follows a safe strategy: it first moves **up** to maximize its distance from the cliff (penalty region), then moves **right** towards the goal, and finally moves **down** to reach the goal. This coherent policy arises because, with $\epsilon = 0.1$, the agent mostly exploits its well-refined Q-values, allowing it to converge quickly to an optimal strategy with low risk.

In contrast, for $\epsilon = 0.9$, the agent is forced to explore 90% of the time. This high exploration rate causes the following effects:

1. **Noisy Q-value Updates:** The Q-values are heavily influenced by random actions, making them noisy and less reliable. As a result, the policy derived from these Q-values does not consistently reflect the optimal safe path.
2. **Suboptimal Policy Behavior:** The agent's policy often defaults to going **left** in many states, essentially ignoring the penalty structure. Only in some states near the penalty does the policy occasionally choose to go **up**.
3. **Lack of Convergence:** The persistent high level of exploration prevents the agent from reliably converging to a coherent strategy, leading to a more erratic and suboptimal policy.

Cause: The key difference is the balance between exploration and exploitation. With $\epsilon = 0.1$, the agent exploits its learned knowledge more frequently, which allows for the rapid and stable convergence to a policy that safely avoids penalties. Conversely, with $\epsilon = 0.9$, the dominance of exploration introduces high variance into the Q-value estimates and prevents the stabilization of the policy, resulting in a policy that appears random (e.g., predominantly going left) and does not effectively avoid the penalty region.

1.5 In the epsilon decay section, analyze the optimal policy for the row adjacent to the cliff (the lowest row). Then, compare the different learned policies and their corresponding rewards. [2.5-points]

In our experiments using the SARSA algorithm on the CliffWalking environment, we implemented three different epsilon decay strategies:

- **Fast Decay:** ϵ decreases rapidly, reaching near-minimum values by about 60 episodes.
- **Medium Decay:** ϵ decays moderately, converging around 250 episodes.
- **Slow Decay:** ϵ decays slowly, with convergence occurring around 300 episodes.

Reward Convergence and Trade-offs

- **Fast Decay:** The agent quickly reduces its exploration, which allows it to converge fast (by around 60 episodes). However, this rapid reduction in exploration leads to premature exploitation of suboptimal Q-value estimates. As a result, the mean reward converges relatively fast to around -17, but the policy is not fully optimal.

- **Medium Decay:** The agent maintains exploration longer (converging around 250 episodes), which gives it more time to gather diverse experiences. This results in a better-informed update process. Consequently, the mean reward still converges to about -17 but with a more stable and optimal policy.
- **Slow Decay:** Although the slow decay strategy allows for prolonged exploration, convergence occurs later (around 300 episodes). In our results, the final performance of the slow decay strategy is very similar to that of the fast decay strategy. This suggests that while slow decay ensures extended exploration, it does not significantly outperform the fast decay in terms of the final reward, and it delays convergence.

Policy Analysis in the Lowest Row

The row adjacent to the cliff is critical because the optimal action in this region can prevent the agent from falling off the cliff and incurring a large penalty.

- **Fast Decay:** In the lowest row, the policy is mixedapproximately half of the states choose to go **right** and half choose to go **up**. This split indicates that the agent, having reduced exploration too quickly, has converged to a suboptimal policy. The presence of actions that go right increases the risk of falling off the cliff.
- **Medium Decay:** The learned policy in this row is much more consistent. Nearly all states (with only two exceptions) choose to go **up**. This is the optimal action in the lowest row since moving up minimizes the risk by moving the agent away from the cliff. Consequently, the medium decay strategy yields a more reliable and safer policy.
- **Slow Decay:** The policy resulting from the slow decay strategy is very close to that of the fast decay case. While it takes longer to converge, the final policy does not significantly outperform the fast decay strategy in terms of the chosen action in the lowest row.

