CSE574 Project 4: Implementation of Reinforcement Learning

Mansi Wagh
Department of Computer Science
University at Buffalo
Buffalo, NY 14226
mansiwag@buffalo.edu

Abstract

The main objective of this project is to build a reinforcement learning agent to navigate the classic 4x4 grid-world environment. The agent will learn an optimal policy through Q-Learning which will allow it to take actions to reach a goal while avoiding obstacles. To simplify the task, we will provide the working environment as well as a framework for a learning agent. The environment and agent will be built to be compatible with OpenAI Gym environments and will run effectively on computationally-limited machines.

1. Introduction

1.1 Reinforcement Learning:

In simple terms, Reinforcement learning is learning best actions based on reward or punishment. It is a machine learning concept which focuses on how automated agents can learn to take actions in response to the current state of an environment so as to maximize some reward. This is framed as a Markov decision process (MDP), as displayed in Figure 1.

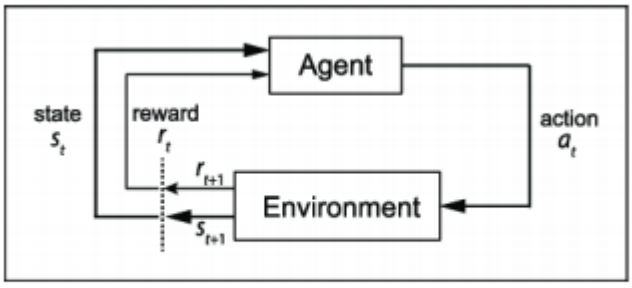


Figure 1: The canonical MDP diagram[1]

An MDP is a 4-tuple (S, A, P, R), where

• S is the set of all possible states for the environment

• A is the set of all possible actions the agent can take

• P = P r(st+1 = s 0 |st = s, at = a) is the state transition probability function

• R : S × A × S → R is the reward function

Our focus is finding an optimal policy**.** A policy tells us how to act from a particular state.We have to find a policy $\pi : S \rightarrow A$ which our agent will use to take actions in the environment which maximize cumulative reward, where

$$\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})$$

$\gamma \in [0, 1]$ is a discounting factor (used to give more weight to more immediate rewards), st is the state at time step t, at is the action the agent took at time step t, and st+1 is the state which the environment transitioned to after the agent took the action.

1.2 Q-Learning:

Q-Learning is a process in which we train some function to learn a mapping from state-action pairs to their Q-value, which is the expected discounted reward for our policy.It is the most commonly used reinforcement learning method, where Q stands for the long-term value of an action. Q-learning is about learning Q-values through observations.It comprises of below procedure:

In the beginning, the agent initializes Q-values to 0 for every state-action pair. Q(s,a) = 0 for all states *s* and actions *a*. This gives no information on long-term reward for each state-action pair.

After training the agent when it starts learning, it takes an action *a* in state *s* and receives reward *r*. It also observes that the state has changed to a new state *s'*. The agent will update Q(s,a) with this formula:

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{a} \right)}^{\text{learned value}}$$

The learning rate is a number between 0 and 1. It is a weight given to the new information versus the old information. The new long-term reward is the current reward, *r*, plus all future rewards in the next state, *s'*, and later states, assuming this agent always takes its best actions in the future. The future rewards are discounted by a discount rate between 0 and 1, meaning future rewards are not as valuable as the reward now.
In this updating method, Q carries memory from the past and takes into account all future steps. We use the maximized Q-value for the new state, assuming we always follow the optimal path afterward. As the agent visits all the states and tries different actions, it ultimately learns the optimal Q-values for all possible state-action pairs. Then it can derive the action in every state that is optimal for the long term.

2.Environment

Reinforcement learning environments can take on many different forms, including physical simulations, video games, stock market simulations, etc. The reinforcement learning community (and, specifically, OpenAI) has developed a standard of how such environments should be designed, and the library which facilitates this is OpenAI's Gym.

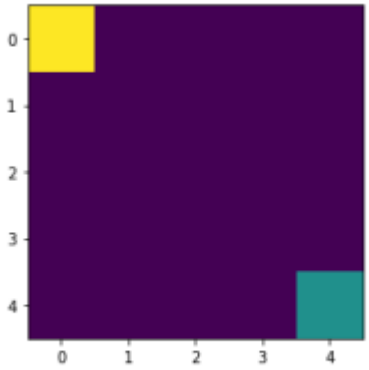


Figure 3: The initial state of our basic grid-world environment.

