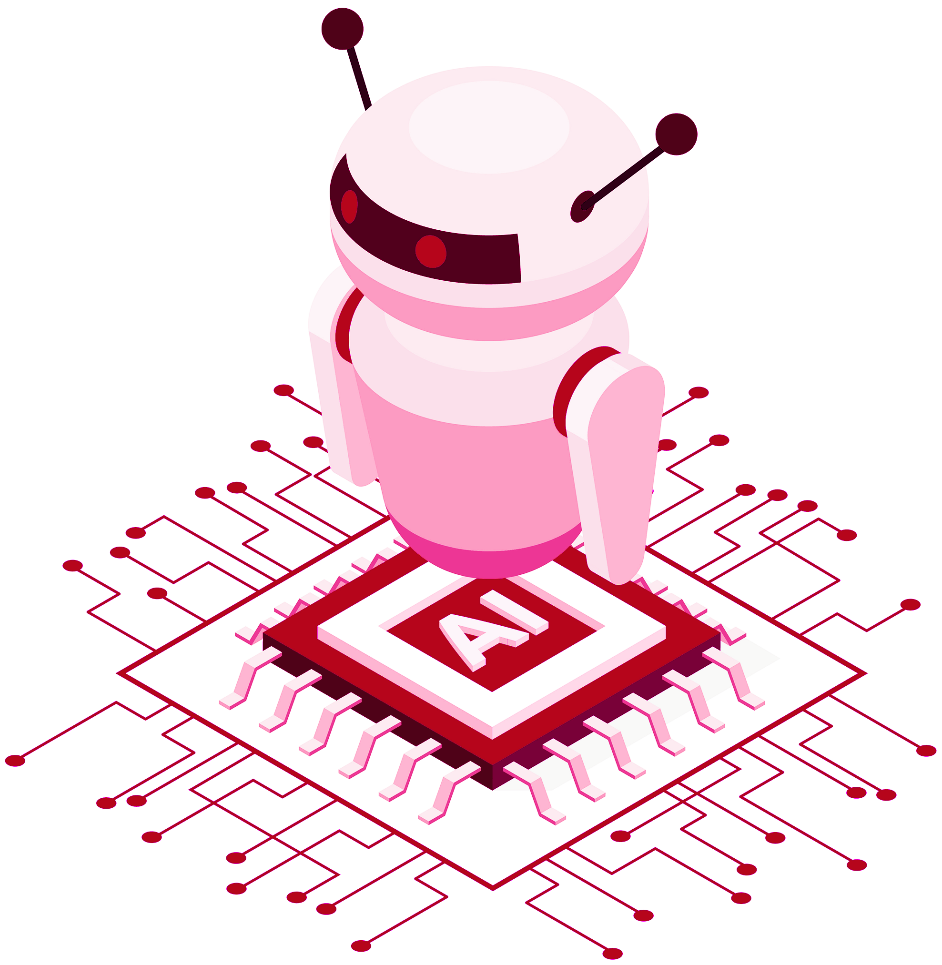

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 2:

Value-Based Methods

By:

Ali Najar

401102701

Spring 2025 w

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]

Early on, the agent explores a lot because it doesn't know which actions are good, so it makes a bunch of mistakes, leading to high regret. But since $\varepsilon = 0.1$, it's mostly exploiting the best-known actions pretty quickly. As it learns, it starts making better choices, and the regret grows more slowly. It never stops increasing because regret is cumulative, but the fact that it flattens out a bit means the agent is getting smarter and making fewer bad moves.