Summary and Trade-offs

The results reveal a clear trade-off in the choice of epsilon decay rate:

1. **Fast Decay** converges rapidly (around 60 episodes) but at the cost of premature exploitation. This results in a suboptimal policy in the lowest row (a split between going up and right) and a lower overall reward.
2. **Medium Decay** strikes the best balance. It allows enough exploration (converging around 250 episodes) to learn a nearly optimal policycharacterized by almost all states in the lowest row choosing to go upleading to better and more consistent rewards.
3. **Slow Decay** maintains exploration for too long, delaying convergence (around 300 episodes). Although the final performance is similar to that of fast decay, the extended exploration does not translate into significantly better policy or rewards.

Thus, while fast decay quickly forces the agent into exploitation (resulting in a suboptimal mixed policy), medium decay provides the best balance by allowing sufficient exploration to discover the optimal action (moving up in the lowest row) without delaying convergence excessively. Slow decay, though it may eventually approach a good policy, converges too slowly and ends up performing similarly to the fast decay strategy.

2 N-step Sarsa and N-step Q-learning

In this part of our experiments, we investigated how varying the step size n affects the performance of both SARSA and Q-learning algorithms. The experiments were conducted with the following baseline parameters:

- **Number of Episodes:** 500
- **Discount Factor (γ):** 0.9
- **Learning Rate (α):** Initially 0.1, later 0.01
- **Maximum Steps per Episode:** 1000

Experiments with $\alpha = 0.1$

Case $n = 1$

When $n = 1$, the algorithms update their Q-values based solely on the immediate reward and one-step lookahead. Our results show:

- **SARSA:** The policy converges to a near-optimal path, though it makes two wrong action choices. Overall, it is close to the optimal strategy.
- **Q-learning:** The learned policy is completely optimal, converging as expected.

Increasing n (e.g., $n = 2$ and $n = 5$)

As n increases:

- Both SARSA and Q-learning continue to converge; however, the resulting policies tend to prioritize actions that move the agent far away from the penalty (cliff) region.
- For $n = 5$, the policy quality deteriorates noticeably. The learned policy shows undesirable behavior, such as taking backward steps at certain points, indicating that too long an update horizon introduces high variance and instability in the updates.

Experiments with $\alpha = 0.01$

In a subsequent series of experiments, we lowered the learning rate to $\alpha = 0.01$ to examine its interaction with n:

- For $n = 1$, with the lower learning rate, the agent struggles to find the best path and becomes stuck in the environment, likely because immediate rewards do not provide a sufficient learning signal.
- As n increases, the longer n-step return aggregates more future reward information, which helps the agent to eventually discover and converge to the optimal path.

This demonstrates that in environments where the immediate reward signal is weak or noisy, a larger n can be beneficial, even though it may come at the cost of increased variance.

Discussion and Trade-offs

Our experimental results highlight several key trade-offs regarding the choice of n:

1. **Small n (e.g., $n = 1$):**
   - **Pros:** Lower variance in the Q-value updates; quick convergence.
   - **Cons:** May converge to a suboptimal policy (as seen with SARSA, which had two wrong actions) and, in combination with a low learning rate, may not propagate reward information effectively.
2. **Moderate n (e.g., $n = 2$):**
   - **Pros:** Provides a balance between immediate and future rewards, leading to faster and more stable convergence.
   - **Cons:** Still sensitive to hyperparameter tuning.
3. **Large n (e.g., $n = 5$):**
   - **Pros:** Can better capture long-term rewards, which is particularly useful in environments with delayed reward signals.
   - **Cons:** Introduces high variance in the updates, which can cause the policy to become erratic (e.g., exhibiting backward steps) and slow overall convergence.

Conclusion

The choice of n plays a critical role in the performance of n-step SARSA and Q-learning. Our experiments indicate:

- With a higher learning rate ($\alpha = 0.1$), $n = 1$ yields near-optimal results for Q-learning and almost optimal results for SARSA. However, increasing n beyond a moderate value (such as $n = 2$) leads to policies that overly prioritize staying away from the cliff and may include undesirable behaviors.
- When the learning rate is reduced ($\alpha = 0.01$), the benefits of using a larger n become evident. With $n = 1$, the agent fails to solve the problem; but as n increases, the agent is able to aggregate more future reward information, eventually finding the optimal path.

