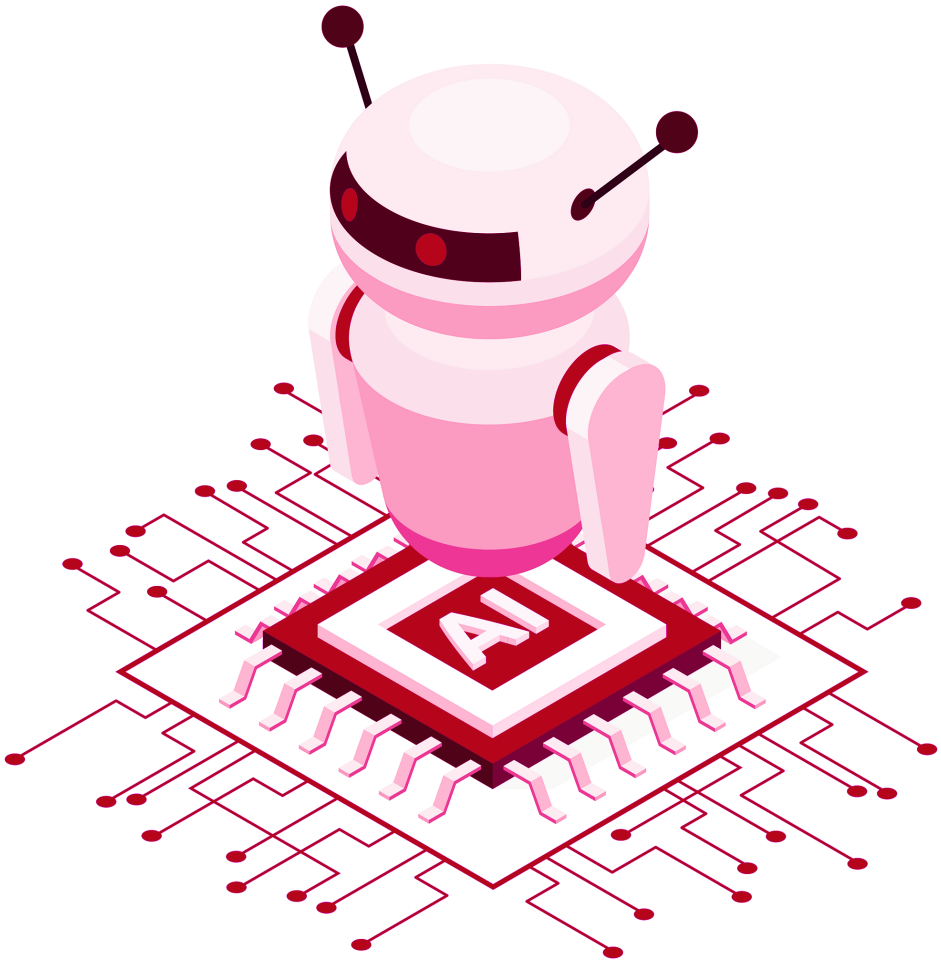

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 2:

Value-Based Methods

By:

Matin M.Babaei

400102114

RIML

Deep RL Course

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

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

At the beginning of learning, the agent explores more due to the ϵ-greedy policy with $\epsilon = 0.1$. This leads to:

1. **Higher Initial Regret:** The agent selects random actions 10% of the time, which may result in suboptimal moves. Since the agent has not yet learned an effective policy, it makes many mistakes early on, increasing regret.
2. **Regret Decreases Over Time:** As training progresses, the Q-values improve, leading to better action selection. Even though ϵ still allows exploration, most actions become more informed. Over time, the agent finds better policies, reducing the gap between its obtained reward and the optimal reward, thereby decreasing regret.

So

- High initial regret is due to random actions early in learning.
- Regret decreases as the agent learns better actions and starts exploiting more.
- A well-tuned ϵ decay can speed up learning by gradually shifting from exploration to exploitation.

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

Higher ϵ values, such as 0.5, mean the agent selects actions randomly more often, leading to "fluctuations in performance" as it explores different state-action pairs.

As the agent learns, Q-values are updated based on new experiences. If a significantly better path is discovered, the agent may suddenly shift its policy, leading to "performance jumps".

Since reinforcement learning updates values "over multiple steps", the impact of a good or bad decision may not be immediately visible. When enough updates accumulate, a sudden improvement (or drop) can occur.

- **Stochastic Environment Factors:** If the environment has stochastic transitions or rewards, the agent may sometimes experience unexpected variations in performance, causing jumps in the reward graph.

