Solving Classic 4X4 Grid World using Q Learning

Gaurav Madhav Pai
Department of Computer Science
University at Buffalo (SUNY)
Buffalo, NY 14226
gmadhavp@buffalo,edu

Abstract

In this project, we solve a 4X4 Grid World Environment from OpenAI gym, using Tabular Q learning Algorithm. An agent is designed that uses the Tabular Q Learning Algorithm to learn from the environment and develop an Optimal Policy that takes it from the source to destination in the minimum possible steps. During the training, the decision to explore the environment or exploit the already learnt policy is done using an Epsilon Greedy strategy.

1. Introduction

1.1 Challenge

The challenge is to build an agent, that navigates through the classic 4X4 grid world. The agent is initially unaware of the dynamics of the environment (model-free). It then has to learn the working of the environment by interacting with the environment and come up with a policy that will enable it to travel from source to the destination. In this project, this challenge is solved using a model free Reinforcement Learning algorithm called Tabular Q Learning.

1.2 Environment

The environment provided in this project is the Classic 4X4 grid world. As the name suggests, each state in this environment is represented by a square. The gird space (0,0) coloured yellow in Fig1 is the source from which the agent starts navigation. The agent should reach the destination grid space (4,4) coloured green by learning an optimal policy.

For each action the Agent performs inside the environment, the environment return 4 values as follows:

1. **Observation**: This represents the complete view of the grid and the position of the agent in the grid
2. **Reward**: The environment is designed in such a way the for each action taken from a state, there is reward associated which facilitates the agent to evaluate its actions and the state in which it has arrived in the grid world. The goal of the agent should be to maximise this reward.
3. **Done**: This can be considered a Boolean value. It is 1 if the agent reached the destination or 0 otherwise.
4. **Info**: This gives diagnostic information which will help in debugging (since the challenge is quite simple in this project. It has never been used)

Observation Space: Observation space is essentially a representation of the state of the environment.

Action Space: Action Space gives us the number of actions an Agent can take from each state. The actions in the Classis 4X4 Grid World are : UP, DOWN, LEFT, RIGHT

