CS 188 Introduction to Artificial Intelligence
Spring 2024 Note 18

Author (all other notes): Nikhil Sharma

Author (Bayes' Nets notes): Josh Hug and Jacky Liang, edited by Regina Wang

Author (Logic notes): Henry Zhu, edited by Peyrin Kao

Credit (Machine Learning and Logic notes): Some sections adapted from the textbook *Artificial Intelligence: A Modern Approach*.

Last updated: August 26, 2023

Value Iteration

Now that we have a framework to test for optimality of the values of states in a MDP, the natural follow-up question to ask is how to actually compute these optimal values. To answer this question, we need **time-limited values** (the natural result of enforcing finite horizons). The time-limited value for a state s with a time-limit of k timesteps is denoted $U_k(s)$, and represents the maximum expected utility attainable from s given that the Markov decision process under consideration terminates in k timesteps. Equivalently, this is what a depth-k expectimax run on the search tree for a MDP returns.

Value iteration is a **dynamic programming algorithm** that uses an iteratively longer time limit to compute time-limited values until convergence (that is, until the U values are the same for each state as they were in the past iteration: $\forall s, U_{k+1}(s) = U_k(s)$). It operates as follows:

1. $\forall s \in S$, initialize $U_0(s) = 0$. This should be intuitive, since setting a time limit of 0 timesteps means no actions can be taken before termination, and so no rewards can be acquired.
2. Repeat the following update rule until convergence:

$$\forall s \in S,\ U_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U_k(s')]$$

   At iteration k of value iteration, we use the time-limited values for with limit k for each state to generate the time-limited values with limit $(k+1)$. In essence, we use computed solutions to subproblems (all the $U_k(s)$) to iteratively build up solutions to larger subproblems (all the $U_{k+1}(s)$); this is what makes value iteration a dynamic programming algorithm.

Note that though the Bellman equation looks essentially identical in construction to the update rule above, they are not the same. The Bellman equation gives a condition for optimality, while the update rule gives a method to iteratively update values until convergence. When convergence is reached, the Bellman equation will hold for every state: $\forall s \in S,\ U_k(s) = U_{k+1}(s) = U^*(s)$.

For conciseness, we frequently denote $U_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U_k(s')]$ with the shorthand $U_{k+1} \leftarrow BU_k$, where B is called the Bellman operator. The Bellman operator is a contraction by γ. To prove this, we will need the following general inequality:

$$|max_z f(z) - max_z h(z)| \leq max_z |f(z) - h(z)|.$$

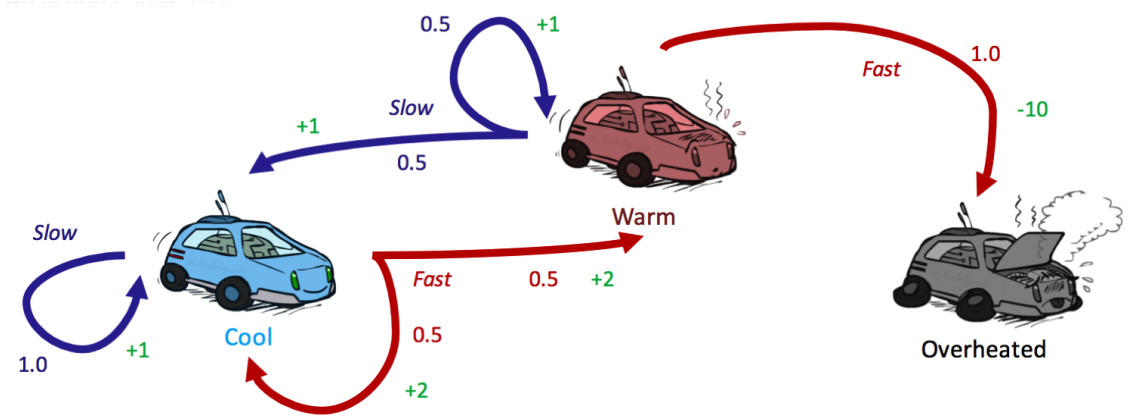
Now consider two value functions evaluated at the same state $U(s)$ and $U'(s)$. We show that the Bellman update B is a contraction by $\gamma \in (0,1)$ with respect to the max norm as follows

