**Report: Assignment 1 - Defining & Solving RL Environments: Part 1.1**

**Assignment Overview:**

The goal of the assignment is to acquire experience in defining and solving reinforcement learning environments, following OpenAI Gym standards. The assignment consists of two parts. The first focuses on defining deterministic and stochastic environments that are based on Markov decision process. In the second part we will apply two tabular methods to solve environments that were previously defined.

**Problem Statement for Part 1.1**

* Define a deterministic environment, where P (s′, r | s, a) = {0, 1}. Run a random agent for at least 10 timesteps to show that the environment logic is defined correctly.
* Define a stochastic environment, where Sum( P(s′,r|s,a)) = 1. A modified version of the environment defined in Part 1.1 should be used. Run a random agent for at least 10 timesteps to show that the environment logic is defined correctly.

**Environment requirements:**

* Min number of states: 12
* Min number of actions: 4
* Min number of rewards: 5

**Specification of my Environment:**

* No of States: 25
* Number of Actions to be taken: 4
* Number of Rewards = 5 Rewards in total
  + +15 reward for reaching the goal state [4,4]
  + +3 reward for reaching states [2,3] and [3,2] – For guiding the agent towards the goal state.
  + -1 reward for reaching the states [0,3] and [3,0] – For telling the agent that its going away.
* The agent start state is at the left hand top [0,0]
* The goal state is at bottom right corner [4,4]