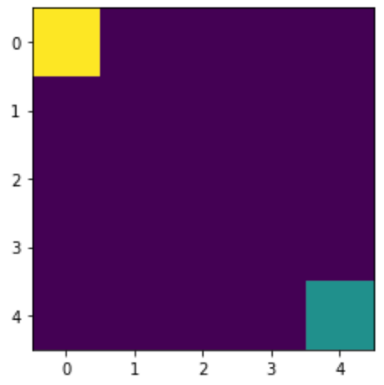


Fig1: Representation of 4X4 Grid World from OpenAI Gym

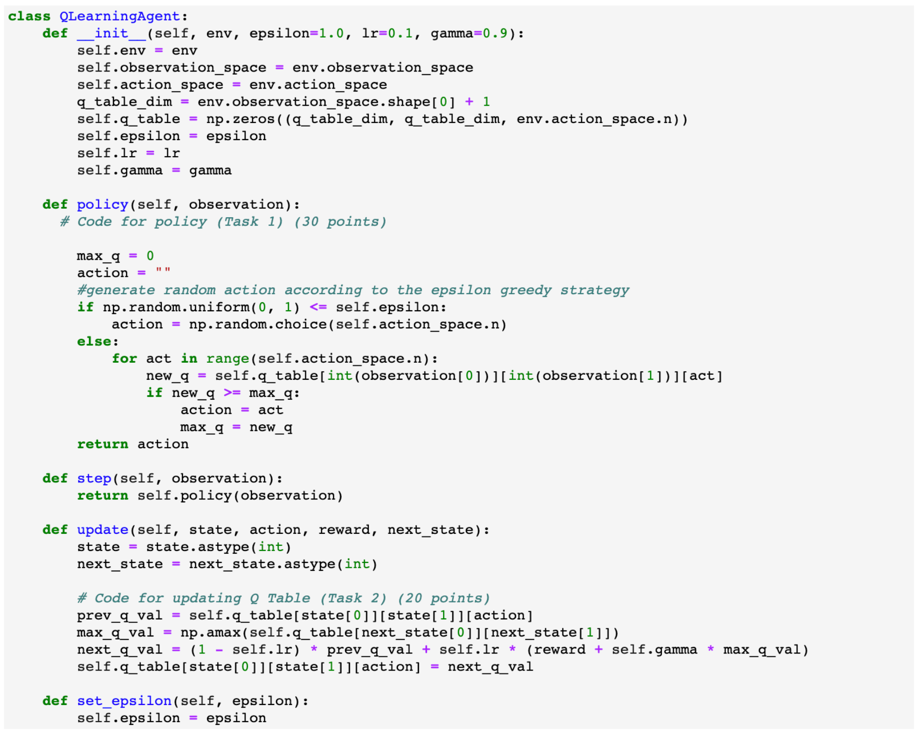
2. Architecture

2.1 Agent

An agent is the entity which navigates through the network by interacting with the environment. The main agenda of the Agent should be to maximise the Cumulative Reward obtained from the environment while achieving the Goal of the problem.

In this project, the agent has to move form source to destination with maximum reward. The reward given by the network are as follows:

1. 1 if agent moves closer to the Destination.
2. -1 if agent moves away or end up in the same position from the Destination.



```
class QLearningAgent:
    def __init__(self, env, epsilon=1.0, lr=0.1, gamma=0.9):
        self.env = env
        self.observation_space = env.observation_space
        self.action_space = env.action_space
        q_table_dim = env.observation_space.shape[0] + 1
        self.q_table = np.zeros((q_table_dim, q_table_dim, env.action_space.n))
        self.epsilon = epsilon
        self.lr = lr
        self.gamma = gamma

    def policy(self, observation):
      # Code for policy (Task 1) (30 points)

        max_q = 0
        action = ""
        #generate random action according to the epsilon greedy strategy
        if np.random.uniform(0, 1) <= self.epsilon:
            action = np.random.choice(self.action_space.n)
        else:
            for act in range(self.action_space.n):
                new_q = self.q_table[int(observation[0])][int(observation[1])][act]
                if new_q >= max_q:
                    action = act
                    max_q = new_q
        return action

    def step(self, observation):
        return self.policy(observation)

    def update(self, state, action, reward, next_state):
        state = state.astype(int)
        next_state = next_state.astype(int)

        # Code for updating Q Table (Task 2) (20 points)
        prev_q_val = self.q_table[state[0]][state[1]][action]
        max_q_val = np.amax(self.q_table[next_state[0]][next_state[1]])
        next_q_val = (1 - self.lr) * prev_q_val + self.lr * (reward + self.gamma * max_q_val)
        self.q_table[state[0]][state[1]][action] = next_q_val

    def set_epsilon(self, epsilon):
        self.epsilon = epsilon
```

Fig2: Code for Agent

The agent has to collect reward from the environment and design an policy to reach the destination with the maximum reward. Policy is the mapping from State to Actions.

2.2 Q Learning Algorithm

Q-learning is an off policy reinforcement learning algorithm that finds the best action to take given the current state. It is an off-policy because the q-learning function learns from actions that are outside the current policy, by sometimes taking random actions not dictated by the policy, therefore does not need a policy.

Q learning algorithm is coded into the Agent which decides which action to take given the state in which it is present.

The algorithm creates q-table or matrix with rows as states and columns representing the actions that can be taken from each state. It initializes the values to zero. Later, it updates and stores q-values for each episode. The q-table becomes a reference table using which the agent decides which action to take given the state.

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

Fig3: Q-learning algorithm update rule

Further, the agent interacts with the environment and makes updates to the q values in the q-table given by Q[state, action]. After each action taken by the Agent, the q-table values are updated using the rule shown n Fig3.

Learning Rate ($\boldsymbol{\alpha}$) gives the rate at which the Agent learns new information from its interactions. It gives amount by which the Q values are updated.