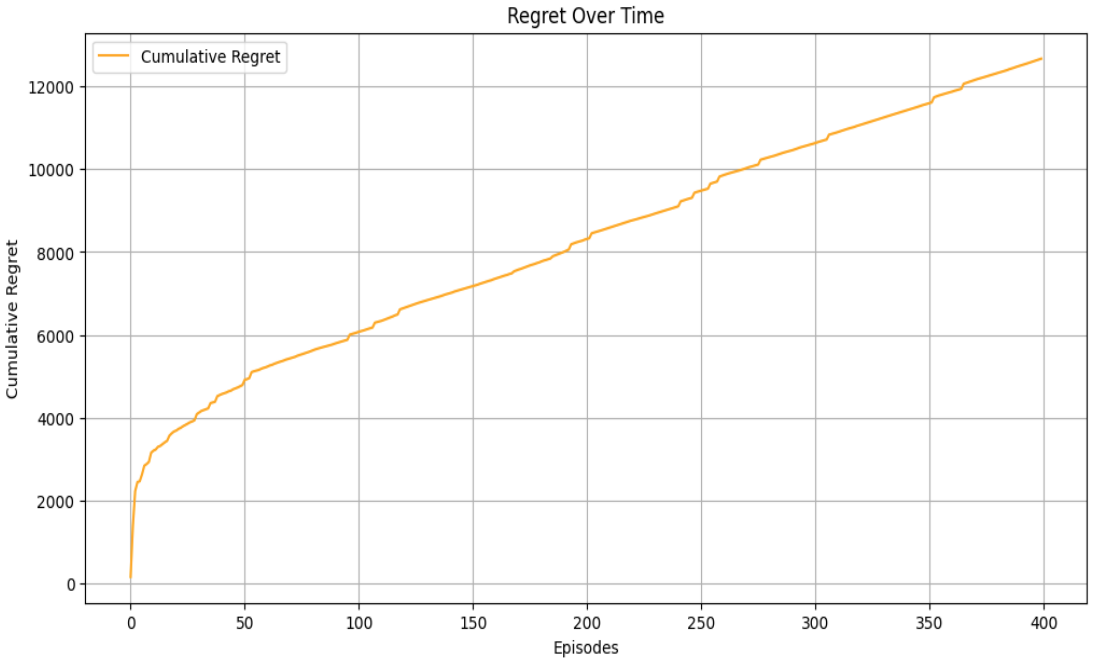

Figure 1: Epsilon = 0.1 Regret

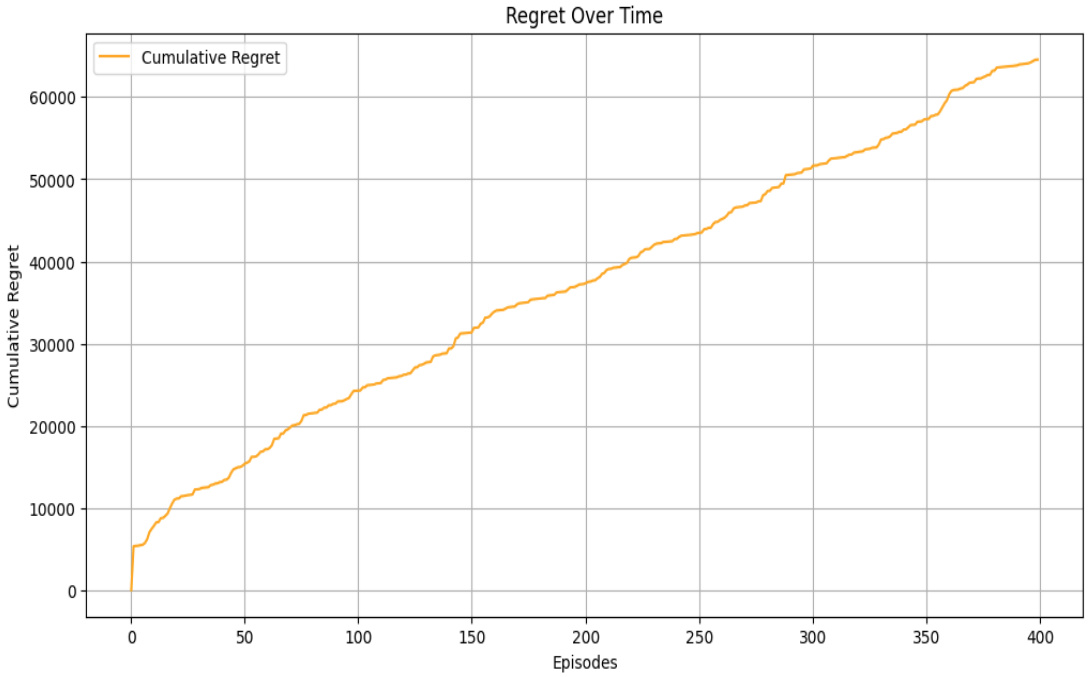

Figure 2: Epsilon = 0.5 Regret

Jumps are visible in the figures 1 and 2.