**1: Question: Describe the stochastic and deterministic environment which were defined.**

**The Base Environment:**

* **Set of Actions** – {0 -> up, 1 -> down, 2 -> left, 3 ->right) where (0,1,2,3) are the actions and (up, down, left, right) are the actual corresponding physical actions that the agent takes in the environment.
* **States** – {S1, S2,S3,…..S25} – These states correspond to the individual cells in the environment matrix – {[0,0],[0,1],…….[4,4]}, where [0,0] is the agent start state, [4,4] is the goal state, [0,3],[3,0],[2,3],[3,2] and [4,4] are the reward states. An additional reward state ([4,4]) has been initialized from the initial report since the agent wasn’t converging to the final state. Another issue was that the agent was oscillating between the reward states (based on the updated q-values). In order to get the agent “unstuck” and transition towards the goal state, we had to initialize a goal-state reward.
* **Rewards** – {-1,-1,3,3,15} – There are five rewards in the environment, the two negative rewards for nudging the agent towards the goal state, two positive rewards for the encouragement for taking the right steps and a final reward for reaching the goal state. Once an reward has been given out, it gets deleted from the system and isn’t available for the rest of the episode. As mentioned before, a reward of 15 has been initialised for reaching the goal state. This reduces the oscillation problem and helps the agent converge to the goal state faster.
* **The Main Objective:** The main objective of the agent is to reach the “goal state” ([4,4] in my environment) while collecting the maximum sum of rewards in least amount of time-steps.
* **Optimization and Safety:** One optimization done in my state is that of limiting an agents action space. If an agent is in state [0,0], the agent can transition only into [1,0] and [0,1]. So the environment doesn’t present the agent with states [-1,0] and [0,-1], as those are beyond the environments boundaries. So instead of choosing from 4 actions at any state, the number of actions that an agent can choose from, or the states that the agent can transition into, are completely dependent on the state that the agent is currently in. Another example is if the agent is in state [3,4], the agent can transition only into [2,4], [3,3] and [4,4].
  + There are multiple advantages of this.
    - The states the agent can go to are clear based on the state it’s in. As a result, it’s easy to create a mental picture of the environment and the agents position at any given time.
    - Second of all, the convergence to goal state is faster since the agent isn’t given a choice to go beyond the boundary in the first place. For example, if the agent is in [0,0] and it chooses to go up [-1,0], then the least amount of steps to goal state [4,4] is 9 – including the attempt to transition into [-1,0], while 8 is the least amount of steps from [0,0] to [4,4]. Now this is for when agent is at [0,0]. But the agent goes via either of the bordering cells [1,0], [0,1],[3,4] and [4,3], so the policy restricts the actions, thus reducing the overall total time taken to reach the goal state.
    - Third of all, because of the restricted state-transition pairs, we don’t need to update some of the q-value (state-action) pairs since the agent is never given the opportunity to taken an action based on them, or consider them for choosing an action. Consider the following sample trained q-value table (syntax [state] -> [qup, qdown, qleft, qright])
    - (0, 0): array([0. , 0.02489157, 0. , 0.02489798]),
    - (0, 1): array([0. , 0.08338281, 0.00741871, 0.08307087]),
    - (0, 2): array([0. , 0.27744099, 0.02477664, -0.25208672]),
    - (0, 3): array([0. , 0.96040055, 0.08307461, 0.1184819 ]),
    - (0, 4): array([0. , 0.3952955 , 0.28537415, 0. ]),
    - (1, 0): array([0.00741923, 0.08103377, 0. , 0.08332043]),
    - (1, 4): array([0.11827047, 1.32355446, 0.94387626, 0. ]),
    - (2, 0): array([0.02431006, 0.08003054, 0. , 0.28060873]),
    - (2, 4): array([0.39085875, 4.43998435, 1.32787628, 0. ]),
    - (3, 0): array([0.08221389, 0.11961928, 0. , 0.80633628]),
    - (3, 4): array([ 1.32176049, 14.85466648, 1.32557003, 0. ]),
    - (4, 0): array([0.13028776, 0. , 0. , 0.39903201]),
    - (4, 1): array([0.7817462 , 0. , 0.11959123, 1.33200749]),
    - (4, 2): array([1.42978918, 0. , 0.3981079 , 4.44769707]),
    - (4, 3): array([ 1.32026465, 0. , 1.32362425, 14.89826654]),
    - (4, 4): array([0., 0., 0., 0.])}
    - Here you can clearly see that state- action pair of [0,0] -> up, left do not get updated.
    - You can also see that state- action pair of [0,1] -> up do not get updated. Etc.
    - Now it’s logical to think that okay, if a state action q value becomes negative, wont the agent choose the 0 option going ahead, since that is technically greater than the negative numbers. However, my environment doesn’t let the agent evaluate the q-values with 0, since that is a state that the agent cannot go in. So the agent evaluates the negative q-values only, and updates those in the future.
  + Another added advantage of this method, is that it ensures the safety of the environment, as it won’t break in any condition. For example, if the agent was to transition into [-1,0] from [0,0], while evaluating the state [-1,0] it would break the environment as its not part of the environment and there are no q-value pair to evaluate from. This makes the entire process more robust and efficient at the same time.

**The Deterministic environment:**

The deterministic environment was defined in a simple way such that whatever action the agent takes, there is no influence on the transition by my environment of any sort. For example if the agent at state [0,0] decides to go down or go up (based on epsilon greedy method), the state will change according without any uncertainty criteria. Along with that, the optimization that was mentioned above remain in place.

**The Stochastic environment:**

The Stochastic environment was designed in such a way that, post choosing an action via the epsilon greedy method, the environment tells the agent to take a random action based on a probability. For example, if the agent chooses to go up, there is a 80% chance that the environment tells the agent to go up, the rest of the 20% is split between the other states the agent can go to. However the sum of all the transition probabilities is equal to 1. Along with that, the optimization that was mentioned above remain in place.

