CSE 574 – Implementing Q-Learning (Reinforcement Learning).

Rohit Lalchand Vishwakarma
Department of Computer Science University at Buffalo
– State University of New York
Buffalo, NY 14214 *rohitlal@buffalo.edu*

Abstract

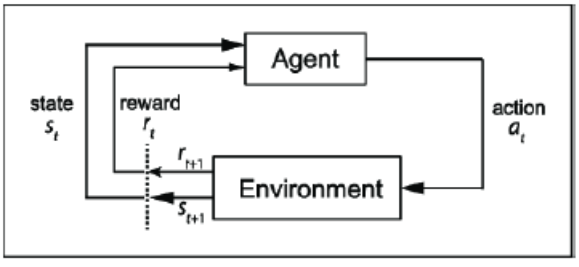
The goal of this project is to simulate a reinforcement learning agent to travel in the classic 4X4 grid-world environment. The agent will learn an optimal policy through Q-Learning which will help it to take actions to reach the goal while avoiding obstacles. Here, to simplify the task we are providing a working environment as well as the framework to the agent which is built to be compatible with OpenAI Gym environments.

1 Introduction

1.1 What is Reinforcement Learning and how we are using it?

Reinforcement Learning, known as a semi-supervised learning model in machine learning, is a technique to allow an agent to take actions and interact with an environment to maximize the total rewards.

This project represents the building of this reinforcement learning agent using Q-learning algorithm to navigate the classic 4X4 grid-world environment where the agent will learn through each action(step) it takes which will award it with rewards for that action and accordingly it will reach the end goal by maximizing the rewards which is last block of the grid i.e. is the last row and column of the 4X4 grid.



2 Dataset Definition

Our dataset consists of the entities which are environment, action, rewards and state.

2.1 Environment

All agent in the reinforcement learning model has an environment where they interact and accordingly performs the next actions. In our project we are using 4X4 Grid as an environment. Figure 2.1 represents the agent (Yellow block) in its environment 4X4 Grid (Purple square area) and the goal destination (Green block).

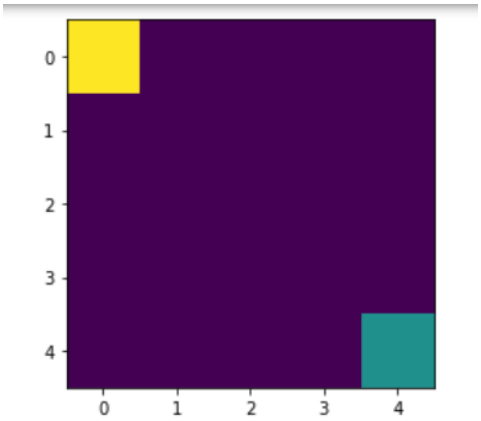


Fig 2.1 – 4X4 Grid Environment

2.2 Action

Action is what agent does in the environment which causes it to change its current state. In our model it is moving left, right, up, or down.

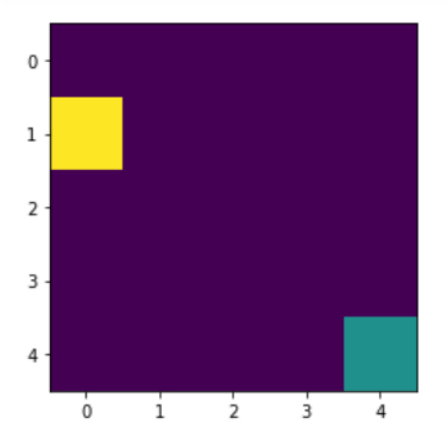


Fig 2.2 – State after first action of agent (down)

2.3 Reward

Reward is what agent is awarded with if it makes a move that takes him closer to our goal. So, in our case we are awarding the agent with 1 if it makes the move closer to the goal otherwise -1. As said the goal is to maximize the rewards so next time the agent will follow that path which maximizes its rewards.