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

When $\epsilon = 0.9$, the agent explores randomly 90% of the time, leading to a more uniform and gradual learning process. The primary reasons for the linear change in performance are:

1. **High Exploration Rate:** With $\epsilon = 0.9$, the agent frequently selects random actions, reducing the chance of getting stuck in a local optimum. This results in a steady and gradual improvement rather than sharp jumps.
2. **Slower Convergence:** Since the agent explores most of the time, it takes longer to accumulate useful knowledge about optimal actions, leading to a linear increase in performance rather than rapid jumps.
3. **Consistent Policy Updates:** Due to high exploration, Q-values are updated more uniformly across different state-action pairs, resulting in a smooth, linear learning curve.
4. **Reduced Exploitation:** Unlike lower ϵ values, where the agent shifts to exploitation sooner, $\epsilon = 0.9$ ensures the agent continues exploring, preventing sudden jumps caused by policy shifts.

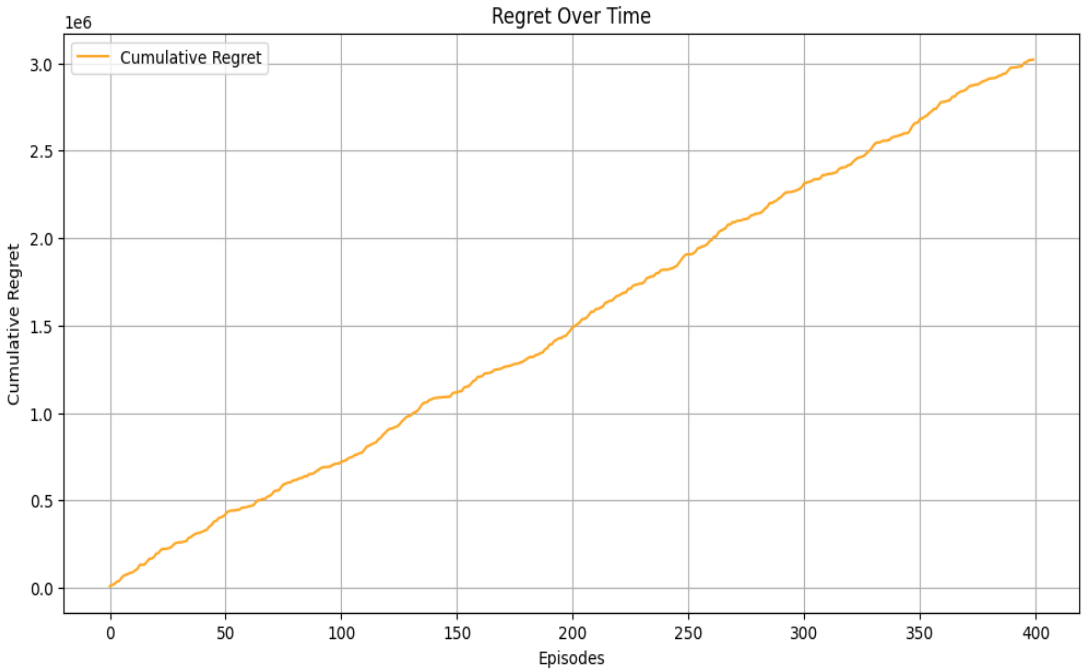

Figure 3: Epsilon = 0.9 Regret

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]

The policies learned with $\epsilon = 0.1$ and $\epsilon = 0.9$ differ significantly due to the balance between exploration and exploitation during training.

- **Epsilon = 0.1 (More Exploitation, Less Exploration):**
  - The agent selects the best-known action most of the time, exploring only 10% of the time.
  - The policy converges faster to a near-optimal solution.
  - The learned policy appears more structured and deterministic, as the agent follows a well-defined path based on learned Q-values.
  - However, if the initial exploration is poor, the agent may settle into a suboptimal policy.