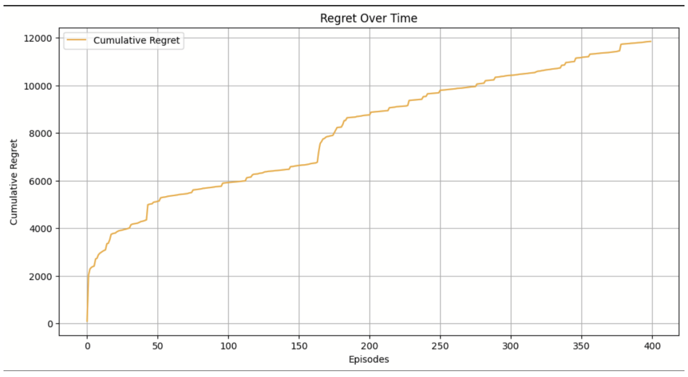

Figure 1: SARSA with $\varepsilon = 0.1$

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

The jumps in cumulative regret for both $\varepsilon = 0.1$ and $\varepsilon = 0.5$ happen because of sudden shifts in the agent's policy, typically due to exploration. (Although note that my learning rate is 0.05 so the changes in policy are quite smooth and (2) does not have any jumps. (Since the randomness is a lot, it cannot get any better after some point so the slope of the cumulative regret is high. But the low learning rate has removed jumps from epsilon 0.5)

However, (1) shows some jumps since epsilon is small and the agent's policy actually gets better since the epsilon is not high to cap its capabilities.

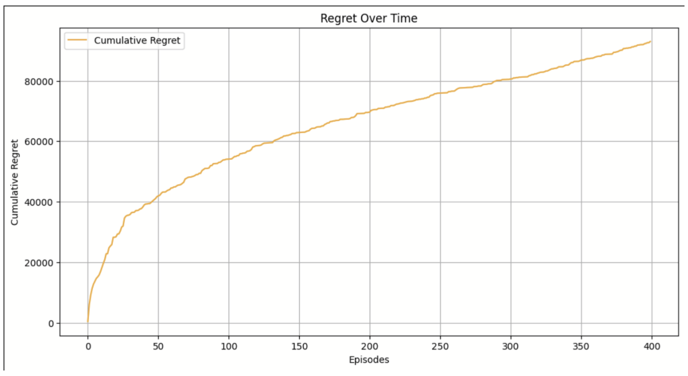

Figure 2: SARSA with $\varepsilon = 0.5$

If we consider the jumps at the beginning both epsilons shows jumps since the agents policy gets better after some time and the regret rate drops rapidly and stays constant after some time.

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

With $\varepsilon = 0.9$, the agent is exploring 90% of the time, meaning it's mostly taking random actions rather than following a learned policy. This results in almost uniformly bad decisions, leading to a steady and linear increase in regret over time.

Since randomness factor is too high the agent never settles into a good strategy. this leads the policy to be almost the same and random all the time leading to a huge constant regret all the time (i.e. making the slope constant so the cumulative regret is linear.). The agent is essentially stuck in exploration mode and never properly transitions to an optimal policy.

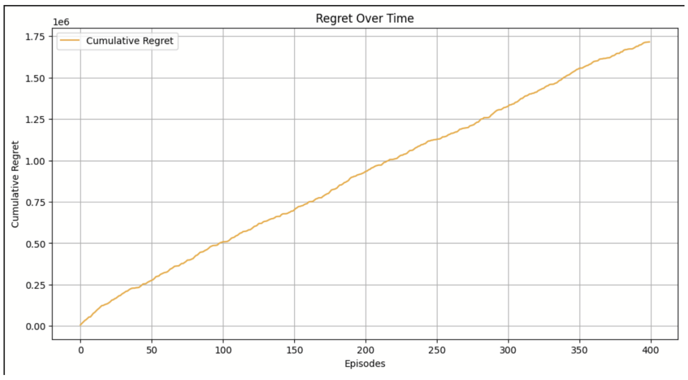

Figure 3: SARSA with $\varepsilon = 0.9$

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]

- $\varepsilon = 0.1$:
  - The agent follows the best-known policy most of the time (90% of the time) and only explores occasionally (10% of the time).
  - Over time, the regret curve flattens because the agent makes fewer mistakes as it refines its policy.
  - The agent initially explores (leading to some regret), but as it learns the best actions, it mostly exploits them, reducing further mistakes.
- $\varepsilon = 0.9$:
  - The agent is random most of the time (90%) and rarely exploits what it has learned.
  - Since mistakes happen at a nearly constant rate, the regret grows linearly, indicating no clear improvement.
  - This prevents it from forming a stable policy, making its decisions highly inconsistent.
  - Since the agent is taking random actions most of the time, there's no noticeable improvement in decision-making.