The environment in our project is a n × n grid-world environment as shown in Figure 3, where the agent (shown as the green square) has to reach the goal (shown as the yellow square) in the least amount of time steps possible. The environment's state space is a n × n matrix with real values on the interval [0, 1] to designate different features and their positions. The agent will work within an action space consisting of four actions: up, down, left, right. At each time step, the agent will take one action and move in the direction described by the action. The agent will receive a reward of +1 for moving closer to the goal and −1 for moving away or remaining the same distance from the goal.

3. Q-Learning Algorithm:

In the introduction we have already seen what q-learning is and how the q-table is updated. Here we will focus on how the q-learning algorithm works.The q-learning algorithm is described in detailed in below steps.

Step1:Initialize the q-table

Here we first build the q-table and initialize all q-values to 0. The q-table consists of 'n' rows and 'm' columns where 'n' represents number of actions an agent can take and 'm' denotes the number of states an agent can be present in.

Step 2 and 3: Choose and perform an action:

This step runs for an indefinite amount of time. This implies that this step keeps on repeating till the training loop in code ends. In the beginning our agent randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because,our agent does not know the environment and it tries to explore the environment by all kinds of things before it starts

to see the patterns. If the random number selected by the agent in the beginning(exploring) is less than epsilon, we return the random choice action space. As the agent keeps learning, it becomes more confident in estimating the q-values. At times the agent will not decide the action randomly rather will predict the reward value based on the current state and pick the action that will give the highest reward(exploiting).Our policy function describes this.

```
def policy(self, observation):
  observation=observation.astype(int)
  if np.random.random() < self.epsilon:
        return self.env.action_space.sample()
  else:
       return np.argmax(self.q_table[observation[0],observation[1]])
```

Step 4 and 5:Evaluate: Now we have taken an action and observed an outcome and reward.We need to update the function Q(s,a). We will repeat this again and again until the learning is stopped. In this way the Q-Table will be updated.

```
def update(self, state, action, reward, next_state):
    state = state.astype(int)
    next_state = next_state.astype(int)
    self.q_table[state[0],state[1]][action] += self.lr * (reward + self.gamma * np.max(self.q_table[next_state[0],next_state
    return self.q_table
```

4.Challenges:

.Firstly, it was difficult to pick the epsilon value as epsilon marks the trade-off between exploration and exploitation. At the beginning, we want epsilon to be high so that the agent take big leaps and learn things. As the agent learn about future rewards, epsilon should decay so that you can exploit the higher Q-values from the Q-table.

Secondly,The real challenge we faced while executing the project was to choose the action that could maximize the reward function by taking mininmum steps. In the beginning the agent knows nothing about the environment and so selects an action randomly. Then as it starts learning, it chooses the state-action pair that gives amximum rewards. As the agent learns through the Markov Decision process(MDP) it only know that it is given +1 reward and rest the agent have to learn on its own.So it assumes that any action-value pair that gives +1 reward is good even if it takes many steps to reach the target.

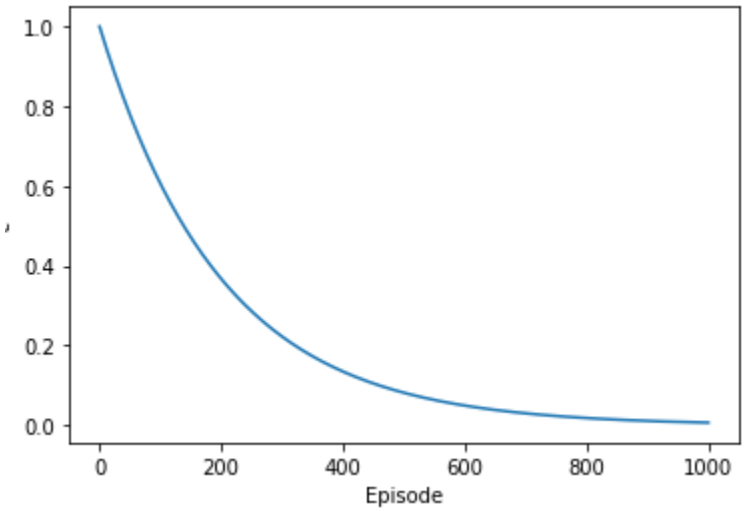
Thirdly, it was a big challenge to train the agent. The training of agent such that it takes the right actions in order to gain maximim reward is a long process as the agent have to be trained for a large number of iterations. This is an iterative process and takes a considerably long time so that the agent learns from it.

5.Results:

We tune the hyperparameters episodes and learning rate to notice the behaviour of our agent. We notice the following chnages in the learning of our agent based on changes in number of episodes and learning rate.