$$\begin{aligned}
&|BU(s) - BU'(s)| \\
&= \left| \left(\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U(s')] \right) - \left(\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U'(s')] \right) \right| \\
&\leq \max_a \left| \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U(s')] - \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U'(s')] \right| \\
&= \max_a \left| \gamma \sum_{s'} T(s,a,s') U(s') - \gamma \sum_{s'} T(s,a,s') U'(s') \right| \\
&= \gamma \max_a \left| \sum_{s'} T(s,a,s')(U(s') - U'(s')) \right| \\
&\leq \gamma \max_a \left| \sum_{s'} T(s,a,s') \max_{s'} |U(s') - U'(s')| \right| \\
&= \gamma \max_{s'} |U(s') - U'(s')| \\
&= \gamma ||U(s') - U'(s')||_\infty,
\end{aligned}$$

where the first inequality follows from the general inequality introduced above, the second inequality follows from taking the maximum of the differences between U and U' and finally in the second to last step we use the fact that probabilities sum to 1 no matter the choice of a. The last step uses the definition of the max norm for a vector $x = (x_1, \ldots, x_n)$ which is $||x||_\infty = \max(|x_1|, \ldots, |x_n|)$.

Because we just proved that value iteration via bellman updates are a contraction in γ, we know that value iteration converges, and convergence happens when we have reached a fixed point which satisfies $U^* = BU^*$.

Let's see a few updates of value iteration in practice by revisiting our racecar MDP from earlier, introducing a discount factor of $\gamma = 0.5$:



We begin value iteration by initialization of all $U_0(s) = 0$:

	cool	**warm**	**overheated**
U_0	0	0	0

In our first round of updates, we can compute $\forall s \in S,\ U_1(s)$ as follows:

$$\begin{aligned}
U_1(cool) &= \max\{1\cdot[1+0.5\cdot 0],\ 0.5\cdot[2+0.5\cdot 0]+0.5\cdot[2+0.5\cdot 0]\} \\
&= \max\{1,2\} \\
&= \boxed{2} \\
U_1(warm) &= \max\{0.5\cdot[1+0.5\cdot 0]+0.5\cdot[1+0.5\cdot 0],\ 1\cdot[-10+0.5\cdot 0]\} \\
&= \max\{1,-10\} \\
&= \boxed{1} \\
U_1(overheated) &= \max\{\} \\
&= \boxed{0}
\end{aligned}$$

	cool	**warm**	**overheated**
U_0	0	0	0
U_1	2	1	0

Similarly, we can repeat the procedure to compute a second round of updates with our newfound values for $U_1(s)$ to compute $U_2(s)$.

$$\begin{aligned}
U_2(cool) &= \max\{1\cdot[1+0.5\cdot 2],\ 0.5\cdot[2+0.5\cdot 2]+0.5\cdot[2+0.5\cdot 1]\} \\
&= \max\{2,2.75\} \\
&= \boxed{2.75} \\
U_2(warm) &= \max\{0.5\cdot[1+0.5\cdot 2]+0.5\cdot[1+0.5\cdot 1],\ 1\cdot[-10+0.5\cdot 0]\} \\
&= \max\{1.75,-10\} \\
&= \boxed{1.75} \\
U_2(overheated) &= \max\{\} \\
&= \boxed{0}
\end{aligned}$$

	cool	**warm**	**overheated**
U_0	0	0	0
U_1	2	1	0
U_2	2.75	1.75	0

It's worthwhile to observe that $U^*(s)$ for any terminal state must be 0, since no actions can ever be taken from any terminal state to reap any rewards.