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]

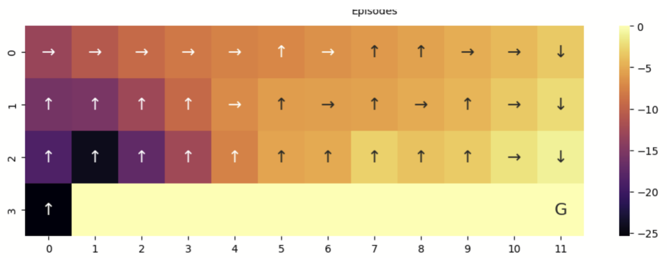

Figure 4: Fast Decay

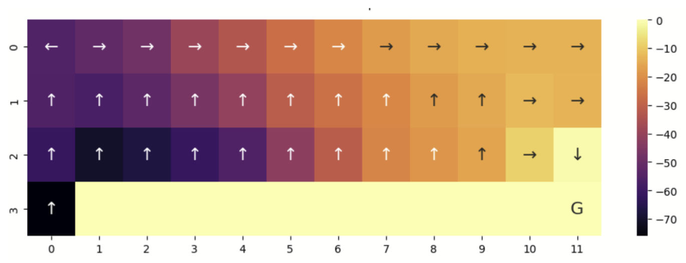

Figure 5: Medium Decay

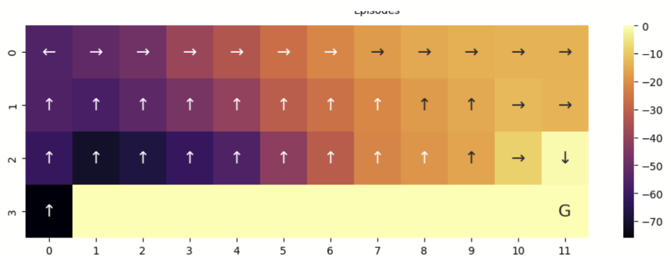

Figure 6: Slow Decay

Optimal Policy Evaluation:

It can be seen from (4), (6) that the agent move with caution near the cliff and tries to keep his distance from the cliff. Most arrows point upward (↑), this suggests that the policy avoids moving toward the cliff (bottom row) and instead prefers to go up to a safer row.

Comparing Learned Policies:

For fast decay the agent takes less caution in the second row and sometimes prefers to move right. this is because the agent's epsilon gets small rapidly and the agent's exploration and random actions become less, so the agent prefers to move forward. But for slow decay, the agent's epsilon decays slowly and the exploration rate is almost high during most of the times so the agent takes caution even in the second row and moves upward (↑) away from the cliff. The medium decay rate's strategy lies somewhere between slow decay and fast decay. Note that the results rely a lot on hyperparameters. Since we have a low learning rate, it affects the policies learned by agents.

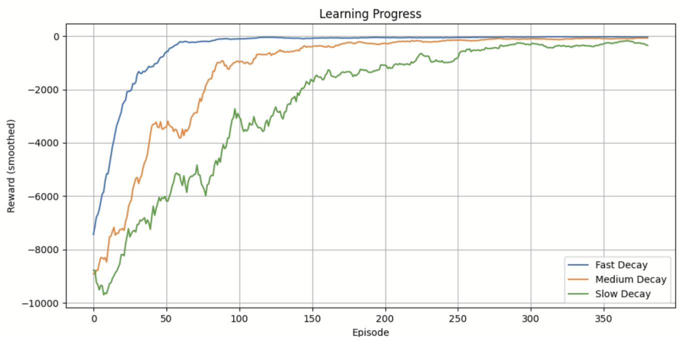

Figure 7: Different decay rates rewards

Comparing Rewards:

It can be seen that with a proper learning rate all of the plots, converge to the optimal policy. But it is evident that the agent with fast decay rate, converges faster and smoother to the optimal policy. This is because the agent in medium and slow decay, explores more and acts more randomly at initial steps.

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:

It is an off-policy algorithm. It updates its Q-values using the maximum estimated Q-value from the next state (i.e., it uses $\max_{a'} Q(s', a')$), regardless of the action actually taken by the agent. This means it learns the value of the optimal policy even if the agent is following a different (exploratory) behavior.