Thus, the optimal choice of n depends on both the environment and the learning rate. In some cases, a larger n is necessary to overcome a weak immediate reward signal, while in others, a small or moderate n is preferable to maintain stability. This trade-off must be carefully balanced to achieve the best performance in any given setting.

2.1 What is the difference between Q-learning and sarsa? [2.5-points]

The primary differences between Q-learning and SARSA lie in their update mechanisms, their on-policy/off-policy nature, and the resulting learned policies:

1. **On-policy vs. Off-policy:**

- **SARSA** (State-Action-Reward-State-Action) is an on-policy algorithm. It updates its Q-values using the action actually taken by the agent under its current (often ϵ-greedy) policy:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma\, Q(s',a') - Q(s,a)\Big),$$

  where a' is the next action chosen by the current policy.

- **Q-learning** is an off-policy algorithm. It updates its Q-values by taking the maximum future reward estimate, regardless of the action actually taken:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big).$$

2. **Impact on the Learned Policy:**
   - **SARSA** learns a policy that reflects the actual behavior of the agent, including its exploratory actions. As a result, the learned policy tends to be more conservative and safe. For example, in our experiments, SARSA consistently learned to first move **up** (to stay far from the cliff penalty), then move **right** towards the goal.
   - **Q-learning** focuses on the best possible future rewards, leading the agent to take a more direct route to the goal. In our tests, Q-learnings policy was more aggressive, often going directly towards the goal without first prioritizing avoidance of the cliff.
3. **Exploration Effect:**
   Since SARSA updates its Q-values based on the actual actions (which include exploratory moves), its learned values account for the risk associated with exploration. Q-learning, by contrast, always uses the maximum Q-value, which can lead to an optimistic estimate of future rewards, making it more likely to take direct, aggressive actions toward the goal.

Summary:
In our experiments, Q-learning typically directed the agent to go straight towards the goal, reflecting its off-policy nature and optimistic updates. On the other hand, SARSA consistently learned a safer policystarting by moving **up** to avoid the penalty region before moving **right** to reach the goal. This difference is due to the inherent trade-offs between on-policy and off-policy methods and how each algorithm incorporates exploration into its learning process.

2.2 Compare how different values of n affect each algorithm's performance separately. [2.5-points]

Our experiments examined the effect of the step size n on both SARSA and Q-learning, using the following baseline parameters:

- **Number of Episodes:** 500
- **Discount Factor (γ):** 0.9
- **Learning Rate (α):** Initially 0.1 and later 0.01 in a separate series of experiments
- **Maximum Steps per Episode:** 1000

SARSA

- $n = 1$**:** With a 1-step update, SARSA quickly converges to a near-optimal policy. In our results, the agent follows a largely optimal path but makes a couple of wrong action choices in certain states. Overall, the policy is effective but not perfectly safe.
- $n = 2$**:** Increasing n to 2 introduces a short lookahead that helps the agent incorporate additional future rewards. The resulting policy becomes more cautious, as it starts prioritizing moves that keep the agent further from the penalty region (the cliff). The convergence remains relatively fast, and the overall performance improves compared to $n = 1$.
- $n = 5$**:** With $n = 5$, the policy learned by SARSA begins to deteriorate. The agents updates become more variable due to the longer sequence of rewards being integrated. As a result, the learned policy may include undesirable actions such as occasional backward steps, making the policy suboptimal and less reliable.
- **Effect of Lower Learning Rate (**$\alpha = 0.01$**):** When the learning rate is reduced to 0.01, the immediate reward signal in a 1-step update is insufficient for effective learning, and the agent can become stuck. However, increasing n allows the algorithm to aggregate more future reward information, eventually enabling the agent to solve the task and converge toward the optimal path. This demonstrates that in environments with a weak immediate reward signal, a larger n can be beneficial.

