Recitation: HW8
Reinforcement Learning

10-301/601: Introduction to Machine Learning

11/15/2021

1 MDPs and the Bellman Equations

A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma, s_0)$, where:

1. $\mathcal{S}$ is the set of states
2. $\mathcal{A}$ is the set of actions
3. $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ is the transition function
4. $R : \mathcal{S} \times A \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function
5. $\gamma \in [0, 1)$ is the discount factor
6. s_0 is the start state

Playing a game

When we play a game, we can model the game using a Markov decision process as follows:

1. We start in some state s_0
2. choose some action $a_0 \in \mathcal{A}$
3. As a result of our choice, the state of the game transitions to some successor state s_1. For non-deterministic transition, we draw the next state according to our transition function. That is to say,
$$P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t) = T(s_t, a_t, s_{t+1})$$
4. Then we pick another action a_1, and so on.

We can represent it as

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Sequence of states and payoff

Let $r_t = R(s_t, a_t, s_{t+1})$. Upon visiting a sequence of states $s_0, s_1, s_2, s_3 \dots$ with respective actions $a_0, a_1, a_2, \dots$ the **total discounted payoff** is given by

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

Note that different sequences of states may have a different payoff value.

Goal of reinforcement learning

Choose actions over time so as to **maximize** the expected value of the total discounted payoff

$$\mathbb{E}\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots\right]$$

Things to notice:

- The reward at timestep t is discounted by a factor of γ^t. As $t \to \infty$ we have $\gamma^t \to 0$.
- To make this expectation large (max), we need to **accrue positive rewards as soon as possible** and postpone negative rewards as long as possible.
- We take the expectation of the total payoff, as to average its value over different sequences of states. Remember that different sequences of states have a different payoff value.

What about the policy function, value function and optimal value function?

- A **policy** is any function $\pi : S \mapsto A$ mapping from the states to the actions.
- We say that we are executing some policy π if, whenever we are in state s, we take action $a = \pi(s)$
- For the optimal policy function π^* we can compute its **value function** as:

  $$V^{\pi^*}(s) = V^*(s) = \mathbb{E}\left[R(s_0, \pi^*(s_0), s_1) + \gamma R(s_1, \pi^*(s_1), s_2) + \gamma^2 R(s_2, \pi^*(s_2), s_3) + \ldots \mid s_0 = s, \pi^*\right]$$

  In other words, this is the best possible expected total payoff that can be attained using **any policy** π.

Why Bellman equations?

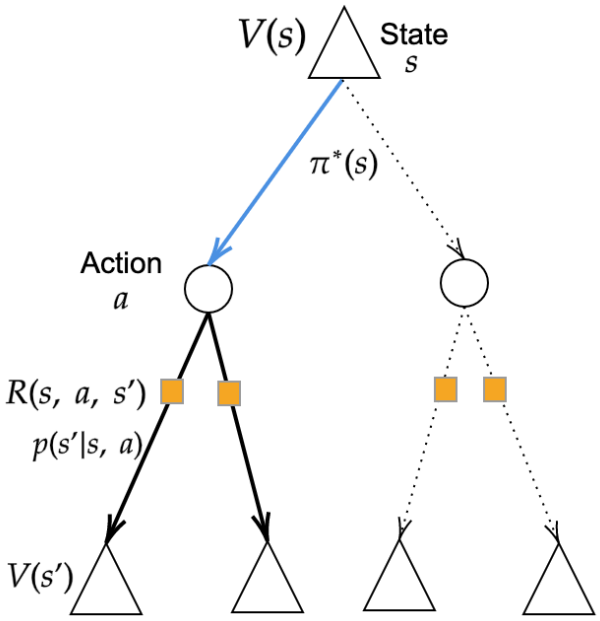


Figure 1: Visualization of Bellman Equation for $V(s)$

The **optimal value function** can be represented using Bellman's equations (named **Bellman optimality equation** for state value function), i.e.,

$$V^*(s) = \max_a \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V^*(s')).$$

If $R(s,a,s') = R(s,a)$ then we have

$$V^*(s) = \max_a (R(s,a) + \sum_{s' \in \mathcal{S}} p(s'|s,a)\gamma V^*(s')).$$

This equation can be understood easily using backtracking in Fig.1:

- Going back from leaf $V(s')$ to circle node a. Each $V(s')$ must be (i) discounted by γ (ii) added to immediate reward $r = R(s,a,s') = R(s,a)$ and (iii) multiplied by transition probability, $p(s'|s,a)$;
- All leaf contribution, after previous step, aggregate at circle node a, i.e., the sum over states s' in the equation above;
- Finally, node $V(s)$ takes the max over its subtrees based on the results from previous step.