SARSA:

It is an on-policy algorithm. It updates its Q-values based on the actual action taken by the agent in the next state (i.e., it uses $Q(s', a')$ where a' is the next action selected by the current policy). This makes SARSA more reflective of the agent's actual behavior, including its exploration strategy.

In summary, Q-learning targets the optimal policy directly, while SARSA learns the value of the policy being executed.

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

Q-learning:

- $n = 1$: It can be seen that in this case (8) that the algorithm updates its estimates with limited lookahead (one-step TD error), which is typically more stable but slower in propagating rewards from distant states. So this leads to slower convergence rate.
- $n = 2$: As in figure (9), it is evident that the convergence of the reward per episode is more smooth. this is because the agents considers more future steps for each update. The update uses more actual future rewards, potentially speeding up learning by propagating reward information faster.

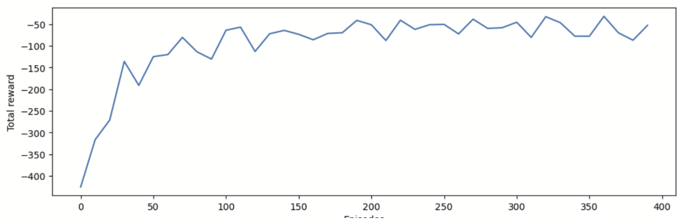

Figure 8: 1 step Q-Learning

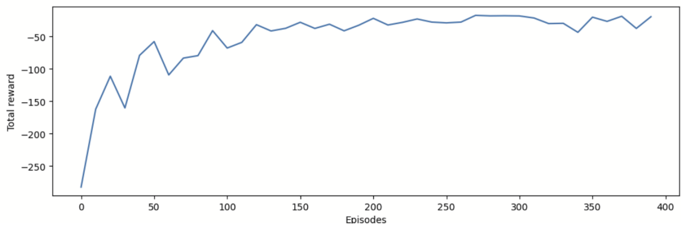

Figure 9: 2 step Q-Learning

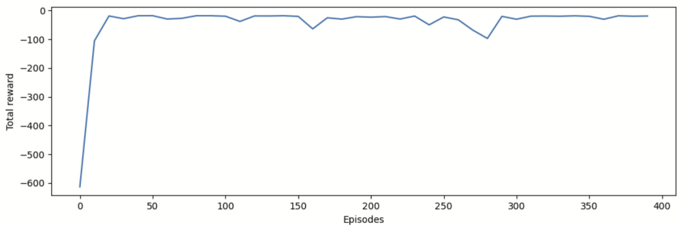

Figure 10: 5 step Q-Learning

- $n = 5$: As in figure (10), it can be seen that in this case we rely more on actual rewards rather than bootstrapping which can lead to better learning speed. The agent converges to the optimal policy early.

  Overall, The changes in reward when we use lower steps is much smaller. This is because the intensity of updates is lower when we use bootstrapping (i.e. lower step size.). But when we use higher step sizes, this leads to more sharp updates, leading to early convergence to the optimal policy.

SARSA:

- $n = 1$: Figure (11) shows a similar behavior to figure (8) Similar to one-step Q-learning, low n in SARSA means the updates are based mostly on immediate rewards and the immediate next action, which keeps variance of updates low. So the convergence to optimal policy is slow.

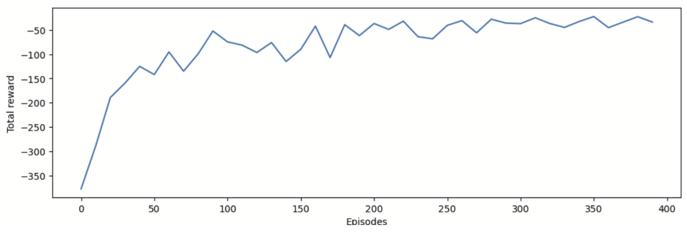

Figure 11: 1 step SARSA

- $n = 2$: It can be seen that the convergence is more smooth in this case. This case is also similar to 2-step Q-learning showing the effect of using actual rewards for more steps.