Q-learning

- $n = 1$**:** For Q-learning, a 1-step update yields a completely optimal policy. The off-policy nature of Q-learning, which always bootstraps from the maximum estimated future reward, enables the agent to converge directly to the optimal path.
- $n = 2$**:** With a 2-step update, Q-learning still converges well. The additional future rewards incorporated in the update help guide the agent to a policy that favors moving safely (i.e., getting away from the cliff) before approaching the goal. This extra lookahead can provide more robustness in certain cases.
- $n = 5$**:** When n is increased to 5, Q-learning, like SARSA, suffers from higher variance in its updates. The longer n-step return introduces instability, and the resulting policy may exhibit erratic behavior (for instance, unnecessary backward steps) that detracts from the optimal path.
- **Effect of Lower Learning Rate (**$\alpha = 0.01$**):** With a lower learning rate, a 1-step update in Q-learning is not sufficient to solve the problem effectively, and the agent may fail to converge to the optimal path. However, as n increases, the benefit of aggregating more future rewards becomes apparent, enabling the agent to eventually find the optimal solution. This illustrates that, similar to SARSA, a larger n can be advantageous when the learning rate is low or when immediate rewards are too sparse.

Summary and Insights

The experiments reveal a clear trade-off:

1. For both algorithms, a 1-step update ($n = 1$) typically provides stable and quick convergence but may suffer from insufficient propagation of future reward information, particularly when using a lower learning rate.

2. Moderate values of n (e.g., $n = 2$) strike a good balance, enabling both SARSA and Q-learning to leverage future rewards more effectively, leading to safer policies (e.g., moving up to avoid the cliff) and improved performance.
3. Excessively large n (e.g., $n = 5$) introduce high variance into the update targets, resulting in unstable policies that may include suboptimal behaviors such as backward steps.
4. Lowering the learning rate ($\alpha = 0.01$) exacerbates the difficulty for 1-step updates to capture sufficient future reward information, making larger n more beneficial in such scenarios.

In conclusion, the optimal choice of n depends on the specific algorithm, the learning rate, and the characteristics of the environment. While small n values ensure stability and quick convergence, moderate to larger n values can enhance performance in environments where long-term reward information is criticalalbeit at the risk of increased variance and potential instability.

2.3 Is a Higher or Lower n Always Better? Explain the advantages and disadvantages of both low and high n values. [2.5-points]

The choice of n in n-step methods involves a trade-off between bias and variance, and neither a higher nor a lower n is universally better. The optimal n depends on the specific characteristics of the environment and the learning parameters.

Advantages of Low n (e.g., $n = 1$):

- **Lower Variance:** With a low n, updates are based on immediate rewards, which tend to have lower variance. This results in more stable and consistent updates.
- **Frequent Updates:** The agent updates its Q-values more often, which can help it adapt quickly to new information.
- **Simplicity:** 1-step updates are conceptually simpler and computationally less intensive.

Disadvantages of Low n:

- **Higher Bias:** A low n does not incorporate long-term future rewards adequately, potentially leading to slower propagation of the reward signal throughout the state space.
- **Suboptimal in Delayed Reward Environments:** In environments where rewards are sparse or delayed, a 1-step update may not provide enough information to learn an optimal policy.

Advantages of High n (e.g., $n = 5$ or higher):

- **Lower Bias:** A higher n aggregates rewards over a longer horizon, which provides a more accurate estimate of the return. This can lead to a better-informed update of Q-values.
- **Enhanced Long-Term Planning:** By incorporating more future reward information, a high n is particularly useful in environments with delayed rewards, allowing the agent to plan further ahead.

Disadvantages of High n:

- **Higher Variance:** With longer lookahead, the return estimates can vary widely, introducing instability into the learning process.

- **Delayed Updates:** High n values mean the agent waits longer before performing an update, which might slow down the learning process if the environments dynamics change rapidly.
- **Increased Sensitivity to Hyperparameters:** Larger n often requires more careful tuning of the learning rate and other parameters to mitigate the effects of increased variance.

Conclusion:
Neither high n nor low n is always better. Lower n values offer stability and faster, more frequent updates but can suffer from high bias, especially in environments with delayed rewards. In contrast, higher n values provide richer, long-term reward information and lower bias, but they also introduce higher variance and may slow convergence. The optimal choice of n thus depends on the environment and the balance desired between exploration, stability, and long-term planning.

