

Homework 4: Peeking Blackjack

Brandon McKinzie

PROBLEM 1: VALUE ITERATION

In this problem, you will perform the value iteration updates manually on a very basic game just to solidify your intuitions about solving MDPs. The set of possible states in this game is $\{-2, -1, 0, 1, 2\}$. You start at state 0, and if you reach either -2 or 2, the game ends. At each state, you can take one of two actions: $\{-1, +1\}$.

If you're in state s and choose $+1$:

- You have a 70% chance of reaching the state $s+1$.
- You have a 30% chance of reaching the state $s-1$.

If you're in state s and choose -1 :

- You have a 20% chance of reaching the state $s+1$.
- You have an 80% chance of reaching the state $s-1$.

If your action results in transitioning to state -2, then you receive a reward of 20. If your action results in transitioning to state 2, then your reward is 100. Otherwise, your reward is -5. Assume the discount factor γ is 1.

(a) Give the value of $V_{opt}(s)$ for each state s after 0, 1, and 2 iterations of value iteration. Iteration 0 just initializes all the values of V to 0. Terminal states do not have any optimal policies and take on a value of 0.

By definition $V_{opt}^{(0)}(s) \leftarrow 0$. Using this and the fact that $\gamma = 1$, we can compute

$$V_{opt}^{(1)}(s) = \max_{a \in \{+1, -1\}} \sum_{s' \in \{s+1, s-1\}} T(s, a, s') R(s, a, s') \quad (1)$$

$$= \max \{0.7R(s, 1, s+1) + 0.3R(s, 1, s-1), 0.2R(s, -1, s+1) + 0.8R(s, -1, s-1)\} \quad (2)$$

$$V_{opt}^{(1)}(0) = \max \{0.7(-5) + 0.3(-5), 0.2(-5) + 0.8(-5)\} = -5 \quad (3)$$

$$V_{opt}^{(1)}(-1) = \max \{0.7(-5) + 0.3(20), 0.2(-5) + 0.8(20)\} = 15 \quad (4)$$

$$V_{opt}^{(1)}(1) = \max \{0.7(100) + 0.3(-5), 0.2(100) + 0.8(-5)\} = 68.5 \quad (5)$$

$$V_{opt}^{(1)}(-2) = V_{opt}^{(1)}(2) = 0 \quad (6)$$

We then use these values to compute the next iteration.

$$V_{opt}^{(2)}(s) = \max_{a \in \{+1, -1\}} \sum_{s' \in \{s+1, s-1\}} T(s, a, s') [R(s, a, s') + V_{opt}^{(1)}(s')] \quad (7)$$

$$V_{opt}^{(2)}(0) = \max \{0.7(-5 + 68.5) + 0.3(-5 + 15), 0.2(-5 + 68.5) + 0.8(-5 + 15)\} \quad (8)$$

$$= 47.45 \quad (9)$$

$$V_{opt}^{(2)}(-1) = \max \{0.7(-5 - 5) + 0.3(20), 0.2(-5 - 5) + 0.8(20)\} \quad (10)$$

$$= 14 \quad (11)$$

$$V_{opt}^{(2)}(1) = \max \{0.7(100) + 0.3(-5 - 5), 0.2(100) + 0.8(-5 - 5)\} \quad (12)$$

$$= 67 \quad (13)$$

$$V_{opt}^{(2)}(-2) = V_{opt}^{(2)}(2) = 0 \quad (14)$$

(b) *What is the resulting optimal policy π_{opt} for all non-terminal states?*

$$\pi_{opt}(0) = +1 \tag{15}$$

$$\pi_{opt}(-1) = -1 \tag{16}$$

$$\pi_{opt}(1) = +1 \tag{17}$$

PROBLEM 2: TRANSFORMING MDPs

Let's implement value iteration to compute the optimal policy on an arbitrary MDP. Later, we'll create the specific MDP for Blackjack.

(a) I provided a counterexample in the code.

(b) Suppose we have an acyclic MDP for which we want to find the optimal value at each node. We could run value iteration, which would require multiple iterations – but it would be nice to be more efficient for MDPs with this acyclic property. Briefly explain an algorithm that will allow us to compute V_{opt} for each node with only a single pass over all the triples.