- **Epsilon = 0.9 (More Exploration, Less Exploitation):**
  - The agent selects random actions 90% of the time, leading to a much broader exploration of the state space.
  - Learning is slower and more stochastic, with the policy evolving gradually.
  - The resulting policy may look more scattered or inconsistent early on, as the agent has not exploited optimal actions sufficiently.
  - Over time, the policy becomes more refined but still retains a degree of randomness due to continued exploration.

Key Differences:

- $\epsilon = 0.1$ leads to faster convergence but risks getting stuck in suboptimal policies.
- $\epsilon = 0.9$ results in wider exploration, which may improve learning but slows convergence.
- The policy for $\epsilon = 0.1$ appears more deterministic, while $\epsilon = 0.9$ produces a more varied and scattered policy.

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]

The row adjacent to the cliff (the lowest row) is particularly important in the Cliff Walking environment because it presents the highest risk of falling into the cliff, leading to significant negative rewards. The learned policies under different epsilon decay rates exhibit distinct behaviors in this region.

$Best\ Reward = 12 \times (-1) + 10(Goal) = -2$

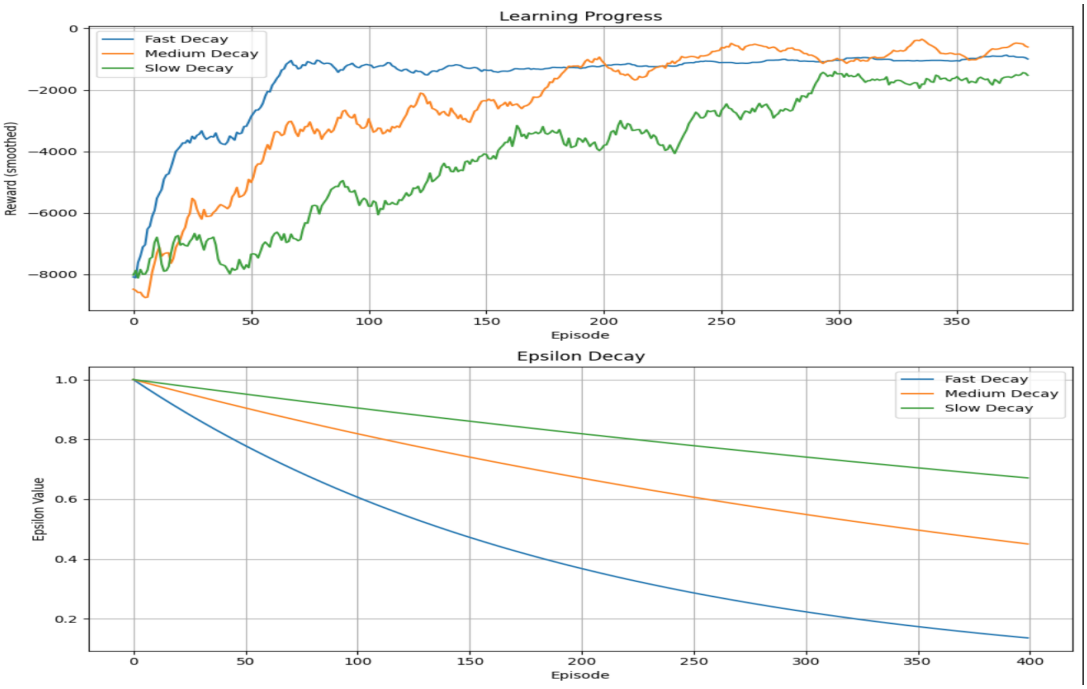

Figure 4: Epsilon Decay Comparison

- **Fast Decay Policy:**
  - Epsilon decreases rapidly, causing the agent to exploit learned knowledge early in training.
  - The agent finds a stable path to the goal faster but may settle into a suboptimal policy due to insufficient exploration.
  - The reward curve stabilizes quickly at a relatively high value.
- **Medium Decay Policy:**
  - The agent maintains exploration for a longer duration before converging to a stable policy.
  - This allows it to discover better routes, leading to higher long-term rewards compared to the fast decay policy.
  - The reward curve continues to improve gradually, surpassing the fast decay policy in later episodes.
- **Slow Decay Policy:**
  - The agent explores significantly longer, often taking suboptimal actions for an extended period.
  - Learning progress is slower, and the agent takes more time to converge to a high-reward policy.
  - Despite the slower improvement, the final policy may be more optimal, but at the cost of a prolonged learning phase.

Comparison of Learned Policies:

- Fast decay results in quick but potentially suboptimal policies.
- Medium decay provides a good balance, leading to higher long-term rewards.
- Slow decay enables thorough exploration, but learning is much slower.

