ASSIGNMENT 1: TABULAR REINFORCEMENT LEARNING

Student name	**Ritesh Sharma**
Student number	S4158059

1 INTRODUCTION

This assignment introduced the fundamental principles of tabular value-based reinforcement learning. Firstly, we implemented Dynamic programming (DP). In DP, we were provided with access to the full environment model with which we were able to find the optimal solution. Then we explored model-free reinforcement learning in which the access of the model was taken away from the agent. In this setting, we focused on policies of exploration like ε-greedy and Boltzmann. Back-up was the next concept implemented by exploring off-policy (Q-learning) and on-policy (SARSA) along with its Depth (n-step Q-learning and Monte Carlo updates). The environment provided for this assignment was Stochastic Windy Gridworld. This environment had 10*7 grids where the agent was allowed to move in all four directions (Right-Left-Up-Down). The random presence of wind in some columns made the environment Stochastic. In this assignment report, we will be going through the implementation, provide methodology, explain results and discuss observations.

2 DYNAMIC PROGRAMMING

Dynamic programming algorithms find solution of difficult problems by breaking them down into simple smaller problems. In this section, we implement one such Dynamic programming algorithm called the Q-value iteration algorithm. In this algorithm, the idea is to iteratively update the estimated state-action value (Q-value table) for each state-action pair present in the provided model to find optimal Q-values. The algorithm update's state-action value by the following equation:

$$Q(s,a) \leftarrow \sum_{s'} p(s'|s,a) \cdot \Big[r(s,a,s') + \gamma \cdot \max_{a'} Q(s',a')\Big] \, (Moerland,\ 2024) \tag{1}$$

Where,

- $Q(s,a)$ represents the Q value at state-action pair (s,a).
- $P(s' \mid s,a)$ is the probability of moving to state s' after performing action a in state s.
- $R(s,a,s')$ represents the reward earned after moving from state s to state s' as a result of action a.
- γ is the discount factor which defines the significance of future rewards over immediate rewards.
- $\max Q(s',a')$ represents the maximum Q value in the next state s'.

2.1 METHODOLOGY

1. Start by initialising the Q value for every state-action pair $Q(s,a)$ to zero and setting an error threshold.
2. Next, the Q value for every state-action pair is updated iteratively with the help of the Bellman optimality equation(1) along with the calculation for the maximum absolute error in $Q(s,a)$.
3. Repeat the iteration until all state-action pair values converge. $Q[s,a]$ is said to be converged when the maximum absolute error at a sweep is less than the threshold.

4. After convergence, the optimal policy is selected which maximises $Q(s,a)$ by selecting action a. A greedy policy is one such policy which selects an action that provides the highest estimated reward. The greedy policy is implemented as an optimal policy with the help of equations,

$$\pi(s) = \arg\max_{a} Q(s,a).(Moerland,\ 2024) \tag{2}$$

Where:

- $\pi(s)$ is a policy defined for state s,
- $\arg\max$ is an operator to maximize over action a,
- $Q(s,a)$ represents the Q value at state-action pair (s,a).

5. Simulate episodes and observe cumulative rewards to evaluate π(s) in comparison to the environment

2.2 RESULT

We iterated over the Q values at each state action pair using the Q value iteration process. Q(s, a) was initially set to zero, but when iterations occurred, the Q values were updated.

- In the beginning, the Q values generated poor estimates of the actual expected returns. (Figure 1)
- Midway, the generated Q-Values were closer to the genuine values, but not optimum. When the maximum absolute error fell below the threshold, we had achieved convergence. (Figure 2)
- At convergence, the Q-values represented the predicted total reward from following the best policy for each state-action pair. (Figure 3)