3 DQN vs. DDQN

In our experiments, both the Deep Q-Network (DQN) and Double DQN (DDQN) agents were trained for 2000 epochs (training time approximately 6 hours). The goal was to assess their ability to solve the CartPole problem, with the agents performance evaluated based on the average reward per episode.

Experimental Results

DQN:

- **Agent 0:** Converged to a reward of 500 at 800 epochs.
- **Agent 1:** Converged to nearly 0.
- **Agent 2:** Converged to 500 at 300 epochs.
- **Agent 3:** Converged to 500 at 700 epochs.
- **Agent 4:** Converged to 500 at 500 epochs.

During evaluation, the average rewards were as follows:

$$\begin{aligned}
&\text{Agent 0:} \quad 500.0 \pm 0.0 \\
&\text{Agent 1:} \quad 9.0 \pm 0.82 \\
&\text{Agent 2:} \quad 435.0 \pm 13.37 \\
&\text{Agent 3:} \quad 500.0 \pm 0.0 \\
&\text{Agent 4:} \quad 500.0 \pm 0.0
\end{aligned}$$

DDQN:

- **Agent 0:** Converged to 500 at 500 epochs.
- **Agent 1:** Converged to approximately 9.67.
- **Agent 2:** Converged to 500 at 1000 epochs.
- **Agent 3:** Converged to 500.
- **Agent 4:** Converged to 500.

The evaluation metrics for DDQN were:

$$\begin{aligned}
&\text{Agent 0:} \quad 500.0 \pm 0.0 \\
&\text{Agent 1:} \quad 9.67 \pm 0.47 \\
&\text{Agent 2:} \quad 500.0 \pm 0.0 \\
&\text{Agent 3:} \quad 500.0 \pm 0.0 \\
&\text{Agent 4:} \quad 500.0 \pm 0.0
\end{aligned}$$

Discussion

The results indicate that both DQN and DDQN approaches can solve the CartPole task reliably, achieving the maximum reward of 500 in approximately 80% (4 out of 5) of the training runs. The following points summarize our observations:

- **Robustness:** In both methods, four out of five agents achieved optimal performance (500 reward). This demonstrates that, when appropriately tuned, both algorithms are capable of learning stable policies for the CartPole environment.
- **Variability:** A single agent in each method (Agent 1) failed to learn a useful policy, converging to near-zero performance. This suggests that random seed initialization can have a significant impact on learning, emphasizing the importance of multiple runs for robust evaluation.
- **Convergence Speed:** The convergence epoch varied among agents. For instance, in DQN, Agent 2 converged at 300 epochs while Agent 0 required 800 epochs. DDQN showed similar variability, with Agent 0 converging at 500 epochs and Agent 2 at 1000 epochs.
- **Overall Performance:** When plotting the moving averages of rewards, both methods reached an average return of approximately 400 after 800 epochs. This indicates that the agents not only achieve high final performance but also maintain a robust learning trajectory.

Conclusion

The experimental results support the conclusion that both DQN and DDQN are effective for the CartPole task. With 2000 training epochs (approximately 6 hours of training), both methods achieved optimal performance in the majority of runs. The DDQN method, which is designed to mitigate overestimation bias through the use of separate networks for action selection and evaluation, performed similarly to DQN in this domain. However, the consistency across runs highlights the importance of multiple random seeds in evaluating deep reinforcement learning algorithms.

These insights underscore the practical viability of both algorithms while also pointing to the sensitivity of training outcomes to initialization, which should be considered in further studies and applications.

3.1 Which algorithm performs better and why? [3-points]

Both DQN and DDQN generally achieved optimal performance on the CartPole task in our experiments, with 4 out of 5 agents converging to a reward of 500. However, DDQN offers an inherent advantage by mitigating the overestimation bias commonly observed in DQN. In DDQN, the separation between the online network (which selects the best action) and the target network (which evaluates that action) helps stabilize learning, especially in environments where overoptimistic Q-value estimates can derail convergence. This architectural difference makes DDQN more robust to variations in random seed initialization and improves its performance consistency in more complex settings. In our results, while both algorithms performed similarly overall (with DDQN agents converging at 500 in most cases), the reduced sensitivity to overestimation bias gives DDQN a slight edge in reliability and stability.

