University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 24: November 15

Lecturer: Alexander Schwing Scribe: Ruichuan Zhang

1 Overview

This scribe is about Lecture 25. In this lecture, we (1) started from reinforcement learning based on Markov Decision Process (MDP) models, which were covered in last lecture (Lecture 24); and (2) discussed reinforcement learning based on MDPs without transition probabilities, and reinforcement learning without MDPs. The main goals of this scribe are:

- Recap: reinforcement learning based on MDP models
  - What differentiates reinforcement learning from supervised learning?
  - What are the ingredients of a MDP model?
  - How to use exhaustive search to find the optimal policy? What is policy evaluation?
  - What are policy iteration and value iteration? What are $V^*(s)$ and $Q^*(s, a)$? What is the relation between the two values?
- Q-Learning
  - What to do when the MDP model is given without the transition probabilities?
  - What to do when the MDP model is unknown?

2 Recap: Reinforcement Learning Based on MDP Models

2.1 Reinforcement learning

Reinforcement learning is different from supervised learning in three ways:

- It has no supervision, only rewards
- It has delayed reward, and
- Actions has an impact on the data.

2.2 MDP

We use MDP to model the reinforcement learning environment. We can think of a MDP model as a finite state machine and represent it as a graph. A MDP model has four ingredients:

- A set of states $s \in S$
- A set of actions $a \in A_s$
- Transition probabilities $P(s'|s, a)$
- Reward functions $R(s, a, s')$

Given a MDP model, we want to find the optimal policy that leads to the highest expected future reward. To do this, we can use three ways:

- Exhaustive search
- Policy iteration
- Value iteration

2.3 Exhaustive searching and policy evaluation

We started from exhaustive searching, where we evaluate the quality of all the possible policies based on the expected future reward $V^\pi(s_0)$ for each policy π at the starting state s_0.

Let's look at an example MDP (Figure 1) (the example used in the slide 18/25 in the lecture). To help illustrate a policy, we draw the policy graph, which restrict to the nodes (i.e., states and actions) that can be visited based on the policy. Three policies, and their policy graphs and policy evaluation results are shown in Table 1. For simple policy graphs that do not contain loops, such as the first and second one in Table 1, we can start from the later states and backpropagate to compute $V^\pi(s)$ of earlier states. For complex policy graphs that contain loops, such as the third one in Table 1, we need to solve a linear system of equations for $V^\pi(s)$.

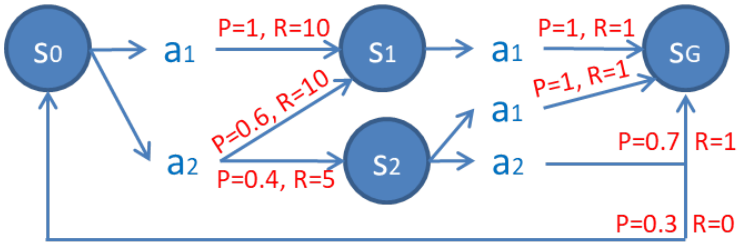


Figure 1: An Example MDP

In a word, for exhaustive searching, to compute expected future rewards at all states, we solve a linear system of equations as per Equation (1) (with the value at ending state s_G equal to 0).

$$V^\pi(s) = \sum_{s' \in S} P(s'|s, \pi(s))[R(s', s, \pi(s)) + V^\pi(s')] \tag{1}$$

A less computationally expensive alternative to solving this linear system of equations is to iteratively update the $V^\pi(s)$ as shown in Algorithm 1.

Algorithm 1 Policy Evaluation

1: Given a specific policy π
2: Initialize all $V^\pi(s) = 0$, for all $s \in S$
3: **while** $V^\pi_{i+1}(s) - V^\pi_i(s) >= \theta$ (a small positive number) **do**
4: Update the $V^\pi(s)$: $V^\pi_{i+1}(s) \leftarrow \sum_{s' \in S} P(s'|s, \pi(s))[R(s', s, \pi(s)) + V^\pi_i(s')]$
5: **end while**
6: **return** $V^\pi(s)$

Let's try to iteratively update the $V^\pi(s)$ to evaluate the third example policy in Table 1 by hand. The first few iterations are shown in Table 2.