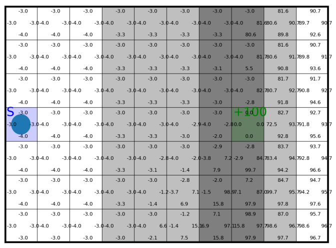

Figure 1: Q-value Iteration in the beginning

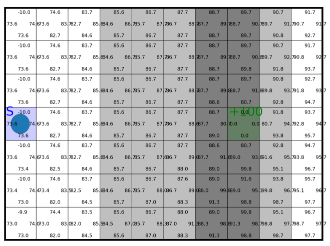

Figure 2: Q-value Iteration Midway

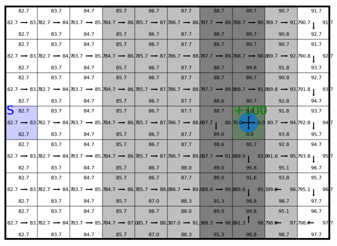

Figure 3: Q-value Iteration over the environment at convergence

Figure 3 above depicts the projected reward at every time step for every action in all states. These are the final Q-values at convergence, which give the agent the shortest path to the goal state. To get to the goal state in the shortest number of steps, the agent must choose the optimal action with the largest reward.

2.3 OBSERVATION

- For the start state (0,3) referred to as S=3 we have four actions (Up, Down, Left, Right). As shown in Figure 1, each of these actions has a value associated with them (82.7, 82.7, 82.7, 83.7 respectively). V*(S=3) will be the maximum of these values, which is 83.7. V*(S=3) = 83.7 means that if the agent follows the optimal policy(greedy) from start to goal state the expected total reward at the end will be 83.7.
- To find the average reward per time step we need to first find how many steps are taken by the agent to reach the goal state (iteration) and calculate the total reward. The agent at every step gets a reward of -1 and reaching the goal state gives it +100 as a reward. The optimal value of state is 83.7 in our case. From this, we can compute that the average steps taken are 16.3. Steps cannot be in decimal so, the average steps taken are 17. Therefore, the average reward per step on optimal value is around 4.92 (83.7/17).

- The `experiment()` function in `DynamicProgramming.py` creates an object of the class `StochasticWindyGridworld` in `environment.py`. Upon initialization, `_construct_model()` is called which handles $S = 52$ as a terminal state. If the location s is the goal state, `_construct_model()` turns it into a self-loop by assigning a transition probability of 1.0 and giving out no reward. Convergence is achieved through the `step()` method in the class `StochasticWindyGridworld`. It checks the state for the goal state, provides a boolean value (`done`), and gives a reward of $+100$. We also reset the evironment when at goal state with `reset()`. An alternative approach could involve using absorbing states.
- Changing the location of the goal states to (6,2) drops the optimal value of the start state to 64.9. It means the average time steps required to reach the goal state increased and the average reward per time step decreased. It is due to the fact that the agent is taking a different path to reach the new goal state with the optimal policy.

3 Exploration

In this section, our reinforcement learning setting changed to model-free. Now we no longer iterated over the state space with greedy policy. In this section we used Q-learning algorithm which is model-free. It functions by learning an action-value function that calculates the projected reward of performing a specific action in a particular state and then follows the optimal policy. Since in this setting, we were not guaranteed to visit all the states, we had to implement the concept of exploration. We did so by changing our policy to ε-greedy policy and Boltzmann policy (softmax). ε-greedy policy:

$$\pi(a|s) = \begin{cases} 1.0 - \varepsilon \cdot \frac{|A|-1}{|A|}, & \text{if } a = \arg\max_{b \in A} \hat{Q}(s,b) \\ \frac{\varepsilon}{(|A|)}, & \text{otherwise} \end{cases} (Moerland,\ 2024) \tag{3}$$

- $\pi(a|s)$ represents the probability of choosing action a while in state s.
- ε is the exploration rate, where $\varepsilon = 1$ indicates a uniformly random policy, while $\varepsilon = 0$ indicates a greedy policy.
- $|A(s)|$ is the number of actions that can occur in state s.
- $\hat{Q}(s,a)$ represents the estimated value of performing action a in state s.