Policy Extraction

Recall that our ultimate goal in solving a MDP is to determine an optimal policy. This can be done once all optimal values for states are determined using a method called **policy extraction**. The intuition behind policy extraction is very simple: if you're in a state s, you should take the action a which yields the maximum

expected utility. Not surprisingly, a is the action which takes us to the Q-state with maximum Q-value, allowing for a formal definition of the optimal policy:

$$\forall s \in S,\ \pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^*(s')]$$

It's useful to keep in mind for performance reasons that it's better for policy extraction to have the optimal Q-values of states, in which case a single argmax operation is all that is required to determine the optimal action from a state. Storing only each $U^*(s)$ means that we must recompute all necessary Q-values with the Bellman equation before applying argmax, equivalent to performing a depth-1 expectimax.

Q-Value Iteration

In solving for an optimal policy using value iteration, we first find all the optimal values, then extract the policy using policy extraction. However, you might have noticed that we also deal with another type of value that encodes information about the optimal policy: Q-values.

Q-value iteration is a dynamic programming algorithm that computes time-limited Q-values. It is described in the following equation:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

Note that this update is only a slight modification over the update rule for value iteration. Indeed, the only real difference is that the position of the max operator over actions has been changed since we select an action before transitioning when we're in a state, but we transition before selecting a new action when we're in a Q-state. Once we have the optimal Q-values for each state and action, we can then find the policy for a state by simply choosing the action which has the highest Q-value.

Policy Iteration

Value iteration can be quite slow. At each iteration, we must update the values of all $|S|$ states (where $|n|$ refers to the cardinality operator), each of which requires iteration over all $|A|$ actions as we compute the Q-value for each action. The computation of each of these Q-values, in turn, requires iteration over each of the $|S|$ states again, leading to a poor runtime of $O(|S|^2|A|)$. Additionally, when all we want to determine is the optimal policy for the MDP, value iteration tends to do a lot of overcomputation since the policy as computed by policy extraction generally converges significantly faster than the values themselves. The fix for these flaws is to use **policy iteration** as an alternative, an algorithm that maintains the optimality of value iteration while providing significant performance gains. Policy iteration operates as follows:

1. Define an *initial policy*. This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until convergence:
   - Evaluate the current policy with **policy evaluation**. For a policy π, policy evaluation means computing $U^\pi(s)$ for all states s, where $U^\pi(s)$ is expected utility of starting in state s when following π:
   $$U^\pi(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma U^\pi(s')]$$

Define the policy at iteration i of policy iteration as π_i. Since we are fixing a single action for each state, we no longer need the max operator which effectively leaves us with a system of $|S|$ equations generated by the above rule. Each $U^{\pi_i}(s)$ can then be computed by simply solving this system. Alternatively, we can also compute $U^{\pi_i}(s)$ by using the following update rule until convergence, just like in value iteration:

$$U_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma U_k^{\pi_i}(s')]$$

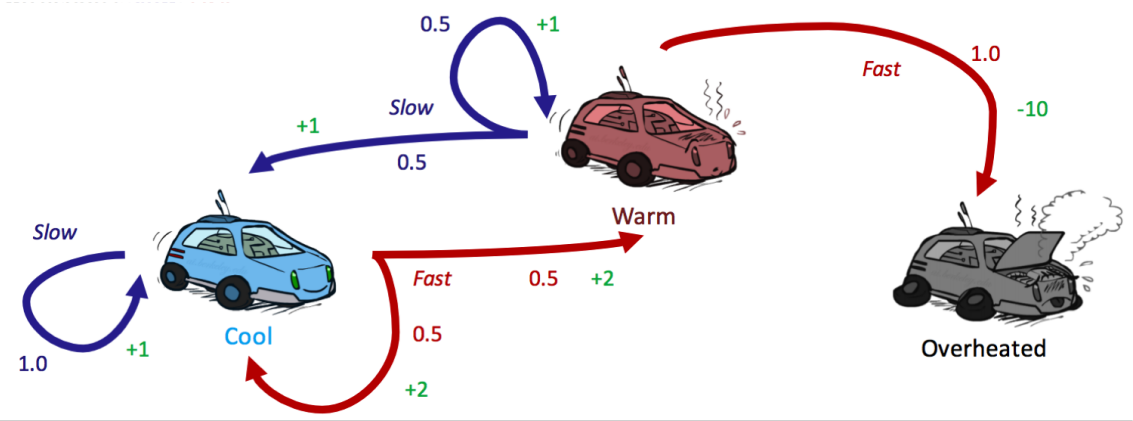
However, this second method is typically slower in practice.