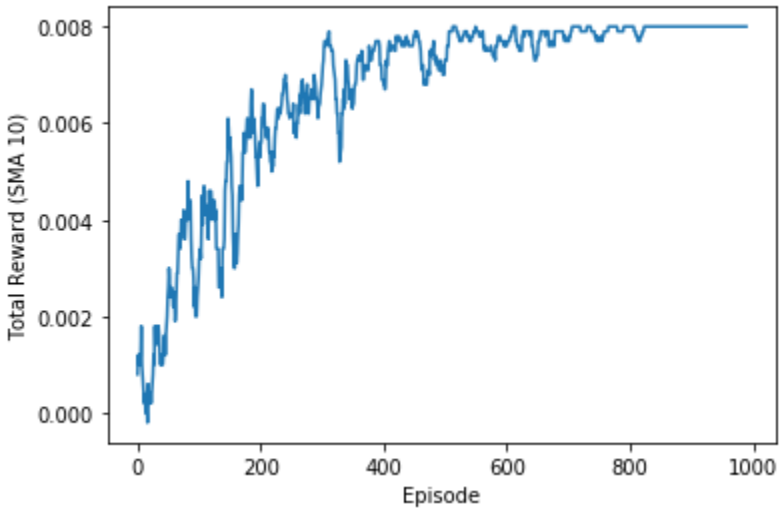
1. For default values of parameters that is number of episodes=1000 and learning rate=0.1 alon with epsilon =1.0 and discount factor(gamma)=0.9, we get the following graphs

Epsilon vs number of Episodes:



Here we see that the epsilon value decays exponentially over each episode.

Total Rewards vs number of Episodes



We can see that, as the agent learns more and more about the environment during its training for 1000 episodes, it picks up the action that maximizes the rewards. In this way, it chooses actions that give maximum rewards and after getting trained for certain number of episodes the reward function reaches its maximum and gets stabilized.

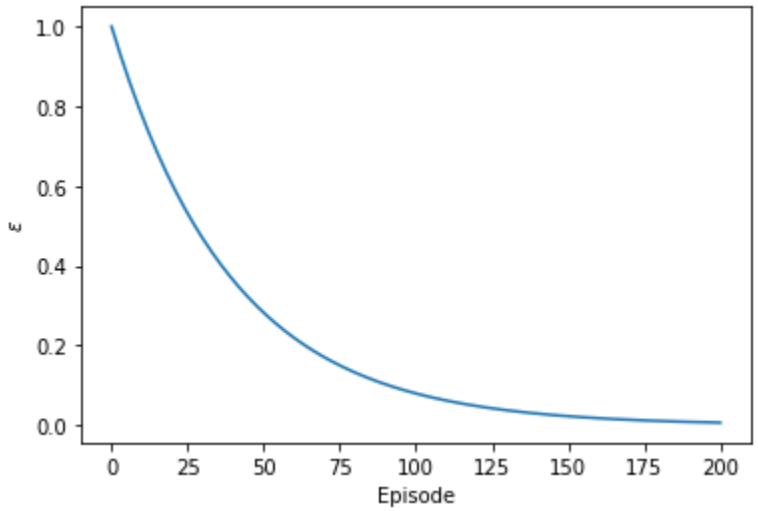
Q-table:

```
print(agent.q_table)
```

```
[[[ 5.6953279   3.84230878  4.86281811  3.53922922]
  [ 4.8447631   1.68200179  1.62816056  1.77751229]
  [ 2.80962922 -0.11803043  0.271       0.2368613 ]
  [ 0.          0.          0.1        -0.05404107]
  [ 0.1171      0.         -0.19        0.        ]]

 [[ 5.217031    3.81089564  4.64799886  3.36249964]
  [ 3.0527035   1.34859503  4.65418151  1.53847764]
  [ 4.09215202  0.40657403  0.93147593  0.88724568]
  [ 0.27775967 -0.1         0.75669559  0.        ]
  [ 0.49443295  0.         -0.19        0.        ]]

 [[ 3.58966958  3.43526816  4.68559     2.78939314]
  [ 3.48039987  2.73412755  4.0951      2.95823419]
  [ 3.439       2.34815261  2.681412    2.57015245]
  [ 2.5754763  -0.15297585  0.331192    0.06327768]
  [ 0.48017835 -0.22719733  0.          0.        ]]

 [[ 0.6686925   0.47088749  3.55671094  0.01537381]
  [ 0.74792658  0.88219769  3.40822611  0.33078134]
  [ 1.57440946  1.49660932  2.71        1.69914617]
  [ 1.9         0.81460582  1.12029349  1.19508967]
  [ 0.         -0.14678395  0.          0.43239667]]

 [[-0.19       -0.00595796  0.57447318 -0.19      ]
  [-0.0820171   0.08325258  0.92530148 -0.07982492]
  [-0.33283664  0.15304748  1.76537711 -0.091     ]
  [-0.18311289  0.54610937  1.          0.22693264]
  [ 0.          0.          0.          0.        ]]]
```