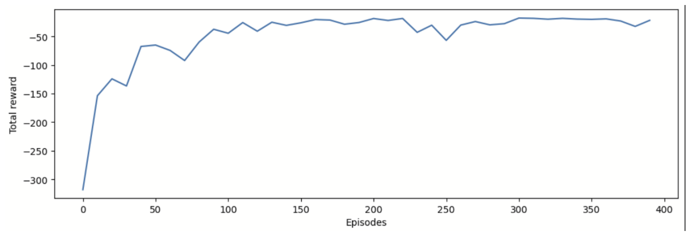

Figure 12: 2 step SARSA

- $n = 5$: The updates here are more rapid than the two other cases. As it can be seen a little exploration led to a major drop of reward in the middle of the plot. This is mainly due to the sharp updates, which is the result of using more steps of actual rewards. But overall in this case the convergence rate is much higher like 5-step Q-learning.

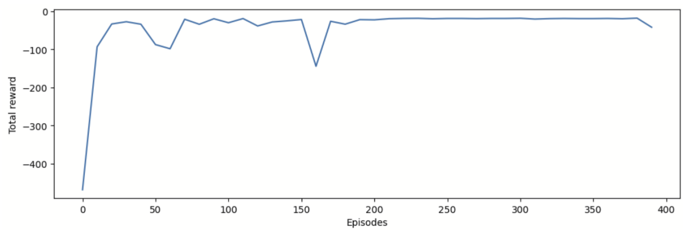

Figure 13: 5 step SARSA

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]

Neither a high n nor a low n is universally better. it's a trade-off between bias and variance of updates, as well as between learning speed and stability.

Low n values offer the advantage of stable and low-variance updates. Because they only uses immediate rewards and a short lookahead, these methods rely heavily on bootstrapping from nearby states. This results in smoother, less noisy updates and faster computation per update, which is particularly beneficial in environments with frequent, consistent rewards. However, the downside is that the heavy reliance on bootstrapped estimates can introduce higher bias, meaning that the learned values might deviate from the true expected returns. Additionally, in environments where rewards are delayed, low n values may slow down the propagation of reward information, delaying learning improvements in earlier states.

High n values, on the other hand, reduce bias by incorporating a longer sequence of actual rewards before bootstrapping, providing a more accurate estimation of the true return. This can be especially advantageous in environments with sparse or delayed rewards, as it accelerates the propagation of reward signals across the state space. The trade-off, however, is an increase in variance; longer rollouts tend to accumulate noise from stochastic transitions, which can lead to more volatile and less stable learning. Additionally, using a high n increases computational demands and makes the algorithm more sensitive to fluctuations in the policy.

3 DQN vs. DDQN

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

DDQN performs much better than DQN as shown in (14). This is mainly because DDQN offers a better estimate of true values, whereas DQN overestimates the values leading to worse performance. This leads to better policy evaluation.

DQN uses a single neural network to approximate the action-value function (Q-values). When updating Q-values, it selects the action with the highest estimated value using a max operator. This approach can lead to overoptimistic estimates, particularly when the Q-function is approximated with noise or error.

DDQN mitigates this issue by decoupling the action selection and action evaluation. It uses two separate networks (or a dual role of networks) so that one network selects the action (using the max operator) and the other evaluates the value of that action. This separation reduces the overestimation bias, leading to more accurate and stable value estimates.

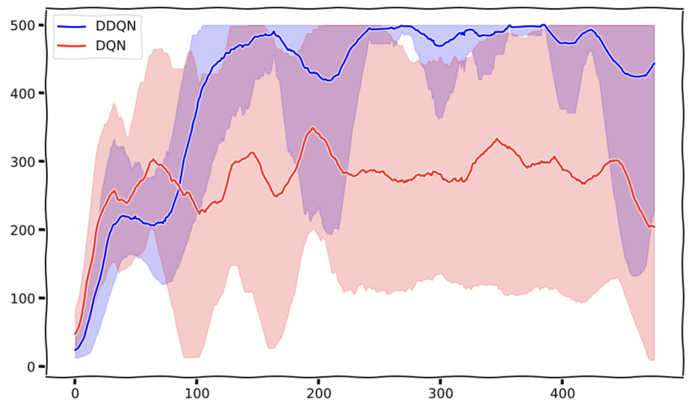

Figure 14: DDQN and DQN mean reward per episode

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

DDQN has a tighter upper and lower bound as in figure (14). This is mainly due to the fact that DDQN does not overestimates values, which leads to a more consistent value evaluation.