- Once we've evaluated the current policy, use **policy improvement** to generate a better policy. Policy improvement uses policy extraction on the values of states generated by policy evaluation to generate this new and improved policy:

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma U^{\pi_i}(s')]$$

If $\pi_{i+1} = \pi_i$, the algorithm has converged, and we can conclude that $\pi_{i+1} = \pi_i = \pi^*$.

Let's run through our racecar example one last time (getting tired of it yet?) to see if we get the same policy using policy iteration as we did with value iteration. Recall that we were using a discount factor of $\gamma = 0.5$.



We start with an initial policy of *Always go slow*:

	cool	**warm**	**overheated**
π_0	*slow*	*slow*	–

Because terminal states have no outgoing actions, no policy can assign a value to one. Hence, it's reasonable to disregard the state *overheated* from consideration as we have done, and simply assign $\forall i,\ U^{\pi_i}(s) = 0$ for any terminal state s. The next step is to run a round of policy evaluation on π_0:

$$\begin{aligned} U^{\pi_0}(cool) &= 1 \cdot [1 + 0.5 \cdot U^{\pi_0}(cool)] \\ U^{\pi_0}(warm) &= 0.5 \cdot [1 + 0.5 \cdot U^{\pi_0}(cool)] + 0.5 \cdot [1 + 0.5 \cdot U^{\pi_0}(warm)] \end{aligned}$$

Solving this system of equations for $U^{\pi_0}(cool)$ and $U^{\pi_0}(warm)$ yields:

	cool	warm	overheated
U^{π_0}	2	2	0

We can now run policy extraction with these values:

$$\begin{aligned}
\pi_1(cool) &= \text{argmax}\{slow : 1 \cdot [1 + 0.5 \cdot 2],\ fast : 0.5 \cdot [2 + 0.5 \cdot 2] + 0.5 \cdot [2 + 0.5 \cdot 2]\} \\
&= \text{argmax}\{slow : 2,\ fast : 3\} \\
&= \boxed{fast} \\
\pi_1(warm) &= \text{argmax}\{slow : 0.5 \cdot [1 + 0.5 \cdot 2] + 0.5 \cdot [1 + 0.5 \cdot 2],\ fast : 1 \cdot [-10 + 0.5 \cdot 0]\} \\
&= \text{argmax}\{slow : 3,\ fast : -10\} \\
&= \boxed{slow}
\end{aligned}$$

Running policy iteration for a second round yields $\pi_2(cool) = fast$ and $\pi_2(warm) = slow$. Since this is the same policy as π_1, we can conclude that $\pi_1 = \pi_2 = \pi^*$. Verify this for practice!

	cool	warm
π_0	$slow$	$slow$
π_1	$fast$	$slow$
π_2	$fast$	$slow$

This example shows the true power of policy iteration: with only two iterations, we've already arrived at the optimal policy for our racecar MDP! This is more than we can say for when we ran value iteration on the same MDP, which was still several iterations from convergence after the two updates we performed.

Summary

The material presented above has much opportunity for confusion. We covered value iteration, policy iteration, policy extraction, and policy evaluation, all of which look similar, using the Bellman equation with subtle variation. Below is a summary of the purpose of each algorithm:

- *Value iteration*: Used for computing the optimal values of states, by iterative updates until convergence.
- *Policy evaluation*: Used for computing the values of states under a specific policy.
- *Policy extraction*: Used for determining a policy given some state value function. If the state values are optimal, this policy will be optimal. This method is used after running value iteration, to compute an optimal policy from the optimal state values; or as a subroutine in policy iteration, to compute the best policy for the currently estimated state values.
- *Policy iteration*: A technique that encapsulates both policy evaluation and policy extraction and is used for iterative convergence to an optimal policy. It tends to outperform value iteration, by virtue of the fact that policies usually converge much faster than the values of states.