The Boltzmann policy (also known as the softmax action selection) can be written as:

$$\pi(a|s) = \frac{e^{\frac{\hat{Q}(s,a)}{\tau}}}{\sum_{b \in A} e^{\frac{\hat{Q}(s,b)}{\tau}}} (Moerland,\ 2024) \tag{4}$$

where:

- τ is the temperature parameter controlling the degree of exploration.
- A is the set of all actions that can occur in state s.

The back-up estimate G_t is computed as:

$$G_t = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')(Moerland,\ 2024) \tag{5}$$

where:

- r_t indicates the immediate reward received at state s_t.
- γ is the discount factor, ranging from 0 to 1, indicating the importance of future rewards.

The tabular learning update is performed with the following equation:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot \left[G_t - \hat{Q}(s_t, a_t) \right] (Moerland,\ 2024) \tag{6}$$

where:

- α is the learning rate, which affects the speed and stability of learning.

3.1 METHODOLOGY

1. The environment, agent and required variables are initialized in the `q_learning()` function in order to track performance and carry out evaluation.

2. Then we iterate till the budget ($n_{\text{\_timesteps}}$):

   (a) On each step we get action a with `select_action()`. This method is defined in `Agent.py`. It uses eq(2), eq(3) and eq(4) to return the desired selected policy.

   (b) With selected action a, we call `step()` in `environment.py` which returns the new state, reward associated and (`done`) (boolean indicating goal state reached or not).

   (c) With these values we do Tabular learning update in `update()` which is defined in class `QLearningAgent`. We find the backup estimate G_t with eq(5) and then do Tabular learning update with eq(6).

   (d) We evaluate to find the mean return for given intervals. When `done` is `True`, we reset state s to start state.

3. Steps 2 to 5 are repeated till the budget ($n_{\text{timesteps}}$) is reached. Once reached we return the evaluation returns and the corresponding time steps arrays.

4. We repeat this for different values of ϵ and τ for ϵ-greedy and softmax respectively to check performance at different exploration parameters.

3.2 RESULT

With the evaluation returns and corresponding time steps arrays, we were able to plot a graph, comparing ε greedy and soft max policies at different values of ε and τ. In the graph, we also compared these policies with dynamic programming optimums.

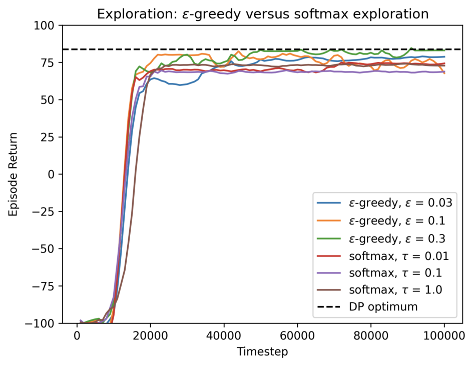

Figure 4: Comparison between ϵ-greedy and softmax policies at different values of ϵ and τ.

In the plotted graph above, we can see that the performance of ε-greedy policy is better than the softmax policy at different values of ε and τ. ε-greedy at ε=0.03 converges fast with stability. Whereas, ε=0.1 and ε=0.3 also converge quickly but have a bit of performance fluctuations. With more time steps ε=0.3 performance matches the DP optimum. Softmax at τ=1.0 takes time to converge but can perform better than softmax with temperature values at τ=0.1 and τ=0.01.

In this Gridworld environment, I would prefer ε greedy policy over softmax because its performance is better on a limited number of state spaces where exploitation helps. While with an increase in the complexity of the environment, softmax might perform better than ε-greedy policy. Reinforcement learning can come very close to the dynamic programming optimum performance. In dynamic programming, we iterated over the entire state space but in reinforcement learning with no model access the performance achieved is very good. With the help of exploration and more time steps, reinforcement, learning could learn and reach optimal performance of DP.