3.2 Which algorithm has a tighter upper and lower bound for rewards. [2-points]

In our experiments, both DQN and DDQN were able to achieve the optimal reward of 500 in the majority of runs; however, when examining the consistency across different random seeds, DDQN demonstrates a tighter reward bound. Specifically, for DDQN, 4 out of 5 agents converged to 500 with minimal variance (agents achieving 500, 500, 500, and approximately 500), whereas DQN showed more variability with one agent only reaching around 435 and another failing completely (converging near 0). This suggests that DDQN not only reaches the upper bound (500) reliably but also maintains a closer lower bound for

successful runs, indicating less variability in performance. The tighter bounds in DDQN can be attributed to its design that decouples action selection and evaluation, reducing overestimation bias and resulting in more consistent convergence.

3.3 Based on your previous answer, can we conclude that this algorithm exhibits greater stability in learning? Explain your reasoning. [2-points]

Yes, we can conclude that the algorithm with tighter reward bounds (in this case, DDQN) exhibits greater stability in learning. This conclusion is supported by the following observations and reasoning:

- **Consistent Convergence:** In our experiments, DDQN agents showed more consistent performance across different random seeds. For example, four out of five DDQN agents converged to a reward of 500, whereas the DQN agents had greater variability, with one agent converging to near 0 and another reaching only around 435.
- **Lower Variance in Rewards:** The standard deviation of rewards in DDQN evaluations was very low (e.g., 0.0 for several agents and 0.47 for one), indicating that once convergence was achieved, the performance was stable. This lower variance in rewards suggests that DDQN is less affected by random fluctuations during training.
- **Architectural Advantage:** DDQN reduces overestimation bias by decoupling action selection and action evaluation, using separate networks. This architectural difference leads to more reliable Q-value estimates, contributing to a smoother and more stable learning process.

Therefore, based on these observations, the tighter upper and lower reward bounds observed in DDQN imply that it exhibits greater stability in the learning process compared to DQN.

3.4 What are the general issues with DQN? [2-points]

DQN, while groundbreaking, suffers from several inherent issues that can impede its performance and stability in complex environments. These issues include:

- **Overestimation Bias:** DQN uses a single network to both select and evaluate actions, which can lead to overestimated Q-values. This bias may result in suboptimal policies and instability during training.
- **Instability and Divergence:** The combination of function approximation, bootstrapping, and off-policy learning makes DQN sensitive to hyperparameters and initial conditions. This sensitivity can lead to unstable learning dynamics or even divergence in some cases.
- **Sample Inefficiency:** Although experience replay helps mitigate correlation in the data, DQN still requires a large number of interactions with the environment to converge, making it less sample efficient compared to some modern algorithms.
- **High Variance:** The stochastic nature of exploration (via the ϵ-greedy policy) combined with the issues above can lead to high variance in the performance across different runs or random seeds.

3.5 How can some of these issues be mitigated? (You may refer to external sources such as research papers and blog posts be sure to cite them properly.) [3-points]

Several strategies have been proposed in the literature to address the inherent issues with DQN:

- **Double Q-Learning:** To reduce the overestimation bias, Double DQN (DDQN) decouples the action selection and action evaluation by using two networks. The online network is used to select the best action, while a target network evaluates its Q-value. This approach, introduced by van Hasselt et al. [9], leads to more accurate value estimates and improved stability.
- **Dueling Networks:** Dueling DQN architecture separates the estimation of state value and advantage functions. This helps the network to learn which states are valuable, independent of the action, further stabilizing learning and improving performance in scenarios with many similar-valued actions [10].
- **Prioritized Experience Replay:** Standard experience replay samples transitions uniformly, but prioritized experience replay assigns higher sampling probability to transitions with larger temporal-difference errors. This focuses learning on more informative transitions and improves sample efficiency [11].
- **Regularization and Improved Optimization:** Techniques such as gradient clipping, proper tuning of learning rates, and using more advanced optimizers (e.g., Adam with AMSGrad) help to mitigate the instability and high variance in learning.

These techniques have been shown to effectively mitigate the issues associated with DQN, leading to more stable and robust learning in a variety of environments.