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 DDQN tends to exhibit greater stability in learning. The tighter reward bounds indicate that the network's estimates are not excessively overoptimistic, which in turn reduces large fluctuations in the Q-value updates. This controlled estimation leads to smoother and more predictable learning dynamics, ultimately resulting in a more stable training process.

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

One of the main issues of DQN is the overestimation of values. The use of a max operator on noisy Q-value estimates can lead to overly high value predictions, causing instability. With a single network handling both action selection and evaluation, the training updates can become instable, leading to divergence in some cases. It is also sample inefficient. DQN often requires a very large number of samples to learn effectively, partly because the experience replay does not prioritize more informative experiences.

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]

DDQN

The first method that migitates some issues is DDQN [4]. By using separate networks for selecting and evaluating actions, DDQN effectively reduces overestimation bias, leading to more reliable value estimates. (15) shows the comparison of DQN and DDQN in some atari games and how DQN overestimates the Q-values.

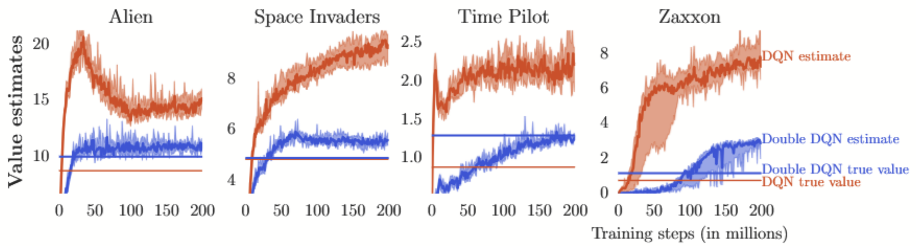

Figure 15: Overestimation of DQN

Maxmin Q-learning

Maxmin Q-learning [7] is an improvement over standard Q-learning designed to mitigate overestimation bias in Deep Q-Networks (DQN).

Maxmin Q-learning proposes a simple but effective solution: instead of using a single Q-network, it maintains multiple Q-networks (like ensemble methods) and updates Q-values using the minimum of the Q-values from a randomly selected subset.

The reason this works is that the max operator in standard Q-learning tends to overestimate Q-values, while the min operator counteracts this by favoring lower estimates. By using multiple Q-networks, variance is reduced, leading to more stable and accurate learning. Also, unlike Double DQN, which only addresses overestimation in action selection, Maxmin Q-learning directly modifies the update step to control bias.

Despite these advantages, Maxmin Q-learning has some cons too. One of them is increased computational cost due to maintaining multiple Q-networks and risk of underestimation if the minimum operator is too aggressive.

Algorithm 1 Maxmin Q-learning

```
Require: Step-size $\alpha$, exploration parameter $\epsilon > 0$, number of action-value functions $N$
 1: Initialize $N$ action-value functions $\{Q^1, Q^2, \ldots, Q^N\}$ randomly
 2: Initialize empty replay buffer $\mathcal{D}$
 3: Observe initial state $s$
 4: while Agent is interacting with the environment do
 5:     $Q^{\min}(s,a) \leftarrow \min_{k\in\{1,\ldots,N\}} Q^k(s,a), \quad \forall a \in \mathcal{A}$
 6:     Choose action $a$ by $\epsilon$-greedy w.r.t. $Q^{\min}(s,a)$
 7:     Execute $a$, observe reward $r$ and next state $s'$
 8:     Store transition $(s,a,r,s')$ in $\mathcal{D}$
 9:     Select a subset $S \subseteq \{1,\ldots,N\}$ (e.g., randomly select one $i$ to update)
10:     for $i \in S$ do
11:         Sample random mini-batch of transitions $(s_{\mathcal{D}}, a_{\mathcal{D}}, r_{\mathcal{D}}, s'_{\mathcal{D}})$ from $\mathcal{D}$
12:         Get target update: $y^{\text{MQ}} \leftarrow r_{\mathcal{D}} + \gamma \max_{a'} Q^{\min}(s'_{\mathcal{D}}, a')$
13:         Update action-value $Q^i$: $Q^i(s_{\mathcal{D}}, a_{\mathcal{D}}) \leftarrow Q^i(s_{\mathcal{D}}, a_{\mathcal{D}}) + \alpha\Big[y^{\text{MQ}} - Q^i(s_{\mathcal{D}}, a_{\mathcal{D}})\Big]$
14:     end for
15:     $s \leftarrow s'$
16: end while
```