3.3 OBSERVATION

- With the second goal introduced in the environment at (3,2), the performance received dropped significantly. This is because the new goal, being closer to the start state, allows the agent to reach it first and terminate the episode. Given that this goal has a reward of +5, which is less than the original reward of +100, the evaluation returns are significantly lower.
- As the exploration parameters are low, the agent will continue to prioritize exploitation to reach the new goal, which offers a lower reward due to the shorter distance between the start state and the new goal. This affects the exploration-exploitation trade-off, as exploitation is preferred, leading the agent to a lower rewarding goal state. In this situation, the agent must explore more to balance its strategy.
- The optimal exploration parameters, such as ϵ for the ϵ-greedy strategy and τ for the soft-max approach, must be adjusted upwards to encourage more exploration. With increased exploration, the agent is more likely to explore other states, which may help it to discover the original goal state with the higher reward of 100.

4 Back-up: On-policy versus off-policy target

In this section, we implemented and discussed the difference between on-policy backups and off-policy backups at different learning rates. In the last section, we implemented the Q learning algorithm (off-policy back-up) with equation (5) where to find the optimal policy, it took the best available action at the next state. In on-policy, the idea is to focus on the action taken by the agent and follow it instead of choosing the optimal value. In this section, we implemented SARSA which follows on-policy back-up. It was implemented with the following equation.

$$G_t = r_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1})(Moerland,\ 2024) \tag{7}$$

Where: $\hat{Q}(s_{t+1}, a_{t+1})$ denotes the anticipated value of action a_{t+1} in next state s_{t+1}.

For tabular learning update we used equation (6)

4.1 METHODOLOGY

1. The environment, agent, and required variables are initialized in the `sarsa()` function to track performance and carry out evaluation. A sample state is initialized, and the desired policy is selected.
2. Then we iterate until the budget (`n_timesteps`) where on each step we get action a with `select_action()`. This method is defined in `Agent.py`. It uses equations (2), (3), and (4) to return the desired selected policy.
3. With the selected action a, we call `step()` in `environment.py` which returns the new state, reward associated, and `done` (a boolean indicating if the goal state is reached or not).
4. With these values, we do Tabular learning updation in `update()` which is defined in the class `SarsaAgent`. We find the on-policy backup estimate G_t with equation (7) and then do Tabular learning updation with equation (6).
5. We evaluate to find the mean return for given intervals. We move to the next state and continue to use the same policy for the next action selected in the iteration. When `done` is True, we reset the state s to the start state. Otherwise, steps 2 to 5 are repeated until the budget (`n_timesteps`) is reached. Once reached, we return the evaluation returns and the corresponding time steps arrays.
6. We repeat this SARSA implementation and the Q-learning implementation for different learning rates α to do a comparison.

4.2 RESULT

With the evaluation returns and corresponding time step arrays, we were able to plot a graph, comparing on-policy and off-policy backups. In the graph, we compare Q-learning (off-policy) and SARSA (on-policy) at different values of learning rate α.

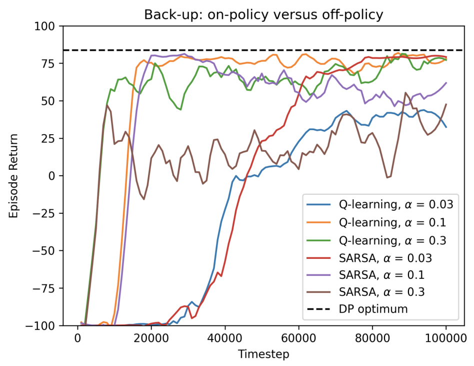

Figure 5: Comparison between on-policy and off-policy at different learning rates α.

In the above graph, Q learning at learning rate α = 0.1 performs the best with stable learning and high episode returns. Also, Q learning performs significantly better at α = 0.1 and α = 0.3 when compared to SARSA at the same learning rate. However, SARSA outperforms Q learning with a lower learning rate (α = 0.03). SARSA at α = 0.03 takes around 20000 steps to start learning but its performance quickly comes close to DP optimum. Q learning at α = 0.1 and SARSA at α = 0.03 outperformed all other values.