**2: Show your visualizations:**

**My Deterministic Environment Rendering**

|  |  |
| --- | --- |
| A picture containing text, clock  Description automatically generated  Step 1: | A picture containing text, clock  Description automatically generated  Step 2 |
| A picture containing text, clock  Description automatically generated  Step 3 | Logo  Description automatically generated  Step 4 |
| Logo  Description automatically generated  Step 5 | A picture containing logo  Description automatically generated Step 6 |
| Logo  Description automatically generated with medium confidence  Step 7 | Icon  Description automatically generated with medium confidence  Step 8 |
| A picture containing shape  Description automatically generated  Step 9 | A picture containing shape  Description automatically generated  Step 10 |
| A picture containing shape  Description automatically generated  Step 11 | Final Reward : -1 + 3 + 3 = 5 |

**3: How did you define the stochastic environment?**

For the stochastic environment, the agent would first take the action as per the epsilon greedy policy. Once the agent takes the action as per the policy, the environment will then take that action and put it through its stochastic process, ie, if the agent chooses to go right, there is a 80% chance it goes right, and 20% chance it goes to the other remaining states. For example, if the agent is in state [0,0] and the policy states that the agent must go right, there is 20 percent chance it goes down. Another example, if the agent is in state [0,3] and the policy tells the agent to go right, then there is a 80% chance the agent goes right, 10% chance agent goes down and 10% chance the agent goes left. Another example. If the agent is in [1,1] and the policy tells the agent to go right, then there is 80% chance the agent goes right, 6.6667% chance the agent goes left, 6.6667% chance the agent goes up and 6.6667% chance the agent goes down.

**Here is the rendering for my stochastic environment:**

|  |  |
| --- | --- |
| A picture containing text, clock  Description automatically generated  Initial State  Epsilon 0.3 | A picture containing text, clock  Description automatically generated  Current Agent POS [0, 0]  Agents want to our Right or the Agents Left  The Agent Ended Up Going Right  Final Agent POS [0, 1] |
| A picture containing text, clock  Description automatically generated  Current Agent POS [0, 1]  Agents want to our Right or the Agents Left  The Agent Ended Up Going Right  Final Agent POS [0, 2] | Logo  Description automatically generated  Current Agent POS [0, 2]  Agents want to our Right or the Agents Left  The Agent Ended Up Going Right  Final Agent POS [0, 3] |
| Logo  Description automatically generated  Current Agent POS [0, 3]  Agents want to our Down  The Agent Ended Up Going Up  Final Agent POS [0, 3] | Logo  Description automatically generated  Current Agent POS [0, 3]  Agents want to our Down  The Agent Ended Up Going Down  Final Agent POS [1, 3] |
| A picture containing text, clock  Description automatically generated  Current Agent POS [1, 3]  Agents want to our Our Left or the Agents Right  The Agent Ended Up Going Left  Final Agent POS [1, 2] | A picture containing logo  Description automatically generated  Current Agent POS [1, 2]  Agents want to our Down  The Agent Ended Up Going Down  Final Agent POS [2, 2] |
| A picture containing logo  Description automatically generated  Current Agent POS [2, 2]  Agents want to our Right or the Agents Left  The Agent Ended Up Going Right  Final Agent POS[2,3] | Icon  Description automatically generated with low confidence  Current Agent POS [2, 3]  Agents want to our Right or the Agents Left  The Agent Ended Up Going Right  Final Agent POS [2, 4] |
| Icon  Description automatically generated with medium confidence  Current Agent POS [2, 4]  Agents want to our Down  The Agent Ended Up Going Down  Final Agent POS[3,4] | Total Reward = -1 +3 = 2 |

**4: What is the difference between the deterministic and stochastic environments?**

The main difference between deterministic and stochastic environment is that in deterministic environment, the environment doesn’t affect the agents actions apart from the limits set for it. For example if the agent chooses to go beyond the boundary of the environment or it enters a fail state (state where the episode ends), then the environment will affect the states involved. However otherwise for legal/appropriate actions (including rewards, etc), the environment will not affect the agent from taking an action. However for stochastic environments, post taking action via the chosen policy (epsilon-greedy), the environment will randomly return a random action with the support of the action that the agent has chosen. For example, if the agent chooses to go right, there is a ((1-epsilon) x100) % chance that the agent goes right (please note that epsilon is set at initialization and not hardcoded). The rest of the epsilon% probability is distributed evenly among the other remaining available actions. So if the agent is in position [1,1], and it chooses to go right, the epsilon greedy method might choose to go towards the goal state (greedy – goal state is at [4,4]) i.e. from [1,1] the agent will take a right [1,2]. Post that selection, the stochastic environment will instead tell the agent to go down or up or left, including from the chosen action. So if the agent chooses to go right(via epsilon policy), there is a 80% chance the agent goes right, and 20% chance it goes to the other states.