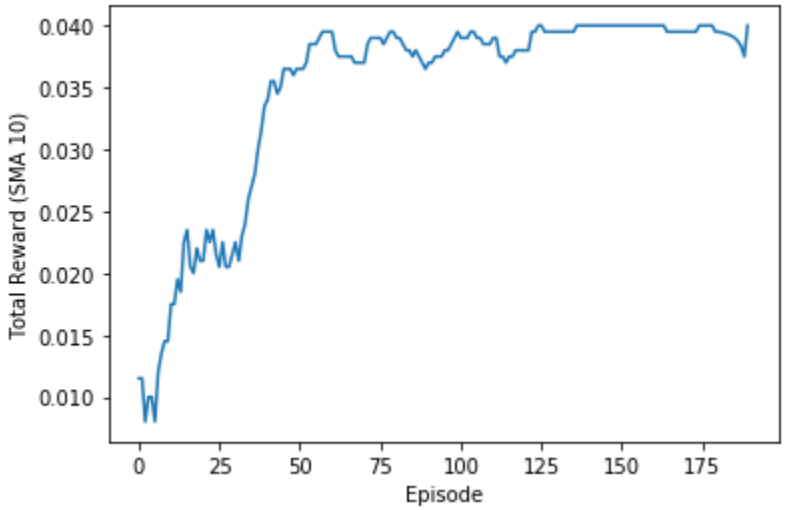
2. For values of parameters, number of episodes=200 and learning rate=0.3 along with epsilon =1.0 and discount factor(gamma)=0.9, we get the following graphs

Epsilon vs number of Episodes:



On decreasing the number of episodes , and increasing the learning rate, we can still see that the epsilon decays exponentially for each episode.

Total Rewards vs number of Episodes



After increasing the learing rate and decreasing the number of episodes, we see that our agent learns and chooses actions that maximizes the reward function upto certain number of episodes. The reward function then stabilizes and after training the agent for few more episodes with 0.3 learning rate, it starts to decrease. This means that we experience overfitting here.

Q-table:

```
print(agent.q_table)
```

```
[[[ 5.6953279   3.25962261  3.29227932  2.2210026 ]
  [ 1.46804356  0.58843272  3.10835285  0.96183768]
  [ 2.19471017 -0.06802594  2.74806108 -0.14043   ]
  [ 0.775179   -0.125121    2.51603229  0.32278746]
  [ 2.38311665  0.         -0.00835521  0.00742075]]

 [[ 1.93127256  2.19326424  5.217031    1.46208595]
  [ 2.36716924  1.47075713  4.68559     2.65226486]
  [ 4.0951      0.36036846  1.48542434  1.77566696]
  [ 0.45957    -0.219       1.88942986  0.        ]
  [ 2.18181301 -0.219       0.          0.02170991]]

 [[ 1.72235129 -0.12701499  0.831879   -0.14973241]
  [ 0.          0.03015942  3.63255391 -0.18597774]
  [ 3.439       0.79780289  1.97711376  1.28731232]
  [ 2.00432865  0.          0.45957     0.        ]
  [ 1.57759715  0.         -0.14043     0.        ]]

 [[ 1.45726292 -0.0486894   0.3         0.        ]
  [ 0.87637785  0.          0.         -0.13806266]
  [ 2.71        0.6742022   0.77792559 -0.42291   ]
  [ 1.83634645  0.          0.          0.        ]
  [ 0.83193     0.         -0.3         0.        ]]

 [[-0.3        -0.23066722  1.87451691 -0.51      ]
  [-0.3        -0.219       2.48671182  0.        ]
  [ 0.29241203  0.69532605  1.9         0.11316055]
  [-0.223137    0.25833291  1.          0.30464289]
  [ 0.          0.          0.          0.        ]]]
```

6.Conclusion:

Thus, we have successfully implemented the Q-learning algorithm and trained our agent to navigate the 4*4 world grid environment. Our agent gives best results when it is trained for following values of parameters:

Number of episodes=1000

Learning Rate=0.1

epsilon=1

Gamma(discount factor)=0.9

Our agent reaches its destination with minimum steps by choosing state-action pairs that gives maximum reward with above mentioned parameters while avoiding the obstacles in its path.

7.References

1.https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

2.https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

3.https://www.freecodecamp.org/news/a-brief-introduction-to-reinforcement-learning-7799af5840db/

4. https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/