4.3 OBSERVATION

- In the given GridWorld environment, I would prefer the Q learning algorithm (off-policy) because its performance is better than SARSA. The agent can achieve high episode returns quickly. The learning in Q learning is comparatively faster because the off-policy always selects the maximum return action despite the selected policy.
- I would prefer SARSA in complex stochastic environments like real-world situations. In the real world, the risk of exploitation could be very high and dangerous. Hence, choosing Q-learning will not be feasible. We need to use on-policy back-up as it focuses on the selected policy and follows it, which will promote stability and limit the risk.

5 BACK-UP: DEPTH OF TARGET

In this section, we explored depth which is a crucial part of back-up in reinforcement learning. In this section, we implemented n-step methods which could add multiple rewards before bootstrapping them. Specifically, we implemented n-step Q-learning and Monte Carlo update. The purpose of this section is to compare various depths of backup targets. In n-step Q-learning the following equation is used to generate the target:

$$G_t = \sum_{i=0}^{n-1} (\gamma^i \cdot r_{t+i}) + \gamma^n \max_a Q(s_{t+n}, a) \quad (Moerland,\ 2024) \tag{8}$$

where:

- r_{t+i} represents reward at $t + i$ time step,
- $Q(s_{t+n}, a)$ denotes the action-value function estimate of state s_{t+n} and action a.

Equation (6) was used to do a Tabular learning update.

For Monte Carlo method, we add all the rewards in an episode and without doing any bootstrapping. The equation used was:

$$G_t = \sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} (Moerland,\ 2024) \tag{9}$$

Again, Equation (6) was used to do tabular learning update

5.1 METHODOLOGY

The methodology for implementing the n-step Q-learning is as follows:

1. The `n_step_Q()` function initializes the environment, agent, and necessary variables for performance tracking and evaluation.
2. We iterate until the budget is reached (`n_timesteps`), initializing empty arrays for `states[]`, `actions[]`, and `rewards[]`. We initialize a counter.
3. We build an inner loop to iterate until `max_episode_length` (100). We select an action and call `step()` in `environment.py` to obtain the new state, associated reward, and `done` (whether the goal was reached or not). These values are added to `states[]`, `actions[]` and `rewards[]`.
4. We incremented the counter and evaluate to find mean return for given intervals. When `done` is True, we exit the inner loop.
5. With these values, we do Tabular learning updation in `update()` which is defined in class `NstepQLearningAgent`.
6. We initialise the episode length and find `m` (number of rewards left to sum).
7. If the state, `m` distance away from current is terminal (checked by with return[iteration + m] $>$ -1), we perform n-step target without bootstrap equation (9). Otherwise, we use equation (8). We update tabular learning with equation (6).
8. We add our counter to iteration (to prevent over-training) and continue for the next episode.
9. Steps 2 to 7 are repeated till budget (`n_timesteps`) is reached. Once reached we return evaluation returns and the corresponding time steps arrays.
10. We perform this for 1-step, 3-step and 10-step Q learning.

The methodology for implementing the Monte Carlo is as follows:

1. The environment, agent and required variables are initialized in the `monte_carlo()` function to track performance and carry out evaluation.
2. Similar steps as n-step Q-learning from Steps 2 to 4.
3. With these values, we do Tabular learning updation in `update()` which is defined in class `MonteCarloAgent`
4. We initialise the episode length.
5. We perform a Compute Monte Carlo target at each step equation (9). Then we do Tabular learning updation with equation (6).
6. We add our counter to iteration(to prevent over-training) and continue for the next episode.
7. Steps 2 to 7 are repeated till budget (`n_timesteps`) is reached. Once reached we return evaluation returns and the corresponding time steps arrays.

