Grid Search in Q-Learning

Term Project- EEN-680

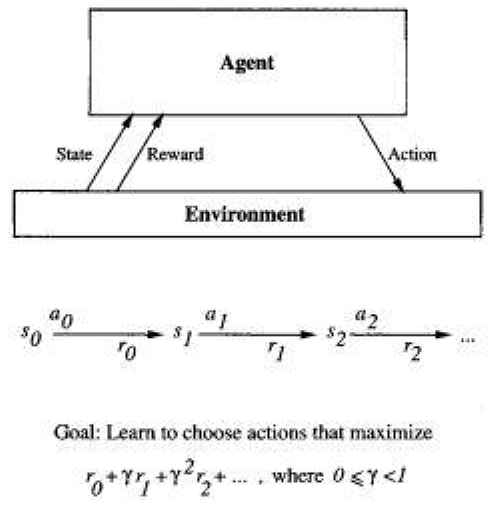
-Purva Chiniya

1. Introduction

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them.

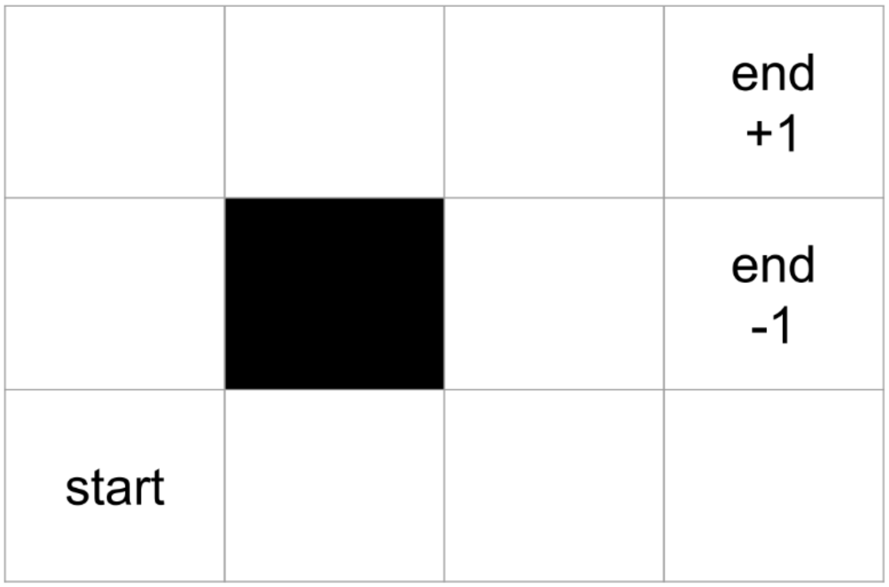
The goals of the agent can be defined by a reward function that assigns a numerical value-an immediate payoff-to each distinct action the agent may take from each distinct state.

The task of the agent is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.


Agent
State
Reward
Action
Environment
$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$
Goal: Learn to choose actions that maximize
$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$, where $0 \leq \gamma < 1$

2. Grid Search Problem

Given a grid of m rows n columns , a start state and an end state . We need to find the optimal policy to maximize the rewards.The rule is simple. Your agent/robot starts at the left-bottom corner(the 'start' sign) and ends at either +1 or -1 which is the corresponding reward. At each step, the agent has 4 possible actions including up, down, left and right, whereas the black block is a wall where your agent won't be able to penetrate through. In order to make it more straightforward, our first implementation **assumes that each action is deterministic**, that is, the agent will go where it intends to go. For instance, when the agent decides to take action up at (2, 0), it will land in (1, 0) rather than (2, 1) or elsewhere. (We will add uncertainty in out second implementation) However, it the agents hit the wall, it will remain at the same position.



2.1 Deterministic Action

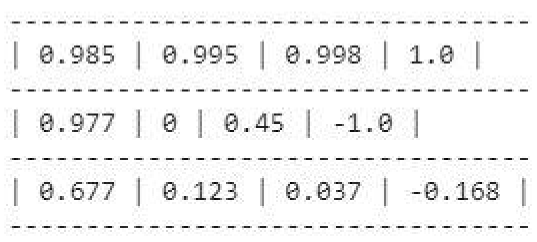
In the deterministic case what our agent will finally learn is a policy, and a policy is a mapping from state to action, simply instructing what the agent should do at each state. In our case, instead of learning a mapping from state to action, we will leverage value iteration to firstly learn a mapping of state to value(which is the estimated reward) and based on the estimation, at each state, our agent will choose the best action that gives the highest estimated reward.