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

From figures (16) and (17) it can be noticed that that values estimated by DQN reaches almost 128 at some stages and remains above 124. But DDQN's estimations are around 120. From the fact that DDQN performs much better than DQN, we can derive that DQN is overestimating the Q-values. So the main purpose of DDQN which is reducing overestimations is vividly clear from the plots.

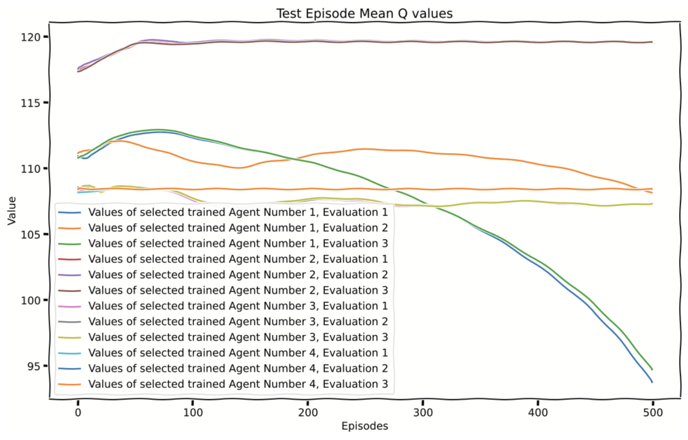

Figure 16: DDQN estimated values

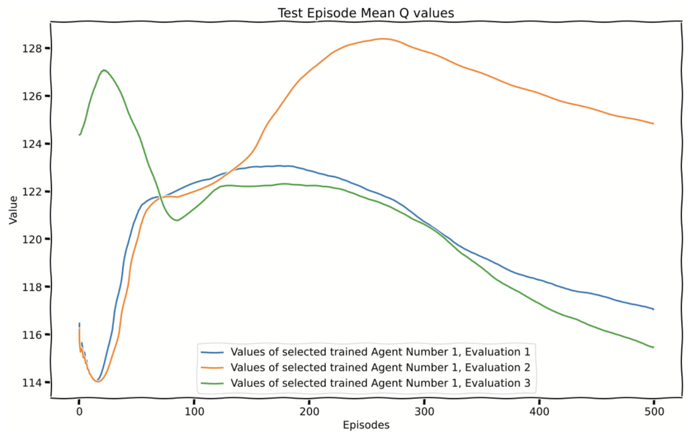

Figure 17: DQN estimated values

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]

Complexity of State and Action Spaces

Environments like Atari 2600 games present high-dimensional inputs (raw pixels of dimension $84 \times 84 \times 4$). Large or complex state spaces increase approximation errors in Q-values. The more complex the state space, the more beneficial DDQN becomes compared to DQN, because it mitigates overestimation errors that compound when using standard DQN. But there are still some minor issues. For example overestimation bias is more pronounced when there are many actions to choose from. Many Atari games have multiple actions (e.g., up to 18 joystick/button combinations). Overestimation bias is more pronounced when there are many actions to choose from. This effect is particularly pronounced when the estimation errors differ across actions, leading to systematic overestimation bias.

Reward Structure and Episode Length

Sparse reward tasks (e.g., Montezuma's Revenge) make it harder to learn accurate Q-values; overestimation errors can remain uncorrected for many steps. On the other hand, dense reward tasks (e.g., Pong) provide frequent feedback, so overestimations are less detrimental. So Overall sparse reward tasks is one challenge that DDQN faces, even though it handles them better than DQN.

Determinism vs. Stochasticity

In purely deterministic settings (many Atari games are mostly deterministic), DQN's overestimation can still appear when the Q-network's approximations are inaccurate. Small errors in value estimation can propagate through bootstrapping, amplifying the overestimations, especially when the environment's behavior is non-linear or partially observable.