5.2 RESULT

From evaluation returns and time step arrays, we created a graph comparing 1-step, 3-step, 10-step, and Monte Carlo Q-learning. The graph below shows that 3-step Q-learning is the fastest learner. It could have balanced immediate and future rewards. Although it takes time to learn, 1-step yields returns comparable to DP optimum. Even after multiple time steps, 10-step Q-learning fails to yield good episode returns. Monte Carlo has poor performance in high stochastic environments due to limitations with full episode returns.

5.3 OBSERVATION

The performance of the Q-learning implementation outperforms the n-step Q-learning implementation with n=1. This might be due to the way n-step is implemented. It is not completely off-policy. Instead of choosing the maximum yielding action, we follow the selected policy for the first n rewards. Therefore, a behavioural policy is followed by the target in n-step implementation causing the difference in performance.

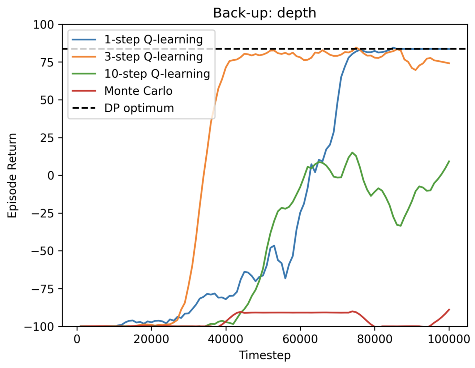

Figure 6: Comparison between 1-step Q-learnimg, 3-step Q-learning, 10-step Q-learning and Monte Carlo.

6 REFLECTION

- DP versus RL: The strength of DP is that with full knowledge of the model, it guarantees optimal returns which are better than the model-free RL. Its weakness is as the model becomes more stochastic, state-action space increases exponentially (Curse of dimensionality). This makes DP not feasible.
- Exploration: My preference depends on environment. For simple stochastic environments, I prefer ε-greedy for effective maximum return action. In complex stochastic environments, I prefer softmax to minimize exploitation risks with the policy. A better exploration method is to adjust hyperparameters over time. Adjusting the balance between exploration and exploitation at different timesteps may improve agent performance.
- Exploration 2: With the default reward for steps being -1, the agent has pressure to find the goal state in the least number of steps. In this case, learning is quick. When the reward per step is changed to zero, the episode reward estimate though increased but the learning became very slow. This is because the agent is not being punished for taking extra steps.
- Discount factor: In the case where John puts discount factor = 1.0 will not yield a short path as with $\gamma = 1.0$ the future rewards and immediate rewards are treated equally. With γ = 0.99 there is some priority given to the immediate reward which might lead to a shorter path. Even though both will find the goal state, γ = 0.99 has a better chance of finding the shortest path.
- Back-up, on-policy versus off-policy: In On-policy SARSA, the idea is to focus on the action taken by the agent and follow it instead of choosing the optimal value. This makes SARSA safe in complex environments like the real world. The disadvantage is that sometimes SARSA performance is not optimum as it does not use the high-yielding action. In our implementation, n-step is not completely off-policy nor on-policy. Even though we maximise over the last action but still instead of choosing the maximum yielding action, we follow the selected policy for the first n rewards. Therefore, a behavioural policy is followed by the target in n-step implementation.
- Back-up, target depth: In 1-step we prefer immediate rewards. This generates low variance but high bias. In n-step, a balance is struck. With multiple steps, it reduces the bias while maintaining variance. It also learns faster. Monte Carlo has very low bias and high variance in complex environments as it considers full episodes together. My preference would be N-step as it strikes a balance with a bias-variance trade-off. N-step methods propagate information faster. The chances of 1-step are high to converge on the optimal policy.
- Curse of Dimensionality: Tabular RL algorithms perform well in discrete state spaces but struggle in continuous state spaces. Tabular RL algorithms struggle to handle the exponential growth of the state action space as dimensions increase. This is also known as the Curse of Dimensionality. Deep learning can identify patterns and predict future moves, overcoming this challenge. This reduces the need to iterate through the entire state-action space.

References

Thomas Moerland. *Assignment: Tabular Reinforcement Learning, Leiden University*. 2024.