**Report: Assignment 1 - Defining & Solving RL Environments: Part 1.2**

**Problem Statement for Part 1.2**

Apply two tabular methods to solve both the deterministic and stochastic environments that were defined in Part 1. You need to implement Q-learning and any other tabular algorithm of your choice (e.g. SARSA, Double Q-learning, Monte Carlo or n-step bootstrapping).

**Tabular Methods Chosen:**

* Method 1: Q-Learning
* Method 2: Double-Q Learning

**Q-Learning Method:**

* Q-learning is a simple model-free reinforcement learning algorithm, that follows a “learned q-table” that tries to maximize the possible rewards over one iteration (or episode). It is a fairly straightforward algorithm that computes a Q-table and takes action based on that. The q-table consists of state as the index and the actions that the agent can take as the column. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Action Up | Action Down | Action Left | Action Right |
| Start State – 0,0 | 0.00 | -1 | 0.00 | 0.05 |
| Middle states – 1,1 | -0.05 | 1.25 | 0.236 | 1.25 |
| End state – 4,4 | 0 | 0 | 0 | 0 |

* At each stage, the agent updates the q-values based on multiple factors such as reward in the next state, the learning rate, the gamma function, etc.
* The updating rule follows the form :
* Q-val (s,a) -> Q-val(s,a) + learning\_rate\*(reward + gamma \* max(Q-val(selected\_state) - Q-val (s,a))
* The updation rule ensures that the agent converges on the optimal path in a efficient manner.

**Double-Q Learning Method:**

* The idea behind in DQ Learning is that we utilize two q-value tables and use their values for choosing an action that the agent should take.
* How this works is that we maintain two q-value tables, say A and B.
* If we want to update table A:
  + Action for which Q-value is max in Table A.
  + Choose that state-action pair q-value from Table B.
  + Then update the q-value in table A.
* Update A:
* Q-vala (s,a) -> Q-vala(s,a)+ LR\*(reward + gam \* Q-valb(selected\_state)(action for q-max(Q-vala(s))- Q-vala(s,a))

**Show and Discuss results after:**

**Applying Q-learning to solve the deterministic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.**

(the code that applies Q-learning to solve the deterministic environment defined in Part 1 is mentioned in Q\_Learning\_Environment\_Final.py).

**Parameters used:**

Learning rate – 0.3, gamma 0.6, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

* Q – Learning Reward per episode for a Deterministic Environment

A picture containing timeline

Description automatically generated

* Q – Epsilon per episode for a Deterministic Environment

Shape

Description automatically generated

* Here we can observe that the agent does a good job of converging on the final optimal path in a relatively quick time. Apart from a few random spikes because of the agents policy, the agent regularly converges on the optimal path with maximum rewards.

**Applying Q-learning to solve the Stochastic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.**

(the code that applies Q-learning to solve the stochastic environment defined in Part 1 is mentioned in Q\_Learning\_Environment\_Final.py).

**Parameters used:**

Learning rate – 0.3, gamma 0.6, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

* Q – Learning Reward per episode for a Stochastic Environment.

**Timeline

Description automatically generated with medium confidence**

* Q – Epsilon per episode for a Stochastic Environment

Shape

Description automatically generated

* Here we can observe that the agent does a good job of converging on the final optimal path in a relatively quick time. However, due to the stochastic nature of the environment, the agent receives more rewards that the reward available on the standard optimal path (due to it taking a random turn). Along with that, we can also see that the agent receives less the optimal number of rewards, indicating that the agent is reaching the final state without going through the optimal path. By observing the above graph, we can say that the agent has fully learned the environment.