2.4 State

State is the current place where the agent is. So, when our agent started it was at (0,0) i.e. 0^{th} row and 0^{th} column it is its first state. Then, after first action its state changes to (1,0) i.e. the 1^{st} row and 0^{th} column. These states are in correspondence that it minimizes the distance between goal and its current state.

3 Architecture

3.1 Markov Decision Processes

Reinforcement learning relies on Markov Decision Processes (MDP's) which consist of states (st) that an agent is in and actions (at) that the agent can take to move onto a new state (st+1) and receive a reward or a punishment (Rt+1). Q-learning seeks to learn the corresponding Q-value (think "quality") of a given action-state pair. When the number of states and actions are both small and well-defined, we can represent these states and actions in a table. However, when there is a situation of huge number of states and actions to be taken the challenge arises.

3.2 Q-Learning Algorithm

3.2.1 What is Q-Learning?

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

3.2.2 What is Q?

The 'q' in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.

3.3 Creating a Q-table

When q-learning is performed, a q-table or matrix that follows the shape of [state, action] is formed. In our project, we have "epsilon" which is an exploration rate. So, here our agent will first randomly select its action at first by certain percentage. Here, it will learn randomly on its own until it finds a pattern. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward.

$$\pi(s_t) = \underset{a \in A}{\operatorname{argmax}}\, Q_\theta(s_t, a)$$

```
if np.random.uniform(0,1)<=self.epsilon:
    action=np.random.choice(self.action_space.n)
else:
    action = np.argmax(self.q_table[observation[0]][observation[1]])
return action
```

3.4 Q-learning and making updates

The next step is simply for the agent to interact with the environment and make updates to the state action pairs in our q-table Q [state, action]. We will update our Q-table according to the agent learning via its actions and getting rewards at each action.

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

The following code snippet shows the update of Q-Table according to the rewards it gets at each action (step).

```
while not done:
    action = agent.step(curr_state)
    obs, reward, done, info = env.step(action)
    next_state = obs.copy()
    agent.update(curr_state, action, reward, next_state)
    curr_state = next_state.copy()
    rewards += reward
```

The updates occur after each step or action and ends when an episode is done. “Done” in this case means reaching some terminal point by the agent. A terminal state for example can be anything like landing on a checkout page, reaching the end of some game, completing some desired objective, etc. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal q-values.
In the update above there are a couple of variables that we haven’t mentioned yet. What’s happening here is we adjust our q-values based on the difference between the discounted new values and the old values. We discount the new values using gamma and we adjust our step size using learning rate.
Learning Rate (lr): learning rate, often referred to as alpha or, can simply be defined as how much you accept the new value vs the old value. Above we are taking the difference between new and old and then multiplying that value by the learning rate. This value then gets added to our previous q-value which essentially moves it in the direction of our latest update.
Gamma : gamma or is a discount factor. It’s used 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.

We also have a decay_rate which will decay the rate of **epsilon** meaning the exploring rate of the agent. A decay is added so that the rate of exploration is decreased as the agent gets better at reaching the target cell.

4 Results

When we keep our hyperparameters as learning rate as 0.1, gamma =0.9, Decay_rate =0.90 with number of episodes 50 we get the following result. Below Fig 4.1 Epsilon decay vs episode states the decay in epsilon rate with increasing number of episodes. So as the number of episodes increases the exploration rate decreases. Moreover, in graph Fig 4.2 Rewards vs Episodes gives us the information about the rewards agent is getting when the episode is increasing.

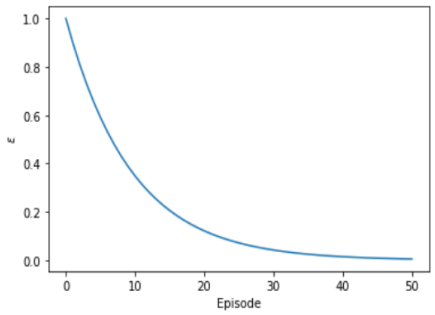


Fig 4.1 Epsilon decay vs Episodes

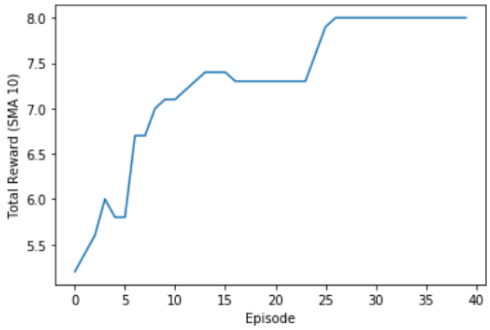


Fig 4.2 Rewards vs Episodes

After changing the Hyper Parameters keeping Learning rate and gamma same, decay_rate =0.92 and Number of Episodes = 100. We get the following result.