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

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

Q-learning and SARSA are both reinforcement learning algorithms used for estimating the optimal action-value function (Q-values), but they differ in their update rules and learning strategies.

- **Q-learning (Off-policy)**
  - Q-learning is an **off-policy** algorithm, meaning it learns the optimal policy independently of the agent's current behavior.
  - The update rule uses the **maximum** Q-value of the next state, assuming a greedy policy in the future:
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$
  - Since Q-learning updates using the best possible future action, it tends to be more aggressive in finding the optimal policy.
  - It can learn optimal policies even if the agent follows a different exploration strategy during training.
- **SARSA (On-policy)**
  - SARSA is an **on-policy** algorithm, meaning it updates the Q-values using the action actually taken by the agent, following its current policy.
  - The update rule considers the next action a_{t+1} chosen by the current policy:
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$
  - Because SARSA follows the agent's policy, it accounts for exploration and is more conservative in learning.
  - It results in a smoother learning process and can be safer in high-risk environments, as it avoids overestimating rewards.

In this implementation, the important difference is in greedy vs epsilon greedy policy.

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

The value of n in n-step SARSA and n-step Q-learning determines how many steps into the future the algorithm considers when updating the Q-values. The choice of n significantly impacts each algorithm's learning efficiency and stability.

- **n-step SARSA:**
  - For **small** n (e.g., $n = 1$), the algorithm behaves similarly to standard SARSA, updating based on immediate rewards.

  - As n increases, the algorithm considers rewards over a longer sequence of actions, leading to smoother learning but requiring more memory and computation.
  - Too high an n can lead to instability, as rewards further in the future become less reliable.
- **n-step Q-learning:**
  - Small n values make the algorithm similar to standard Q-learning, relying on single-step lookahead.
  - Increasing n allows Q-learning to leverage more distant rewards, improving long-term planning but making updates noisier.
  - A high n can cause slower convergence since updates depend on a larger sequence of actions.

Comparison:

- **Low** n leads to more frequent updates but relies on short-term rewards.
- **High** n smooths out learning and incorporates long-term rewards but increases variance and memory requirements.
- The optimal n depends on the environment—too high can slow learning, too low can lead to shortsighted policies.
- In the result obtained, n-Sarsa has better rewards than n-Qlearning with any n.
- As n increases, the difference between Q values of START and TERMINAL points increases.

[Results were 6 images so I decided not to bring them here. You cen refer to the notebook.]

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 is crucial, and neither a high nor low n is always better. Each has its own advantages and disadvantages.

- **Low** n **(e.g.,** $n = 1$ **or** $n = 2$**)**
  - **Advantages:**
    * More frequent updates, leading to faster feedback and learning.
    * Less variance, making updates more stable.
    * Requires less memory and computation.
  - **Disadvantages:**
    * Focuses too much on immediate rewards, potentially leading to suboptimal policies.
    * Can struggle with long-term credit assignment.
- **High** n **(e.g.,** $n = 10$ **or more)**
  - **Advantages:**

* Incorporates long-term rewards, improving strategic decision-making.
* Reduces the variance in updates, leading to smoother learning.

– **Disadvantages:**

* Requires storing and processing more transitions, increasing memory and computational cost.
* Higher variance in rewards can slow convergence and introduce instability.
* If n is too large, it may delay necessary updates, making learning inefficient.

The best choice of n depends on the environment and the trade-off between stability and long-term planning. A moderate n often balances learning speed and accuracy effectively. For example in this case $n = 2$ looks optimal.(figure 5)