Immediate Reward ($\mathbf{r_t}$) is the reward given by the environment at time-step t after completing an action from a state in which the agent was present at time-step t.

Discount Factor ($\boldsymbol{\gamma}$) is to balance immediate and future reward. From our update rule above you can see that we apply the discount to the future reward. Typically this value can range anywhere from 0.8 to 0.99.

While taking action in the environment, the Agent has two options. The first is to use the q-table as a reference and chooses an action when it is in a particular state based on the maximum value of the actions (Q-value) for that state.. This is known as Exploitation. The second way to take action is to act randomly. This is called Exploration. Instead of using the q-table to decide the action the Agent selects an action at random. Acting randomly is extremely essential as it allows the agent to explore the environment and discover new states that were not discovered before and use that information to update the q-table.

The Agent handles the Exploration vs Exploitation decisions using an Epsilon Greedy Strategy dictated by the Epsilon(ϵ) value of the Agent. In this strategy the Agent generates a random number and if it is less than or equal to ϵ value, it Explores the environment, otherwise it Exploits by taking action by referencing the q-table. This Epsilon value is decreased little by little as the training progresses because as the Agent learns better and better q-values the need for Exploration is decreased.

The updates occur after each step or action and ends when an episode terminates. A terminal is when the agent reaches its goal. After iterating through some amount of episodes and updating the q-table values, the Agent eventually converges and learns the optimal q-values or q-star (Q∗). Using this Optimal Policy the agent can then navigate from source to destination in the 4X4 Grid World with the maximum reward. In this project, the Agent is trained using the Q-Learning Algorithm for 100 Episodes. But it converges to the Optimal Policy at around 60th episode.

The policy and update functions in the QLearningAgent class shown in Fig2 is the implementation of the Q-Learning Algorithm in the Agent.

3. Results

3.1 Decay of Epsilon (ϵ)

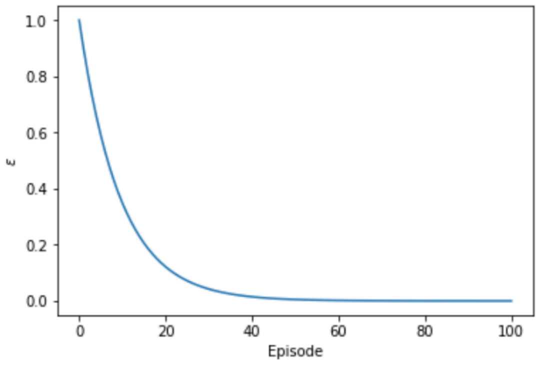


Fig 4: Epsilon Decay for the Greedy Epsilon Strategy

3.2 Total Rewards over the course of 100 episodes of Training

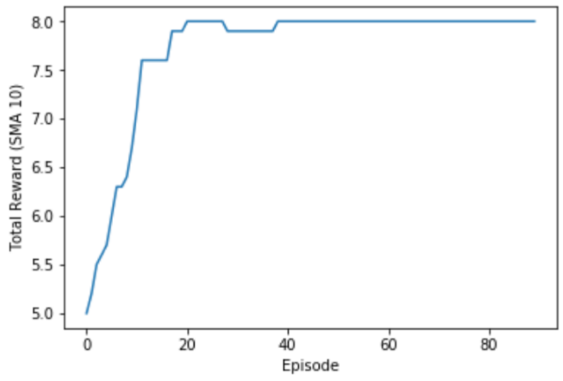


Fig 5: Total Reward vs Episodes

4. Conclusion

By looking at the graph of Total Rewards vs Episodes shown in Fig5 it can be concluded the the environment is successfully solved using the Q-Learning Algorithm. It can be said so because the graph shows that at the end of the training, the agent achieves a total reward of 8 per episode. By considering the Reward Dynamics of the environment, 8 is the minimum number of steps required to reach the destination from the source and hence the Q-learning Algorithm has converged to give Optimal Policy.