For acyclic graphs, we can work our way from terminal (end-state) nodes back to the start state node. This will allow us to only do a single pass over the (s, a, s') triples because for all of the end state nodes s_{end} , we'll have $V_{opt}(s_{end}) = 0$. All edges that originate from a chance node to an end state node will be computed next. Specifically, for each end-state node s_{end} , we'll compute

$$T(s, a, s_{end})[R(s, a, s_{end}) + \gamma V_{opt}(s_{end})] \quad \forall (s, a) : T(s, a, s_{end}) > 0 \quad (18)$$

and continue working our way back until we reach the start state. Note that this is a dynamic programming approach.

(c) Suppose we have an MDP with states $States$ and a discount factor $\gamma < 1$, but we have an MDP solver that can only solve MDPs with discount factor of 1. How can we leverage the MDP solver to solve the original MDP?

Let us define a new MDP with states $States' = States \cup \{o\}$, where o is a new state. Let's use the same actions ($Actions'(s) = Actions(s)$), but we need to keep the discount $\gamma' = 1$. Your job is to define new transition probabilities $T'(s, a, s')$ and rewards $Reward'(s, a, s')$ in terms of the old MDP such that the optimal values $V_{opt}(s) \forall s \in States$ are equal under the original MDP and the new MDP.

We can arrive at the result by first seeing what happens when we factor out gamma in the summation of

$$V_{opt}(s) = \max_{a \in Actions(s)} Q(s, a) \quad (19)$$

$$Q(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_{opt}(s')] \quad (20)$$

$$= \sum_{s'} \gamma T(s, a, s') \left[\frac{1}{\gamma} R(s, a, s') + V_{opt}(s') \right] \quad (21)$$

$$= \sum_{s'} T'(s, a, s') [R'(s, a, s') + V_{opt}(s')] \quad (22)$$

and voilà, γ is gone, if we define

$$T'(s, a, s') = \gamma T(s, a, s') \quad (23)$$

$$R'(s, a, s') = \frac{1}{\gamma} R(s, a, s') \quad (24)$$

where $s' \neq o$. In order to satisfy the constraint that $\sum_{s'} T'(s, a, s') = 1$, we define

$$T'(s, a, o) := 1 - \gamma \quad (25)$$

which gives us

$$\sum_{s'} T'(s, a, s') = \left(\sum_{s' \neq o} \gamma T(s, a, s') \right) + T'(s, a, o) \quad (26)$$

$$= (\gamma) + (1 - \gamma) \quad (27)$$

$$= 1 \quad (28)$$

We see that γ in this reformulation represents the probability of transitioning to any state that isn't o . Lastly, to ensure that the new state o does not impact V_{opt} , we define

$$R(s, a, o) := 0 \quad (29)$$

$$IsEnd(o) := \text{True} \quad (30)$$

which also implies that $V_{opt}(o) = 0$.

PROBLEM 4: LEARNING TO PLAY BLACKJACK

(b) Now let's apply Q-learning to an MDP and see how well it performs in comparison with value iteration. First, call `simulate` using your Q-learning code and the `identityFeatureExtractor()` on the MDP `smallMDP` (defined for you in `submission.py`), with 30000 trials and default `explorationProb`.

How does the Q-learning policy compare with a policy learned by value iteration (i.e., for how many states do they produce a different action)? (Don't forget to set the `explorationProb` of your Q-learning algorithm to 0 after learning the policy.) Now run `simulate()` on `largeMDP`, again with 30,000 trials. How does the policy learned in this case compare to the policy learned by value iteration? What went wrong?

For the `smallMDP`, Q-learning and value iteration returned the same action for about 80 percent of the states (given by `mdp.states`). For the `largeMDP`, they agreed for 83 percent of all states. The larger MDP has a substantially larger state space to explore.

(d) Sometimes, we might reasonably wonder how an optimal policy learned for one MDP might perform if applied to another MDP with similar structure but slightly different characteristics. For example, imagine that you created an MDP to choose an optimal strategy for playing "traditional" blackjack, with a standard card deck and a threshold of 21. You're living it up in Vegas every weekend, but the casinos get wise to your approach and decide to make a change to the game to disrupt your strategy: going forward, the threshold for the blackjack tables is 17 instead of 21. If you continued playing the modified game with your original policy, how well would you do? (This is just a hypothetical example; we won't look specifically at the blackjack game in this problem.)

The problem with `FixedRLAlgorithm` is that it is unable to modify its policy to better suit the modified MDP. This leads to suboptimal results when run it on `modifiedMDP`. Q-learning, however, is able to modify its policy in order to adapt to the modified MPD. Therefore, we expect to observe higher rewards when running Q-learning on `modifiedMDP` compared to `FixedRLAlgorithm`¹

¹There is some bug in my Q-learning implementation that is causing it to output lower rewards than I'd expect.