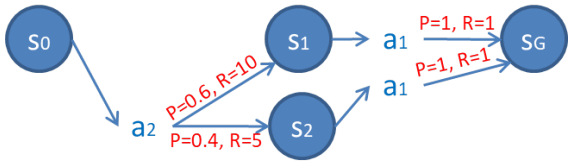
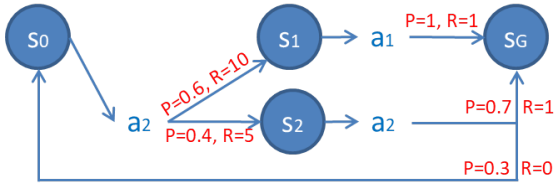
Policy	Policy graph	Solution
$\pi(s_0) = a_1,\ \pi(s_1) = a_1$		$V^\pi(s_1) = 1$ $V^\pi(s_0) = 11$
$\pi(s_0) = a_2,\ \pi(s_2) = a_1$		$V^\pi(s_1) = 1$ $V^\pi(s_2) = 1$ $V^\pi(s_0) = 0.6\times(10+1)+$ $0.4 \times (5+1) = 9$
$\pi(s_0) = a_2,\ \pi(s_2) = a_2$		$V^\pi(s_1) = 1$ $V^\pi(s_2) = 0.7\times 1+0.3\times$ $V^\pi(s_0)$ $V^\pi(s_0) = 0.4 \times (5 +$ $V^\pi(s_2)) + 0.6 \times (10 +$ $V^\pi(s_1))$

Table 1: Example Policies and Policy Evaluation by Solving Linear System of Equations

Iteration	$V^\pi(s_1)$	$V^\pi(s_2)$	$V^\pi(s_0)$
Initialize	0	0	0
Iteration 1	$1\times(1+0) = 1$	$0.7 \times (1+0) + 0.3 \times (0+0) = 0.7$	$0.6\times(10+1)+0.4\times(5+0) = 8.6$
Iteration 2	$1\times(1+0) = 1$ converge	$0.7\times(1+0)+0.3\times(0+8.6) = 3.28$	$0.6 \times (10+1) + 0.4 \times (5+0.7) =$ 8.88
Iteration 3	1	$0.7 \times (1+0) + 0.3 \times (0+8.88) =$ 3.36	$0.6\times(10+1)+0.4\times(5+3.28) =$ 9.91
Iteration k	1	close to 3.73	close to 10.09

Table 2: Iterative Policy Evaluation Example

2.4 Policy iteration and value iteration

Since there are at most $|A|^{|S|}$ distinct policies, exhaustively searching over the entire policy space is too expensive. Instead, we use policy or value iteration.

2.4.1 Policy iteration

Policy iteration, as shown in Algorithm 2, is to randomly choose one policy, conduct the policy evaluation, and then update/improve the policy, as per Equation (2). It can be seen as a gradient descent on policy.

$$\pi(s) = arg\max_{a\in A_s} \sum_{s'\in S} P(s'|s,a)[R(s',a,s) + V^\pi(s')] \tag{2}$$

Algorithm 2 Policy Iteration

1: Initialize an arbitrary policy π
2: **while** π still changes **do**
3: Compute the value function using π:
4: Solve the linear system of equations [Equation (2)]
5: Or iteratively evaluate the policy [Algorithm (1)]
6: Decode policy π at each state [Equation (3)]
7: **end while**
8: **return** π

Let's go back to the example MDP depicted in Figure 1. Assume we use policy iteration to find the optimal policy for this MDP, and we initialize an arbitrary policy $\pi(s_0) = a_2$, $\pi(s_2) = a_1$. The iterations are shown in Table 3. The policy iteration stops after one iteration because the policy converges. And thus the optimal policy is $\pi(s_0) = a_1$, $\pi(s_1) = a_1$, $\pi(s_2) = a_2$.

iteration	current policy	$V^\pi(s_0)$	$V^\pi(s_1)$	$V^\pi(s_2)$
Initialize	$\pi(s_0) = a_2$,$\pi(s_1) = a_1$, $\pi(s_2) = a_1$	10.09	1	3.73
Iteration 1	$\pi(s_0) = a_1$,$\pi(s_1) = a_1$, $\pi(s_2) = a_2$	11	1	4

Table 3: Policy Iteration Example

2.4.2 Value iteration

On the other hand, instead of gradually improving the policy, in value iteration, we try to find the $V^*(s)$, the expected future reward being at state s and following the optimal policy, directly. To do this, we introduce $Q(s, a)$, the expected future reward being at state s and taking action a. Different from $V(s)$, which depends on the state only, $Q(s, a)$ depends on both state and the action. $Q(s, a)$as per Equation (3) (i.e., the Bellman optimality principle). The relation between the two values thus is shown in Equation (3) and (4). And these two equations lead to the Bellman optimality principle, which is Equation (5).