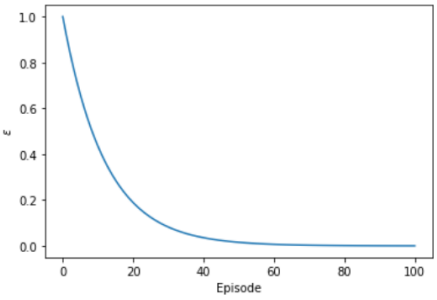


Fig 4.3 Epsilon decay vs Episodes

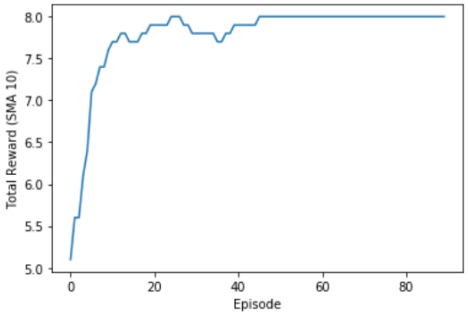


Fig 4.4 Rewards vs Episodes

After changing the Hyper Parameters keeping Learning rate and gamma same, decay_rate =0.95 and Number of Episodes = 500. We get the following result.

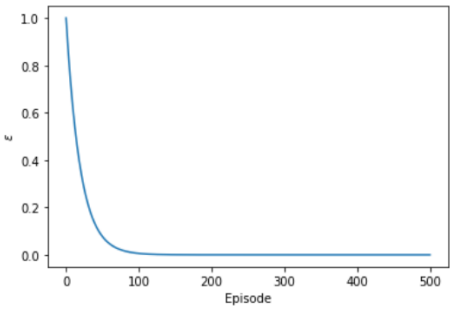


Fig 4.5 Epsilon decay vs Episodes

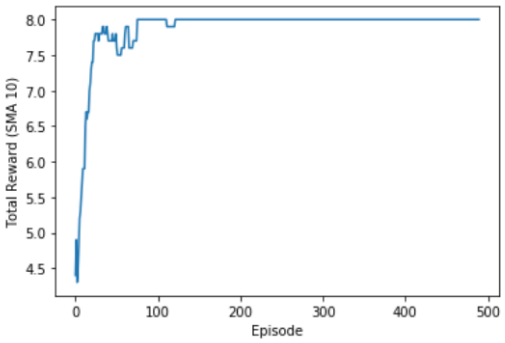
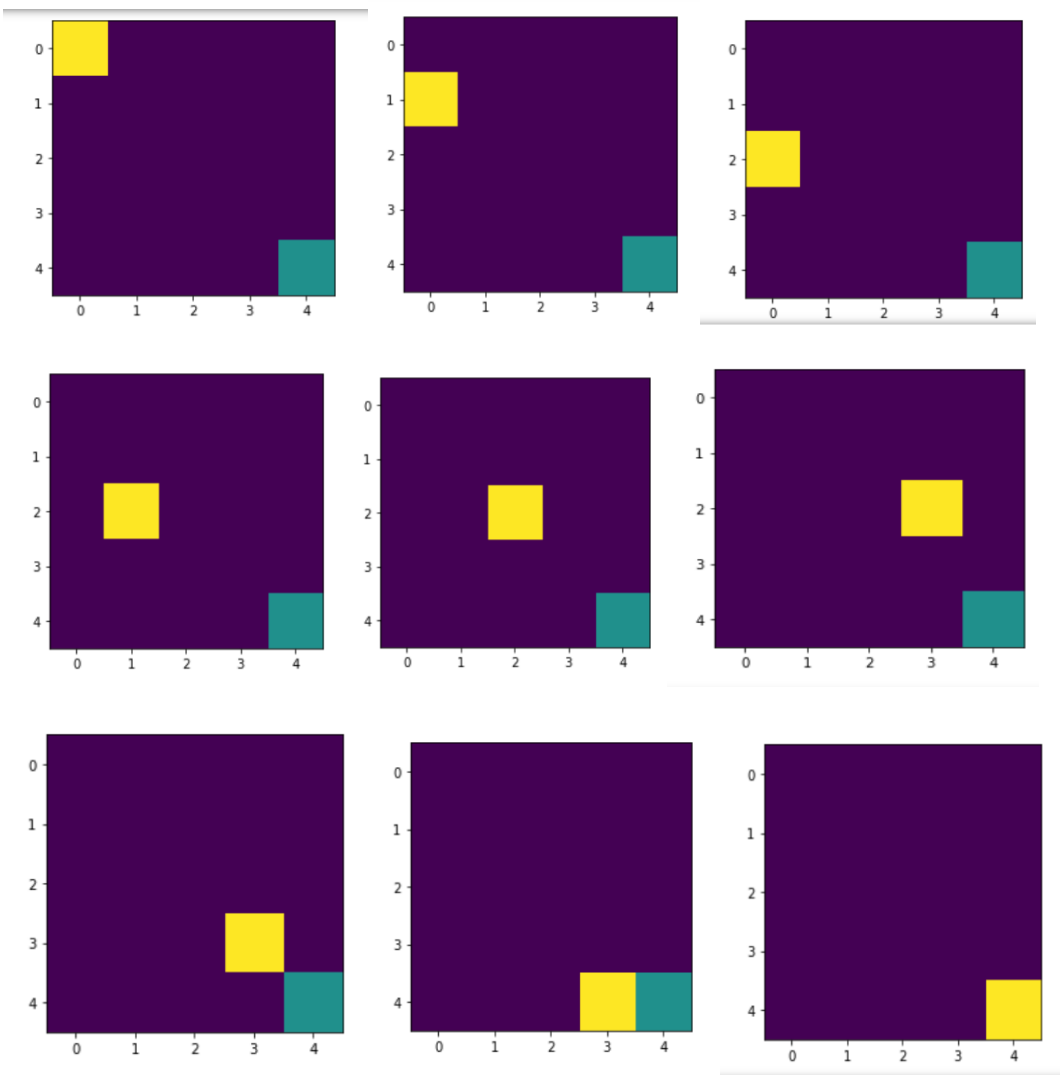


Fig 4.6 Rewards vs Episodes

Fig 4.5 states us that after a number of episodes there is no exploration and agent update the Q value according to what it has learnt through these episodes and had not updated it randomly.

Fig 4.6 specifies that after episode ~ 45 agent has learnt fully and hence the rewards stays at the maximum as there is no scope of increasing the rewards

Below figure shows the final navigation of agent in the environment based on its learning from Q-learning algorithm.



6 Conclusion

Our project has implemented Q-learning policy on a provided 4X4 grid world environment. Significant improvements were observed over the random and heuristic agents. The agent being greedy and trying to maximize the rewards in each episode converged better than the former two agents. The agent ran through 500 episodes with the exploration rate of 0.95, learning rate of 0.1 and gamma of 0.9. Tuning was employed to check if the model could perform better with changes in the epsilon rates and number of episodes. It could be seen that the agent was able to converge better with an epsilon value is 0.95, number of episodes is more (500). Further improvements can be done by changing the learning rate from 0.001 and gamma value and modifying the epsilon value. We learned that q-learning uses future rewards to influence the current action given a state and therefore helps the agent select best actions that maximize total reward.

7 References

1] Project_description.pdf (Document for Project 4)

2] https://www.datahubbs.com/intro-to-q-learning/

3] https://towardsdatascience.com/simple-reinforcement-learning-q-learning