**Applying any algorithm of your choice to solve the Deterministic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.**

(the code that applies DQ-learning to solve the deterministic environment defined in Part 1 is mentioned in DQ\_Learning\_Environment\_Final.py).

**Parameters used:**

Learning rate – 0.3, gamma 0.6, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

* DQ – Learning Reward per episode for a Deterministic Environment

Box and whisker chart

Description automatically generated with low confidence

* DQ – Learning Epsilon per episode for a Deterministic Environment

Shape

Description automatically generated

* Here we can observe that the agent does a good job of converging on the final optimal path in a relatively quick time, similar to the Q-learning graph. Apart from a few random spikes on the graph ,due to epsilon greedy or a rouge q-value that was updated later on, we can see that the agent converges on a consistent optimal path and receives a uniform maximum reward.

**Applying any algorithm of your choice to solve the Stochastic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.**

(the code that applies DQ-learning to solve the stochastic environment defined in Part 1 is mentioned in DQ\_Learning\_Environment\_Final.py).

**Parameters used:**

Learning rate – 0.3, gamma 0.6, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

* DQ – Learning Reward per episode for a Stochastic Environment.

A picture containing graphical user interface

Description automatically generated

* DQ – Learning Epsilon per episode for a Stochastic Environment.

**Shape

Description automatically generated**

* Here we can observe that the agent does a good job of converging on the final optimal path in a relatively quick time. We can probably say that it converged faster than the q-learning algorithm, however that can be contested. However, due to the stochastic nature of the environment, the agent receives more rewards that the reward available on the standard optimal path (due to it taking a random turn). Along with that, we can also see that the agent receives less the optimal number of rewards, indicating that the agent is reaching the final state without going through the optimal path. By observing the above graph, we can say that the agent has fully learned the environment.

**Provide the evaluation results. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.**

**Parameters used:**

Learning rate – 0.3, gamma 0.3, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

Graphical user interface

Description automatically generated

* Here is the render of my deterministic environment after learning the greedy policy: A picture containing text, clock

  Description automatically generatedA picture containing text, clock

  Description automatically generatedA picture containing text, clock, first-aid kit

  Description automatically generatedShape, square

  Description automatically generated with medium confidenceA picture containing shape

  Description automatically generatedA picture containing icon

  Description automatically generatedA picture containing icon

  Description automatically generatedA picture containing logo

  Description automatically generated

A picture containing icon

Description automatically generated

Here is a render of my stochastic environment for greedy policy:

A picture containing text, clock

Description automatically generatedA picture containing text, clock

Description automatically generatedA picture containing text, clock

Description automatically generatedA picture containing text, clock

Description automatically generatedA picture containing text, clock, first-aid kit

Description automatically generatedA picture containing text, clock

Description automatically generatedA picture containing text, clock, first-aid kit

Description automatically generatedA picture containing logo

Description automatically generatedA picture containing shape

Description automatically generatedA picture containing icon

Description automatically generatedA picture containing icon

Description automatically generatedA picture containing logo

Description automatically generatedA picture containing icon

Description automatically generated

**Compare the performance of both algorithms on the same deterministic environment (e.g. show one graph with two reward dynamics) and give your interpretation of the results.**

**Parameters used:**

Learning rate – 0.3, gamma 0.3, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

**Box and whisker chart

Description automatically generated with low confidence**

* In the above graph, we can see that the Double Q Learning converges slightly faster as compared to the Q-learning algorithm. This could be due to the fact that the it avoids overestimating the q-values (which is done by using the second table). Another interpretation could be that the second table helps normalize the actions taken by the agent had it been following one table only. So for a simple grid-world scenario like the one chosen for my report, there is a good chance that Double Q learning performs better as compared to simple Q-Learning algorithm.
* However multiple comparisons of the runtimes of the both the algorithms didn’t yield any significant difference between the two algorithms for my grid-world. However, most likely, Double Q-learning will perform a lot better as compared to simple q-learning for more complex environments.