$$V^*(s) = arg\max_{a \in A_s} Q^*(s, a) \tag{3}$$

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a)[R(s', a, s) + V^*(s')] \tag{4}$$

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a)[R(s', a, s) + \max_{a' \in A_{s'}} Q^*(s', a')] \tag{5}$$

After obtaining the $Q^*(s, a)$, we can decode the optimal policy easily, as per Equation (6).

$$\pi(s) = arg\max_{a \in A_s} Q^*(s, a) \tag{6}$$

To sum up, value iteration, as shown in Algorithm 3, is to iteratively compute $Q^*(s, a)$ as per Equation (3) and update the the $V^*(s)$ as per Equation (4), and decode the optimal policy as per Equation (6).

Algorithm 3 Value Iteration

```
Initialize V(s)
while V(s) does not converge do
    for all s ∈ S do
        for all a ∈ A_s do
            Compute Q*(s, a) [Equation (5)]
        end for
        Update V(s) [Equation (6)]
    end for
end while
Decode π [Equation (4)]
```

Let's again go back to the example MDP depicted in Figure 1 and use value iteration to find the optimal policy for this MDP. The iterations are shown in Table 4. The value iteration stops after three iterations when the $V^*(s)$ converges. The optimal policy is $\pi(s_0) = a_1$, $\pi(s_1) = a_1$, $\pi(s_2) = a_2$.

Iteration	$V^*(s_0)$	$Q^*(s_0, a_1)$	$Q^*(s_0, a_2)$	$V^*(s_1)$	$Q^*(s_1, a_1)$	$V^*(s_2)$	$Q^*(s_2, a_1)$	$Q^*(s_2, a_2)$
Initialize	0	0	0	0	0	0	0	0
Iteration 1	10	$1 \times (10 + 0) = 10$	$0.6 \times (10 + 0) + 0.4 \times (5 + 0) = 8$	1	1	1	1	$0.7 \times (1 + 0) + 0.3 \times 0 = 0.7$
Iteration 2	11	$1 \times (10 + 1) = 11$	$0.6 \times (10 + 1) + 0.4 \times (5 + 1) = 9$	1	1	3.7	1	$0.7 \times (1 + 0) + 0.3 \times (0 + 10) = 3.7$
Iteration 3	11	$1 \times (10 + 1) = 11$	$0.6 \times (10 + 1) + 0.4 \times (5 + 1) = 9$	1	1	4	1	$0.7 \times (1 + 0) + 0.3 \times (0+11) = 4$

Table 4: Value Iteration Example

3 Q-Learning

Now some ingredients of the MDP model are missing, e.g., the transition probabilities and/or the reward functions. And sometimes (or in most of the real world scenarios) we do not even have a MDP model. We started from the scenario where only the transition probabilities are missing.

3.1 An initial solution

The most straightforward solution to the problem of missing transition probabilities is to run a simulator to collect experience tuples $(s, a, r, s^{'})$, and approximate the transition probabilities using those tuples.

The problem of this initial solution is that we need too many times of simulation to obtain enough tuples to get a good estimate of the transition probabilities, especially for a large MDP model.

3.2 A better solution: update $Q(s, a)$

Instead of getting many experience tuples at once, we obtain some tuples (e.g., one tuple) at a time, and then approximate the $Q(s, a)$ as per Equation (7).

$$Q(s,a) \approx \underbrace{y_{(s,a,r,s')}}_{\text{learned value}} = \underbrace{r}_{\text{reward}} + \underbrace{\max_{a' \in A_{s'}} Q(s', a')}_{\text{optimal future value}} \tag{7}$$

To account for missing transition probability, we keep running average to update the $Q(s, a)$ as per the equation (8).

$$\underbrace{Q(s,a)}_{\text{new value}} \leftarrow (1-\alpha) \underbrace{Q(s,a)}_{\text{current value}} + \alpha \underbrace{y_{(s,a,r,s')}}_{\text{learned value}} \tag{8}$$

which can be seen as a gradient descent on Q value as shown in equation (9).

$$Q(s,a) \leftarrow Q(s,a) + \alpha(y_{(s,a,r,s')} - Q(s,a)) \tag{9}$$

The Q-learning algorithm is shown in Algorithm 3. Note that the $Q(s, a)$ is initialized randomly, with a larger α, so that $Q(s, a)$ can be updated more toward y during the initial learning.