All these factors have influence on the performance of DDQN. They can affect its sample efficiency and value estimation. But even in these scenarios, DDQN manages to perform with outstanding results, reaching high rewards in most games.

CartPole

This environment is much simpler than Atari games. It does not exhibit the characteristics of these games.

- Small State Space: CartPole's state is just 4 continuous variables (cart position, velocity, pole angle, angular velocity). This is drastically smaller than the high-dimensional pixel input in Atari. This leads to less function-approximation error and DDQN's performs perfect without any issues.
- Very Few Actions: CartPole has only 2 discrete actions (push left or push right). With fewer actions, the overestimation bias from the $\max$ operator is minimal.
- Dense Reward & Short Episodes: CartPole typically gives a reward of $+1$ per time step for balancing the pole and episodes end quickly (once the pole tips or the cart goes out of bounds). This means that any overestimated Q-values do not accumulate as severely, and learning is relatively straightforward.

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]

Prioritized experience replay (PER)

Another method is to change sampling mechanism as mentioned in [6]. Instead of sampling uniformly from the replay buffer, this method gives priority to experiences that have a higher learning potential, improving sample efficiency and speeding up convergence. PER assigns a priority to each experience based on its temporal-difference (TD) error, ensuring that transitions with larger TD errors—which indicate more surprising or informative experiences—are replayed more often. Each transition i is assigned a priority based on its TD error:

$$p_i = |\delta_i| + \epsilon,$$

where ϵ is a small constant to avoid zero priority.

The TD error for transition i is computed as:

$$\delta_i = r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-) - Q(s_i, a_i; \theta),$$

The probability of sampling transition i is given by:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where α determines the degree of prioritization (with $\alpha = 0$ corresponding to uniform sampling).

To correct for the bias introduced by non-uniform sampling, importance sampling weights are computed as:

$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta,$$

where N is the total number of transitions in the replay buffer and β adjusts the amount of bias correction (often annealed during training). These weights are usually normalized by dividing by the maximum weight in a minibatch.

The loss function for the Q-network, incorporating the IS weights, is:

$$L(\theta) = \mathbb{E}_{i \sim P(\cdot)} \left[w_i \left(Q(s_i, a_i; \theta) - y_i \right)^2 \right],$$

where y_i is the target Q-value for transition i.

The algorithm mentioned in paper [6] is as follows:

Algorithm 2 Double DQN with Proportional Prioritization

```
Require: Minibatch size $k$, step size $\eta$, replay period $K$, memory size $N$, exponents $\alpha, \beta$, budget $T$
 1: Initialize replay memory $\mathcal{H} \leftarrow \emptyset$, $\Delta \leftarrow 0$
 2: Initialize policy network parameters $\theta$, copy to target network $\theta^-$
 3: for $t \leftarrow 1$ to $T$ do
 4:     Observe $S_t, R_t, \gamma_t$
 5:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with priority $p \leftarrow \max_{i<t} p_i$
 6:     if $t \bmod K = 0$ then
 7:         for $j = 1$ to $k$ do
 8:             Sample transition $j$ with probability: $P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$
 9:             Compute importance sampling weight: $w_j = \left(\frac{1}{N \cdot P(j)}\right)^\beta / \max_i w_i$
10:             Compute TD-error: $\delta_j = R_j + \gamma_j\, Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
11:             Update priority: $p_j \leftarrow |\delta_j| + \varepsilon$
12:             Accumulate weighted gradient: $\Delta \leftarrow \Delta + w_j\, \delta_j\, \nabla_\theta Q(S_{j-1}, A_{j-1})$
13:         end for
14:         if $t \bmod k = 0$ then
15:             Update weights: $\theta \leftarrow \theta + \eta\, \Delta, \quad \Delta \leftarrow 0$
16:             From time to time, copy weights to target network $\theta_{\text{target}} \leftarrow \theta$
17:         end if
18:     end if
19:     Choose action $A_t \sim \pi_\theta(S_t)$
20: end for
```

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] Prioritized Experience Replay. Available: https://arxiv.org/abs/1511.05952

[7] Maxmin Q-learning: Controlling the Estimation Bias of Q-learning. Available: https://arxiv.org/abs/2002.06487