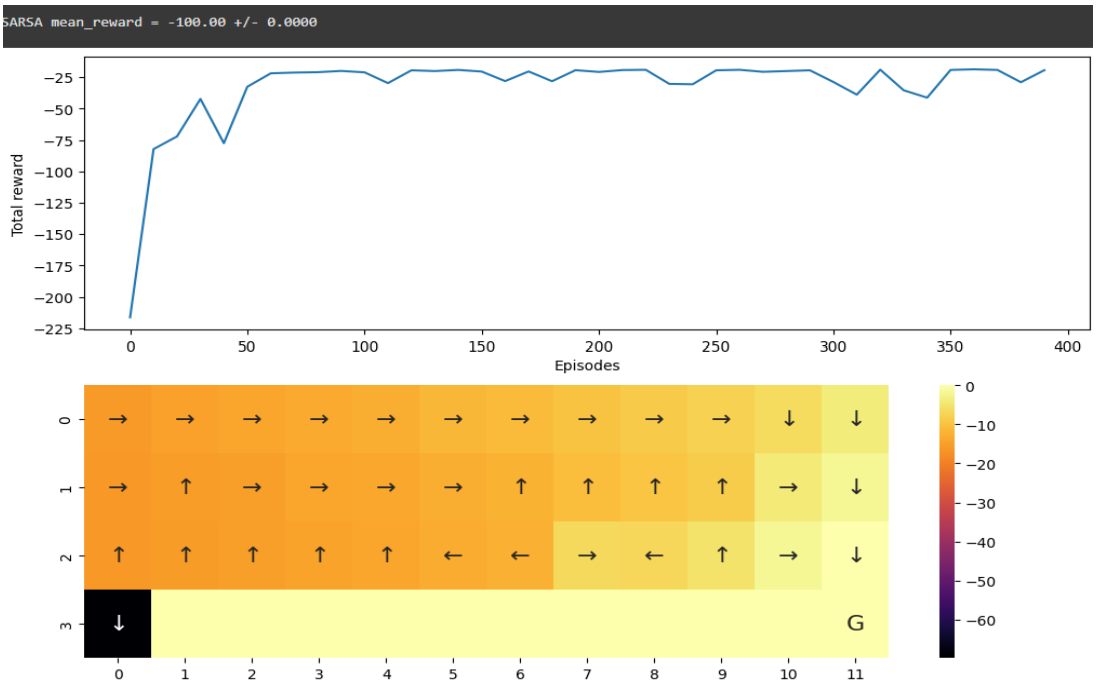

Figure 5: Sarsa n=2 Rewards in episodes

3 DQN vs. DDQN

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

DDQN generally performs better than DQN. The reason is that DQN suffers from overestimation bias when updating the Q-values. In DQN, the same network is used to both select and evaluate actions, which can lead to the algorithm overestimating the Q-values. On the other hand, DDQN mitigates this issue by decoupling the action selection and evaluation processes. DDQN uses one network to select actions and another (target) network to evaluate them, which reduces overestimation bias and leads to more stable and accurate Q-value updates. As a result, DDQN often achieves better performance and faster convergence in various environments.

As you can see in figure 6, DDQN (blue) achieves higher mean rewards than DQN (red) and appears to stabilize at a higher performance level. This suggests that DDQN learns better policies, likely due to its reduced overestimation bias, which helps it make more accurate action-value estimates.