Algorithm 4 Q-learning

1: Initialize $Q(s, a)$
2: **while** $Q(s, a)$ does not converge **do**
3: Simulate to obtain a sample transition (s, a, r, s')
4: Approximate $Q(s, a)$ [Equation (7)]
5: Update $Q(s, a)$ [Equation (8)]
6: **end while**
7: **Decode** π [Equation (4)]

Let's look at an example. For the reinforcement learning environment depicted in Figure 2, given a starting position (e.g., Row 1 Column 3) and an action (e.g., upward), we can initialize the Q-learning algorithm and update the Q values for each pair of state and action. Finally, after several iterations (e.g., in the example of Figure 2, 1,000 iterations), we get a table of Q values that can be used to choose the best policy.

Based on the table of Q values, we decode the optimal policy according to Equation (6) for each and every state (i.e., cell). For example, assume the agent/robot now is at the bottom left corner cell. The agent/robot should first move up because this action has the largest Q value (i.e., 0.54) out of the four possible actions, at this state; and then the agent/robot should move up, right, right, right, and arrive at the cell that gives a reward of 1.

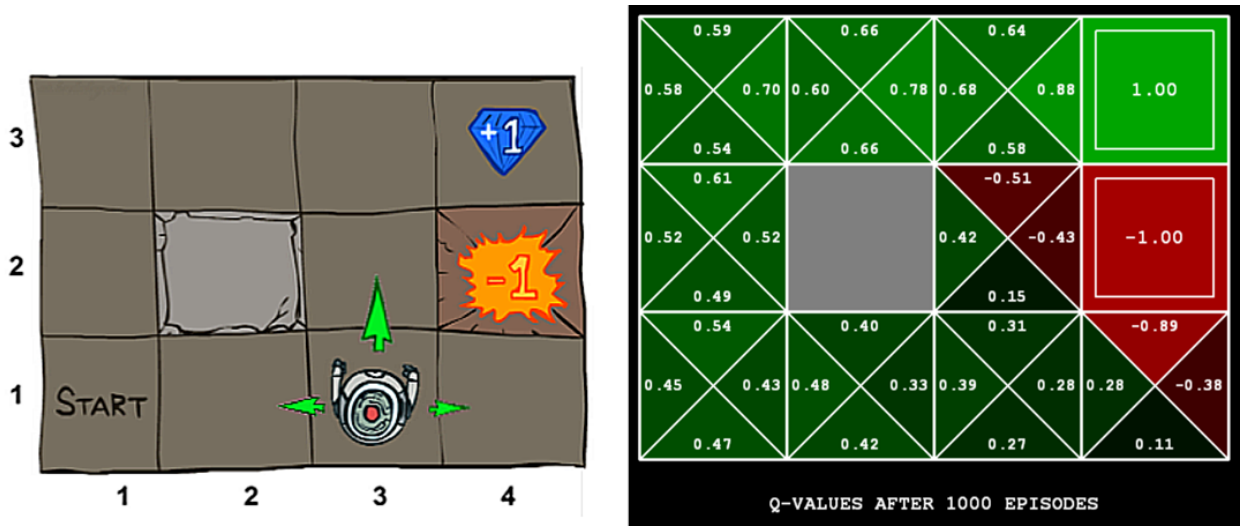


Figure 2: Example Q-Learning with Known States

Now let's do one more iteration of Q-learning by hand based on what we have in the table in Figure 2. Assume the agent/robot is at Row 1, Column 3 (denoted by s), and is moving upward (denoted by a) to Row 2, Column 3 (denoted by s'). And the learning rate α is 0.1.

The learned Q value will be:

$y = r + \max_{a' \in A_{s'}} Q(s', a') = 0 + max(-0.51, -0.43, 0.15, 0.42) = 0.42$

Update the $Q(s, a)$ to be:

$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha y = (1 - 0.1) \times 0.31 + 0.1 \times 0.42 = 0.32$

3.3 Deep Q-learning

Now think about Atari games (Figure 3). In Breakout, one of the Atari games, the actions could be moving the panel left, right, up, and down. The rewards could be the scores the player earns after eliminate the breaks. But we do not know the states of the MDP model, and thus we are not able to compute the table of Q values as what we do in Section 3.2 and decode the optimal policy accordingly.

Figure 3: Atari Games (Left-to-right) Pong, Breakout, Space Invaders, Seaquest, Beam Rider [1]

The solution is to use a deep neural network to take in a image that describing the current state, and output the expected future reward (i.e., Q values) for each of the possible actions, as shown in Figure 4.