The interpretation of Bellman equation is also very intuitive: how valuable is the current state under a policy? Well, we take a action $a = \pi^*(s)$ under the policy, then transition to a

new state s', and get the reward $R(s, a, s')$. Since the transition can be stochastic ($p(s'|s, a)$) and the value of the new state s' is only realized "in the future", we discount this "future value" and take its expectation under transition probabilities.

Bellman Equations with Q-Values

Can we define a value function that has the action as an input as well? Yes!
We define a Q function $Q(s, a)$ that returns the expected discounted future value of taking action a at state s.

Question: What are the Bellman equation (and optimality equation) for action value function $Q(s, a)$? Use the visualization in Fig.2 to help your derivation.

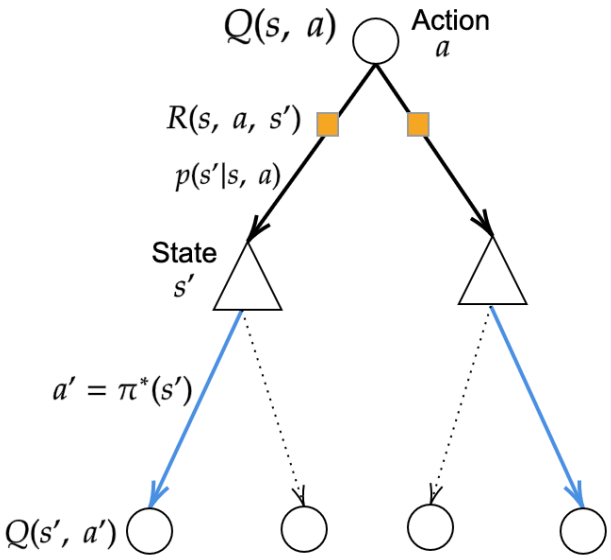


Figure 2: Visualization of Bellman Equation for Action Value Function, $Q(s, a)$

Answer:

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a)(R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')).$$

In the event that $R(s, a, s') = R(s, a)$, we can write this as:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) \max_{a' \in \mathcal{A}} Q^*(s', a')$$

Optimal policy

With the considerations above, we define the optimal policy $\pi^* : S \mapsto A$ as

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V^*(s')) = \arg\max_{a \in \mathcal{A}} Q^*(s,a)$$

Note that $\pi^*(s)$ is a function from states to actions. In particular, the function $\pi^*(s)$ is the same for all starting states.

Question: How to solve the finite-state MDP?

Answer: Value iteration, policy iteration.

2 Value Iteration

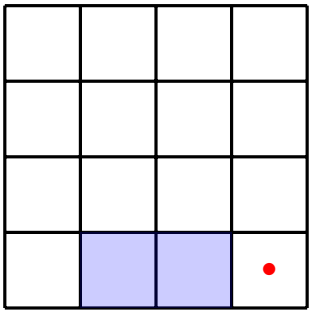


Figure 3: The cliff-walking environment

Here, we assume a 4x4 grid environment. The reward for reaching all cells is 0, except for the cell with the red dot, which has a reward of 1, and the shaded cells, which have a reward of -1. The episode ends if the agent lands in either the shaded cells or the red-dot cell. The state space (S) is simply all the cells. The action space (A) is up, down, left or right. Our transition function is deterministic. If the agent tries to move out of the grid, it simply goes back to its previous state. The discount factor, γ, is 0.9.

Bellman optimality equation for state value function:

$$V^*(s) = \max_a \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V^*(s'))$$

We can numerically approximate V^* using synchronous value iteration. That is, we use the recurrence relation defined as follows: $\forall s \in \mathcal{S}$

$$V_0(s) = 0$$
$$V_{k+1}(s) = \max_a \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V_k(s'))$$

1. Using value iteration, find the updated value of each cell for iterations 2 to 4.
   k=1

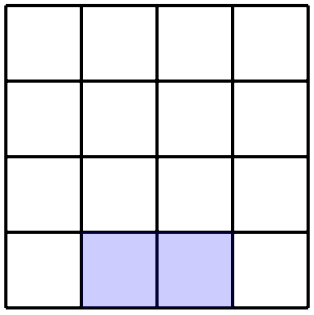
0	0	0	0
0	0	0	0
0	0	0	1
0	0	0	0

   k=2

   k=3

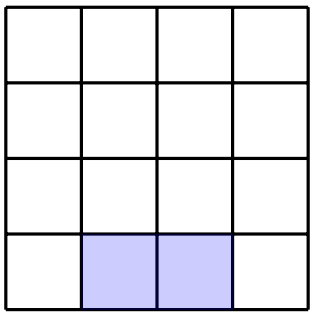
   k=4

2. What is an optimal policy after you run value iteration to convergence?

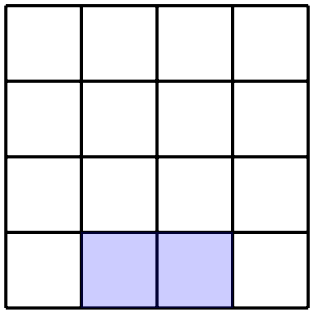


3. Now suppose that the environment is sloped downward towards the cliff, with all the other settings unchanged. For every action taken, there is a 0.5 probability that the agent will move as intended and a 0.5 probability that the agent will slip and move 1 cell down instead.