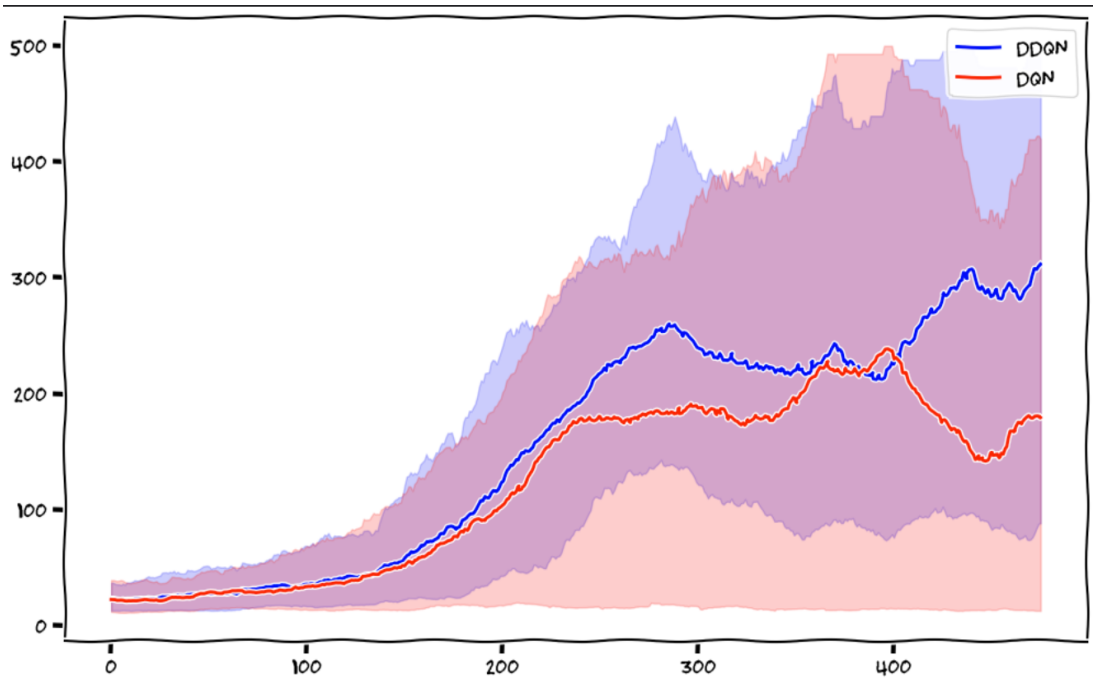

Figure 6: DQN vs. DDQN Reward Comparison

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

DDQN typically has a tighter upper and lower bound for rewards compared to DQN. This is because DDQN reduces the overestimation bias present in DQN. The Q-values in DQN can often become overestimated, which may cause an exaggerated reward signal, whereas DDQN's decoupling of the action selection and evaluation helps maintain more realistic Q-values, resulting in less fluctuation and a tighter range of possible rewards.

As you can see in figure 6, DDQN has a narrower shaded region compared to DQN, meaning its upper and lower bounds are tighter. This suggests that DDQN has more consistent performance across different episodes.

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

Yes, DDQN exhibits greater stability in learning. The reason is that by reducing the overestimation bias, DDQN prevents the Q-values from drifting too far away from their true values, which can happen in DQN. Overestimations can destabilize the training process and slow down convergence. By decoupling the action selection from evaluation, DDQN ensures that the learning process is more stable and consistent, leading to better overall performance.

And intuitively, since DDQN has a tighter reward distribution and its learning curve appears smoother, it indicates more stable learning. This is because DDQN reduces overestimation bias, which prevents drastic fluctuations in Q-values and leads to more stable policy improvements.

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

Some general issues with DQN include:

1. Overestimation Bias: DQN tends to overestimate Q-values, as it uses the same network for both action selection and evaluation. This can lead to poor decisions, especially in environments with noisy rewards.
2. Instability in Training: DQN can exhibit high variance in learning and is sensitive to hyperparameters such as the learning rate and discount factor.
3. Sample Efficiency: DQN may require a large number of samples to converge, especially in complex environments.
4. High Memory Usage: DQN needs to store a large replay buffer for experience replay, which can become inefficient in environments with very large state spaces.
5. Sensitivity to hyperparameters.

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 approaches can mitigate DQN's issues:[Github]

1. Double Q-learning (DDQN): As mentioned, using DDQN can reduce the overestimation bias by decoupling action selection from evaluation.
2. Prioritized Experience Replay: Instead of sampling random experiences, prioritize transitions that are expected to have the most impact on learning. This can help improve sample efficiency.
3. Dueling Network Architecture: This architecture separates the value and advantage streams of the Q-value function, which helps in environments where the agent can take similar actions but with different values, improving the stability and convergence rate.
4. Double DQN with Noisy Networks: Adding noise to the network can encourage exploration while reducing overfitting, thus improving stability and convergence.

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 figure 6 shows that DDQN achieves higher and more stable rewards, confirming that it reduces overestimation bias and stabilizes learning compared to DQN.

In fact, the DDQN curve demonstrated smoother learning with fewer fluctuations in reward compared to DQN. While DQN may show sharper increases and decreases in rewards due to overestimation, DDQN's more stable learning curve reflects its ability to produce more consistent and reliable Q-values.