**Compare how both algorithms perform in the same stochastic environment (e.g. Show one graph with two reward dynamics ) and give your interpretation of the results.**

**Parameters used:**

Learning rate – 0.3, gamma 0.3, max-time-steps – 30, epsilon=0.9,epsilon\_decay=0.01, episodes = 1000

**A picture containing diagram

Description automatically generated**

* In the above graph, we can see that the Double Q Learning converges approximately the same as compared to the Q-learning algorithm. This could due to the fact the added stochasticity randomizes the actions, thus doing a better job at approximating the q-values from a long term perspective.
* Following on the above point, we can observe that both the learning algorithms are achieving similar rewards per episode (ie 18 rewards which are available along the optimal path).

**Hyperparameter Tuning Bonus [5 points]**

**Provide the analysis after tuning at least two hyperparameters listed below:**

* **Discount factor (γ)**
* **Epsilon decay rate**
* **Epsilon min/max values – Number of episodes**
* **Max timesteps**

**Try at least 3 different values for each of the parameters that you choose. Provide the reward graphs and your explanation for each of the results. In total you should have at least 6 graphs and your explanations. Make your suggestion on the most efficient hyperparameters values for your problem setup.**

**Hyper-parameter 1**: Max timesteps

* What is a time-step?
* Time-step is a unit measure of the times that the agent takes an action in the environment. If the agent decides to go from [1,1] to [1,2], that is considered to be one time-step. Similarly, for the purposes of our project, we set a maximum time-step which helps restrict the number of time-steps that an agent can take before the episodes terminate.
* So how does max-time-steps help?
* That will be answered with our reward per episode graphs:
* Max time-steps chosen [11,20,50,100]

**Graphical user interface

Description automatically generated**

* We are testing maximum-timesteps of 11,20,50 and 100 to see how the maximum time-steps affect the convergence of the agent to the max permissible reward and the finding the optimal path to the goal state.
* We can observe here that as we increase the max-time-steps parameter in the environment, the agent seems to become faster at converging on the optimal number of rewards in the optimal path, however if we decrease the max-number of time-steps to 11, the agent takes a significantly larger number of episodes to converge on the optimal path-rewards per episode.
* One main reason for this is that in the initial discovery phase (or the phase in which the agent takes random actions), the agent can roam around more and calculate more q-values in one iteration as compared to the number of q-values calculated for iteration with less number of time-steps. As the agent learns the optimal q-values more quickly with max time-step of 100, it converges quickly to the path with optimal number of time-steps and maximum possible rewards.
* So ideally, we want to have the max-steps between 20 and 100. Now how do we select from this?
* We need to trade-off between number of

**Hyper-parameter 1: Epsilon Decay**

* What the epsilon decay?
* Epsilon value is the value that the agent takes into consideration while choosing his/her next step/action/state. Either the action is a greedy one or a exploratory one. Its considered that the agent takes (1-epsilon)\*100% greedy actions and epsilon\*100% exploratory actions. However it’s important that this value reduces over time so as to allow the agent to use the “learned” values and exploit the system to get
* Time-step is a unit measure of the times that the agent takes an action in the environment. If the agent decides to go from [1,1] to [1,2], that is considered to be one time-step. Similarly, for the purposes of our project, we set a maximum time-step which helps restrict the number of time-steps that an agent can take before the episodes terminate.
* So how does max-time-steps help?
* That will be answered with our reward per episode graphs:

Graphical user interface

Description automatically generated

As we can see from the above observation, as the epsilon takes longer to decay (i.e. the epsilon decay becomes smaller) we can see that the agent takes longer and longer to converge on the optimal path in the given environment. When the decay is large, it immediately settles on the greedy path that it has found and doesn’t explore any more states, as may be required to find a higher reward optimal path. From the above graphs, we can take the value of epsilon decay as “0.1” if we want to quickly converge on the greedy path, or set it from “0.1 to 0.01”, which will provide a good balance of exploration and exploitation. Additionally, we may want to try out epsilon values of 0.002 to check if the agent does well over time.