   k=1



   k=2



4. Below is the result after 100 iterations. What is the optimal policy now?

0.14	0.22	0.54	0.81
0.09	-0.03	0.38	0.9
0.07	-0.47	-0.05	1
0.07			

5. How could one change the environment setting to result in the optimal policy below?

→	→	→	→
→	→	→	→
↑	↑	↑	→
←			

◯ The agent receives a reward of -1 when it reaches any cell except the red-dot cell

◯ The agent receives a reward of 1 when it reaches any non-shaded cell.

◯ The discount factor is now 0.1 instead of 0.9.

◯ Instead of slipping downwards, the agent now has a 0.5 probability of slipping to the right side of its intended direction.

◯ None of the above.

3 Q-learning

In this section, we will be going over the Mountain Car environment from the homework.

In Mountain Car you control a car that starts at the bottom of a valley. Your goal is to reach the flag at the top right, as seen in Figure 4. However, your car is under-powered and can not climb up the hill by itself. Instead you must learn to leverage gravity and momentum to make your way to the flag. It would also be good to get to this flag as fast as possible.

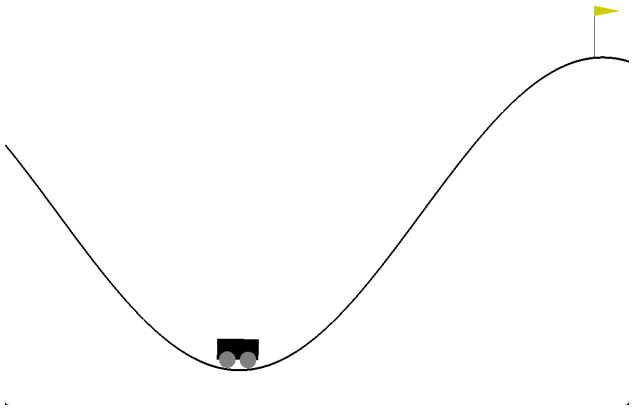


Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

The state of the environment is represented by two variables, `position` and `velocity`. `position` can be between -1.2 and 0.6 inclusive and `velocity` can be between -0.07 and 0.07 inclusive. These are just measurements along the horizontal axis.

The actions that you may take at any state are $\{0, 1, 2\}$ which respectively correspond to (0) pushing the car left, (1) doing nothing, and (2) pushing the car right.

1. To solve Mountain car environment, would you use value iteration or Q-learning? Why?

2. Approximate Q-learning involves using a function that estimates q-values given state and action pairs. A linear approximation function could take on the form :

$$Q(s, a; \mathbf{w}) = \mathbf{w}^T \mathbf{s} + b$$

where $\mathbf{s} = [\texttt{position}, \texttt{velocity}]^T$. $\mathbf{w} \in \mathbb{R}^2$ and b are the parameters to be learned.

Instead, let us consider a different linear approximation function:

$$Q(\mathbf{s}, a; \mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2) = \mathbf{w}_a^T \mathbf{s} + b = \begin{cases} \mathbf{w}_0^T \mathbf{s} + b & \text{if } a = 0 \\ \mathbf{w}_1^T \mathbf{s} + b & \text{if } a = 1 \\ \mathbf{w}_2^T \mathbf{s} + b & \text{if } a = 2 \end{cases}$$

Let $\mathbf{s} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$, $\mathbf{w}_0 = \begin{bmatrix} -1 \\ -10 \end{bmatrix}$, $\mathbf{w}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\mathbf{w}_2 = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$, $b = 0.5$

What are the q values for all actions performed at state $\mathbf{s}$?