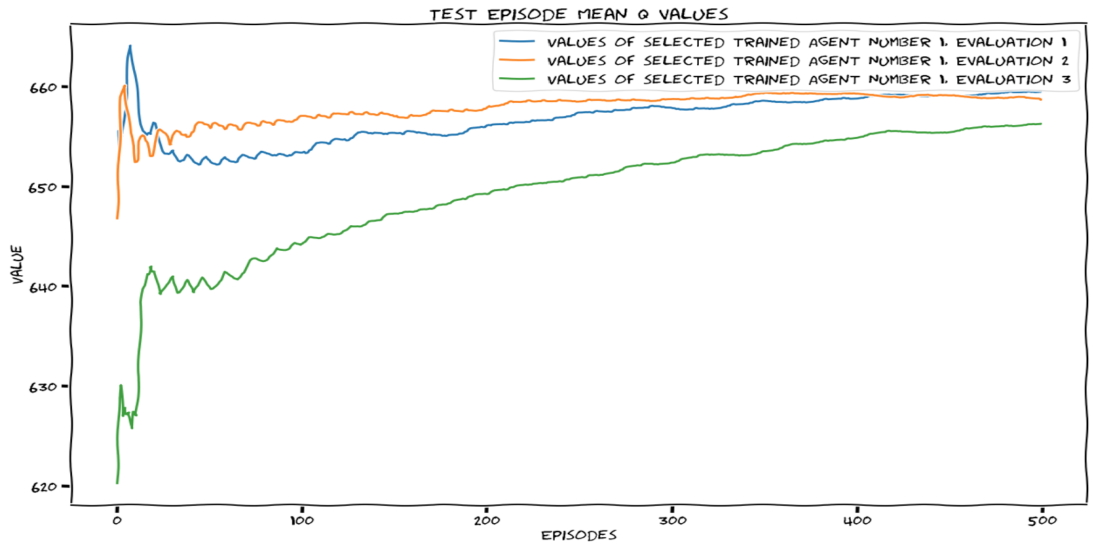

Figure 7: DQN Q values

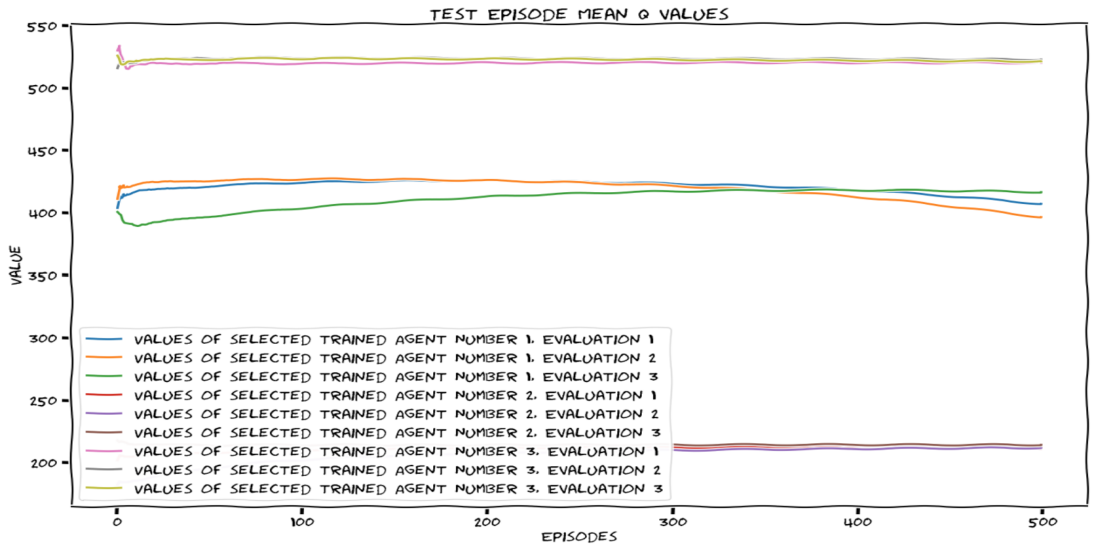

Figure 8: DDQN Q values

Comparing figure 7 and figure 8, we can infer that DDQN overestimates Q values less then DQN.

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]

Environments with complex dynamics and high reward variance can expose the weaknesses of DDQN. For example, environments with many possible actions and long-term dependencies in rewards require more precise control over Q-value estimates to avoid divergence. In contrast, simpler environments like CartPole exhibit lower complexity and have less variability in rewards, which makes them less prone to the issues that DDQN addresses. CartPole is relatively simple in terms of dynamics and typically has more deterministic outcomes, so DDQN's advantages, such as mitigating overestimation bias, may not be as pronounced in this case. Therefore, CartPole does not fully exhibit the characteristics of more complex environments where DDQN's performance improvements are more noticeable. CartPole is relatively simple, with deterministic dynamics and dense rewards, so the impact of DDQN is present but less significant than in more complex tasks like **Atari games**.

Note:

- Complexity: Environments with sparse rewards or large state spaces benefit more from DDQN.
- Dynamics: High variability in transition dynamics makes DDQN more useful due to reduced overestimation bias.

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 further improved by:

Using More Advanced Exploration Strategies: Techniques like Noisy Networks or Bootstrapped DQN [7] can improve exploration by introducing randomness into the network's weights, which encourages the agent to explore more effectively.

Model-based Reinforcement Learning: Integrating model-based approaches, where the agent learns a model of the environment to predict future states, could improve sample efficiency.[6]

Learning Rate Scheduling: Adaptive learning rate schedules, such as Adam or RMSProp, could stabilize training by adjusting the learning rate based on the gradients.

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] Model-based rL

[7] Bootstrapped DQN