2.2 Non Deterministic Case

Non-deterministic means that the agent will not be able to go where it intends to go. When it takes an action, it will has a probability to crash in a different action.
Whereas V(s) is a mapping from state to estimated value of that state, the Q function — Q(s, a) is only one component different from V function. Instead of thinking that you receive a value when you are in a specific state, think one step forward, you are in a state and by taking a specific action you receive a corresponding value. In essence, both functions have no difference, but by binding state with action together convenient our life. For instance, recall the result of grid world using value iteration, we get a estimated value for each state, but to have our policy $\pi(s, a)$, which is a mapping from state to action, we need to go one step further by choosing the action that could get into the maximum value of next state. However, in Q-function, state and action are paired in the first place, which means when one has the optimal Q-function, he has the optimal action of that state.

3. Results

3.1 Deterministic Case

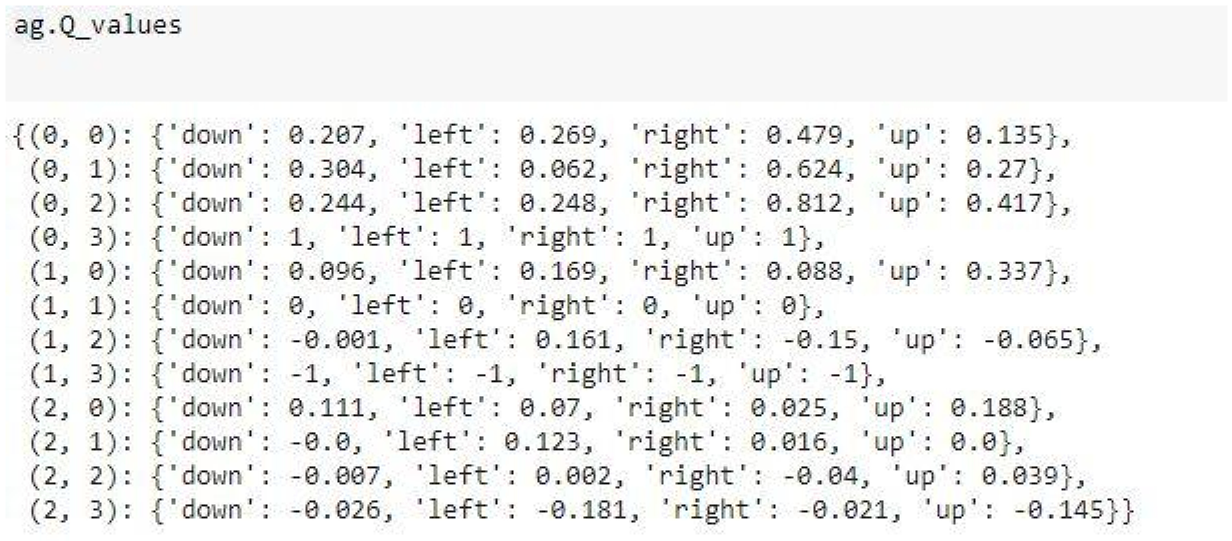


```
---------------------------------
| 0.985 | 0.995 | 0.998 | 1.0 |
---------------------------------
| 0.977 | 0 | 0.45 | -1.0 |
---------------------------------
| 0.677 | 0.123 | 0.037 | -0.168 |
---------------------------------
```

Our optimal sequence would be : (2,0) ->(1,0) -> (0,0) -> (0,1) -> (0,2) -> (0,3)

That is up -> up -> right -> right -> right .

3.2 Non Deterministic Case



```
ag.Q_values
```

```
{(0, 0): {'down': 0.207, 'left': 0.269, 'right': 0.479, 'up': 0.135},
 (0, 1): {'down': 0.304, 'left': 0.062, 'right': 0.624, 'up': 0.27},
 (0, 2): {'down': 0.244, 'left': 0.248, 'right': 0.812, 'up': 0.417},
 (0, 3): {'down': 1, 'left': 1, 'right': 1, 'up': 1},
 (1, 0): {'down': 0.096, 'left': 0.169, 'right': 0.088, 'up': 0.337},
 (1, 1): {'down': 0, 'left': 0, 'right': 0, 'up': 0},
 (1, 2): {'down': -0.001, 'left': 0.161, 'right': -0.15, 'up': -0.065},
 (1, 3): {'down': -1, 'left': -1, 'right': -1, 'up': -1},
 (2, 0): {'down': 0.111, 'left': 0.07, 'right': 0.025, 'up': 0.188},
 (2, 1): {'down': -0.0, 'left': 0.123, 'right': 0.016, 'up': 0.0},
 (2, 2): {'down': -0.007, 'left': 0.002, 'right': -0.04, 'up': 0.039},
 (2, 3): {'down': -0.026, 'left': -0.181, 'right': -0.021, 'up': -0.145}}
```

Our optimal sequence would be : (2,0) ->(1,0) -> (0,0) -> (0,1) -> (0,2) -> (0,3)

That is up -> up -> right -> right -> right .