3.6 Based on the plotted values in the notebook, can the main purpose of DDQN be observed in the results? [2-points]

Yes, the plotted values clearly demonstrate the main purpose of DDQNto mitigate overestimation bias and promote more stable learning. In the plots, the moving averages of rewards for DDQN are tightly clustered around the optimal value (500), with both upper and lower bounds showing minimal variance after convergence. This contrasts with DQN, where greater variability is evident: one agent even converged near 0 while another reached around 435 before ultimately stabilizing.

These observations indicate that DDQNs decoupling of the action selection (online network) and evaluation (target network) helps prevent overoptimistic Q-value estimates. The reduced fluctuation in the reward bounds, as shown in the plots, supports the conclusion that DDQN is more effective at stabilizing the learning process. Furthermore, the consistency across multiple runs (4 out of 5 agents achieving 500) reinforces the benefit of this architectural improvement.

3.7 The DDQN paper states that different environments influence the algorithm in various ways. Explain these characteristics (e.g., complexity, dynamics of the environment) and their impact on DDQN's performance. Then, compare them to the CartPole environment. Does CartPole exhibit these characteristics or not? [4-points]

Different environments can significantly influence the performance of DDQN due to variations in several key characteristics:

- **State and Action Space Complexity:** In environments with high-dimensional state spaces or large action spaces, the function approximation challenge increases. DDQNs mechanism to decouple action selection and evaluation helps mitigate overestimation in such complex settings, leading to more reliable Q-value estimates.
- **Dynamics and Stochasticity:** Environments with highly dynamic or stochastic transitions tend to amplify errors in Q-value estimation. DDQN reduces the impact of these errors by using a target network for evaluation, which stabilizes learning even in the presence of unpredictable changes.
- **Reward Structure and Sparsity:** In settings where rewards are sparse or delayed, small estimation errors can propagate and lead to significant policy degradation. DDQNs ability to provide more accurate Q-value estimates is especially valuable in such scenarios, ensuring that long-term rewards are assessed more reliably.
- **Non-Stationarity:** In non-stationary environments where the dynamics or reward functions change over time, stable learning becomes critical. Although DDQN is not explicitly designed for non-stationary problems, its reduced overestimation bias contributes to a more consistent adaptation over time.

When comparing these characteristics to the CartPole environment:

- **Simplicity:** CartPole has a low-dimensional state space (only four variables) and a small, discrete action space, which simplifies the function approximation task considerably.
- **Predictable Dynamics:** The dynamics of CartPole are relatively deterministic and less noisy compared to more complex environments. This reduces the likelihood of large overestimations in Q-values.
- **Dense Reward Signal:** With a constant reward (typically 1 per timestep for balancing the pole), the environment provides frequent feedback. This continuous reward structure makes the learning process less sensitive to overestimation errors.
- **Stationarity:** The CartPole environment is stationary; its dynamics do not change over time, which further contributes to the stability of learning.

In summary, while DDQN is particularly designed to address challenges that arise in complex, stochastic, or sparse-reward environments, CartPole does not fully exhibit these challenging characteristics. Therefore, although DDQN still provides benefits over DQN in terms of stability and consistency, the differences between the two algorithms may be more pronounced in more complex environments.

3.8 How do you think DQN can be further improved? (This question is for your own analysis, but you may refer to external sources such as research papers and blog posts be sure to cite them properly.) [2-points]

DQN can be enhanced through several modifications that address its inherent limitations, such as overestimation bias, sample inefficiency, and high variance. Key improvements include:

- **Double Q-Learning:** As demonstrated by van Hasselt et al. [9], using Double DQN (DDQN) decouples the selection of actions from their evaluation. This modification reduces the overestimation bias seen in standard DQN, leading to more accurate Q-value estimates and improved training stability.
- **Dueling Network Architecture:** Wang et al. [10] proposed a dueling architecture that separately estimates the state-value function and the advantages of each action. This separation allows the network to more effectively distinguish between valuable and non-valuable states, especially in scenarios where many actions result in similar outcomes, thereby enhancing learning efficiency.
- **Prioritized Experience Replay:** Schaul et al. [11] introduced prioritized experience replay, where experiences are sampled based on their temporal-difference error rather than uniformly. This approach focuses learning on more informative transitions, improving convergence speed and sample efficiency.
- **Multi-Step Learning:** Incorporating multi-step returns allows the agent to consider cumulative rewards over multiple future steps instead of relying solely on one-step updates. This approach accelerates the propagation of reward signals and can lead to more robust policy updates.
- **Distributional Reinforcement Learning:** Instead of predicting a single expected return, distributional RL methods (e.g., Bellemare et al. [12]) estimate the full distribution of possible returns. This provides a richer understanding of uncertainty in reward estimates and can improve decision-making under variability.
- **Enhanced Exploration Strategies:** Beyond the basic ϵ-greedy approach, strategies such as parameter noise [14] or bootstrapped DQN introduce a more sophisticated exploration mechanism. These methods can help the agent escape local optima and explore the state space more effectively.
- **Integrating Multiple Improvements (Rainbow DQN):** The Rainbow DQN framework [13] combines many of the aforementioned improvementsDouble Q-Learning, Dueling Networks, Prioritized Replay, Multi-Step Learning, Distributional RL, and Noisy Networks for explorationresulting in a robust and state-of-the-art algorithm that addresses many of the pitfalls of the original DQN.

By integrating these enhancements, DQN can achieve faster convergence, improved stability, and higher overall performance, especially in more complex and stochastic environments.

Final Remarks

This notebook was run on Kaggle over a period of approximately 6 hours. You can view the notebook via the following link: HW2: DQN vs DDQN Notebook. Note that the notebook is currently private; it will be made public in the future.

It is important to note that during the development of this notebook, a critical bug was identified in

the DQN implementation: the DQN agent was erroneously using a target policy. As a result, the agent produced very poor results in some runs, and this issue was discovered only after extensive experimentation. I spent a significant amount of time debugging and troubleshooting this problem. For future assignments, please ensure that such errors are thoroughly checked to avoid similar delays.

References

[1] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2nd Edition, 2020. Available: http://incompleteideas.net/book/the-book-2nd.html.

[2] Gymnasium Documentation. Available: https://gymnasium.farama.org/

[3] Grokking Deep Reinforcement Learning. Available: https://www.manning.com/books/grokking-deep-reinforcement-learning

[4] Deep Reinforcement Learning with Double Q-learning. Available: https://arxiv.org/abs/1509.06461

[5] Cover image designed by freepik

[6] Hasselt, H. van, Guez, A., Silver, D. (2016). *Deep reinforcement learning with double Q-learning*. In Proceedings of the AAAI Conference on Artificial Intelligence.

[7] Wang, Z., Schaul, T., Hessel, M., Hasselt, H. van, Silver, D., De Freitas, N. (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. In Proceedings of the 33rd International Conference on Machine Learning (ICML).

[8] Schaul, T., Quan, J., Antonoglou, I., Silver, D. (2016). *Prioritized Experience Replay*. In Proceedings of the International Conference on Learning Representations (ICLR).

[9] Hasselt, H. van, Guez, A., & Silver, D. (2016). *Deep Reinforcement Learning with Double Q-Learning*. Proceedings of the AAAI Conference on Artificial Intelligence.

[10] Wang, Z., Schaul, T., Hessel, M., Hasselt, H. van, Silver, D., & De Freitas, N. (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. Proceedings of the 33rd International Conference on Machine Learning (ICML).

[11] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). *Prioritized Experience Replay*. International Conference on Learning Representations (ICLR).

[12] Bellemare, M.G., Dabney, W., & Munos, R. (2017). *A Distributional Perspective on Reinforcement Learning*. Proceedings of the 34th International Conference on Machine Learning (ICML).

[13] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Silver, D., & Sutton, R.S. (2018). *Rainbow: Combining Improvements in Deep Reinforcement Learning*. Proceedings of the AAAI Conference on Artificial Intelligence.

[14] Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., ... & Blundell, C. (2017). *Noisy Networks for Exploration*. International Conference on Learning Representations (ICLR).