Let's go back to the Breakout example. For deep Q-learning of Breakout, a vector of pixel values of an image/snapshot of the game is fed into a convolutional neural network, which consists of a series of convolution, ReLU, pooling, and fully connected layers. The output of the network is a vector of Q values, each corresponding to a possible action the player can take (e.g., moving the panel left, right, up, down, etc.).

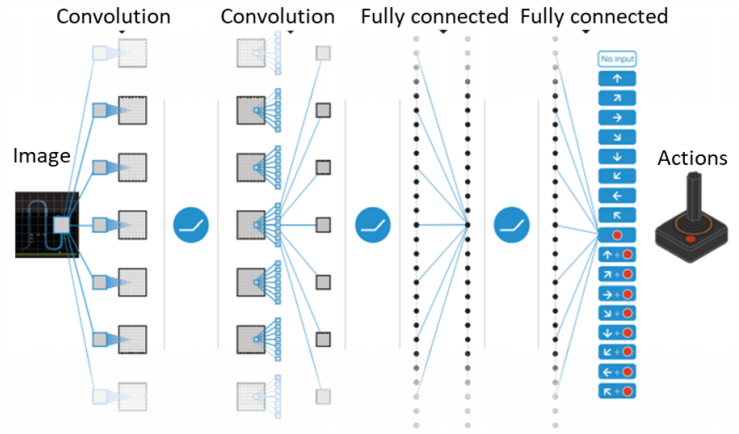


Figure 4: Deep Q-Learning [2]

Given experience tuples $D = (s_j, a_j, r_j, s_{j+1})$, the training of deep Q-learning networks consists of the following steps:

1. Sample a minibatch $B \subset D$
2. Compute target value $y_j = r_j + \gamma max_a Q_\theta(s_{j+1}, a)$
3. Use stochastic (semi-) gradient descent to optimize w.r.t. parameter θ. Notice that the objective function (10) is actually the the loss function for regression, which aims to minimize the mean squared error between the Q values predicted by the deep Q-learning network and the target value y:
$$\min_\theta \sum_{(s_j, a_j, r_j, s_{j+1}) \in B} (Q_{\theta^-}(s_j, a_j) - y_j)^2 \tag{10}$$
4. Based on the new Q values, take a new action to simulate more tuples to supplement D. The new action has a probability of ϵ to be randomly chosen and a probability of $(1 - \epsilon)$ to be chosen according to the current estimate of Q values. This is called ϵ greedy action.

Note that the whole process is initialized with an empty D, and taking a random action. We want to initialize reasonably small Q values, so that the y is not dominated by the Q values but rather by the reward r. The dataset continues to grow in the iterations, and thus the agent have better idea what actions to take.

3.4 Summary

	Known MDP	Unknown MDP
To compute V^*, Q^*, π^*	Value or policy iteration	Q-learning Only transition probabilities are unknown: learn the table of Q values Whole MDP model is unknown: deep Q-learning
To evaluate a given/fixed policy	Policy evaluation	Value learning

Table 5: Summary

4 Quiz Answers

1. What differentiates reinforcement learning from supervised learning?

   Ans: Please refer to subsection 2.1.

2. What is a MDP?

   Ans: Please refer to subsection 2.2.

3. What to do if no transition probabilities are available?

   Ans: We simulate to get tuples $D = (s_j, a_j, r_j, s_{j+1})$, approximate and update the Q values iteratively.

5 Questions/Discussions in Class

1. **Q**: Is a policy deterministic?

   A: A policy is always deterministic.

2. **Q**: How to change the rewards in the example MDP to make the second policy better than the first policy in Table 1?

   A: For example, we can increase the rewards $R(s_0, a_2, s_1)$ or $R(s_0, a_2, s_2)$.

3. **Q**: Why do we usually prefer value iteration over policy iteration?

   A: Value iteration allows easier and less computationally expensive decoding of the optimal policy.

4. **Q**: In Figure 2, why the Q value corresponding to the Row 1, Column 2 cell, and going upward is negative?

   A: It should (finally) be positive. The fact that it is negative is likely because there has been only 1000 iterations, which are not enough for Q values to converge.

5. **Q**: Where does the dataset used in deep Q-learning $D = (s_j, a_j, r_j, s_{j+1})$ come from?

   A: The dataset D is empty at the beginning of learning. We choose a starting point and randomly take an action and get a tuple (s_j, a_j, r_j, s_{j+1}) to initiate the learning iterations. Each iteration we perform ϵ greedy action to get new tuples to grow the dataset D.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.