3. What is an example of non-linear approximations that we could use in approximate Q-learning?

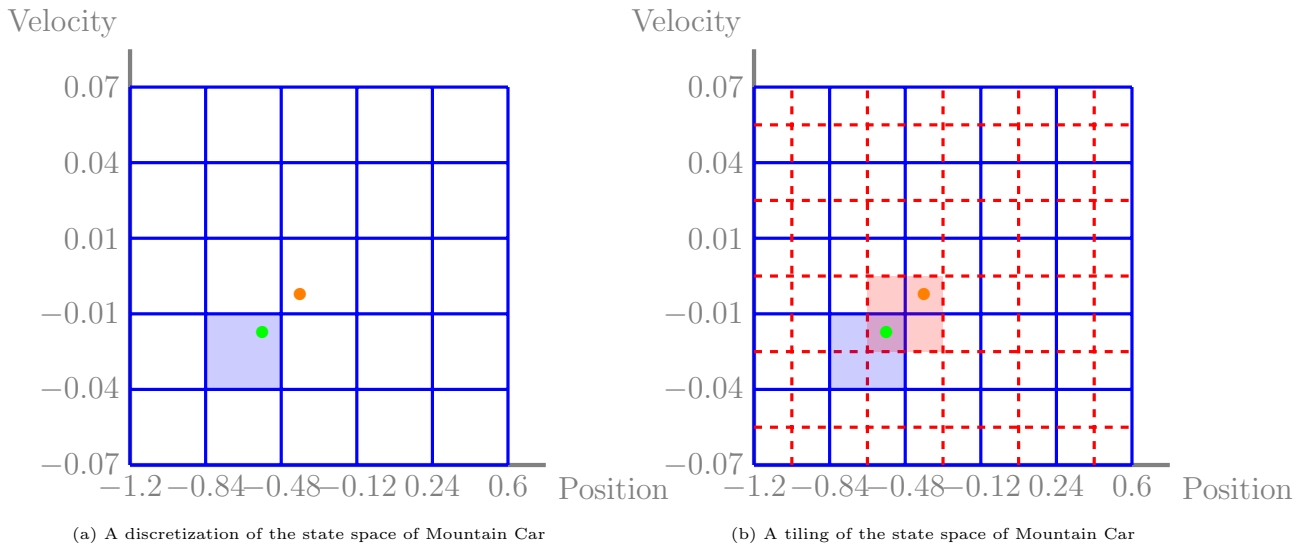
4. If we solve the mountain car environment using Q-learning, we need to define a state representation. Consider the following state representation with two real-valued features.

$$\mathbf{s} = \begin{bmatrix} \texttt{position} \\ \texttt{velocity} \end{bmatrix}$$

If we have to solve using Q-learning with linear approximation, do you see any disadvantages with the above state representation?

5. For the Mountain Car environment, we know that `position` and `velocity` are both bounded. What we can do is draw a grid over the possible `position-velocity` combinations as seen in Figure 5a. We then enumerate the grid from bottom left to top right, row by row. Then we map all states that fall into a grid square with the corresponding one-hot encoding of the grid number. For efficiency reasons we will just use the index that is non-zero. For example the green point would be mapped to $\{6\}$. This is called a *discretization* of the state space. **Is there any downside to this approach?**

6. In addition to 2D discretization, we can draw two grids over the state space, each offset slightly from each other as in Figure 5b. The grids are indexed in row major order left to right and then bottom to top, then blue grid to red grid. For example, the top right blue grid is 24 while the bottom left red grid is 25. The new state is a "two-hot" encoding marking which blue and red grid the point falls within. **What are the two indices, one for each grid that map the green point?**



(a) A discretization of the state space of Mountain Car

(b) A tiling of the state space of Mountain Car

Figure 5: State representations for the states of Mountain Car

7. Suppose we had decreased the size of the grid squares in Figure (a) to match the precision of the tiling in Figure (b). Compare the dimensionality of states under this new grid scheme versus the tiling scheme in Figure (b). What is the advantage of tiling over this single grid discretization?

4 Code Signature for HW6

We recommend that you follow the given code signature for HW6.

```
class LinearModel:
    def __init__(self, state_size: int, action_size: int,
                 lr: float, indices: bool):
        """indices is True if indices are used as input for one-hot features.
            Otherwise, use the sparse representation of state as features
        """

    def predict(self, state: Dict[int, int]) -> List[float]:
        """
        Given state, makes predictions.
        """

    def update(self, state: Dict[int, int], action: int, target: int):
        """
        Given state, action, and target, update weights.
        """

class QLearningAgent:
```

```
    def __init__(self, env: MountainCar, mode: str = None, gamma: float = 0.9,
                 lr: float = 0.01, epsilon:float = 0.05):

    def get_action(self, state: Dict[int, int]) -> int:
        """epsilon-greedy strategy.
        Given state, returns action.
        """

    def train(self, episodes: int, max_iterations: int) -> List[float]:
        """training function.
        Train for 'episodes' iterations, where at most 'max_iterations' iterations
        should be run for each episode. Returns a list of returns.
        """

if __name__ == "main":
    # run parser and get parameters values
    env = MountainCar(mode=mode)
    agent = QLearningAgent(env, mode=mode, gamma=gamma, epsilon=epsilon, lr=lr)
    returns = agent.train(episodes, iterations)
```