Markov Decision Processes and Reinforcement Learning

Aug 2, 2015

In this post, we'll review Markov Decision Processes and Reinforcement Learning. This material is from Chapters 17 and 21 in Russell and Norvig (2010).

Markov Decision Processes

The general idea with this situation is that we are an agent in some world, and we have a set of states, actions (for each state), a transition probability (which we may not actually know), and a reward function. The goal for a rational agent is to maximize the expected sum of (discounted) rewards for a state $\mathbb{E}[\sum_t \gamma^t R(s_t)]$, where $0 \leq \gamma \leq 1$ is our discount factor, which we usually assume is equal to one for toy cases only.

This is the setup of a Markov Decision Process (MDP), with the added constraint that the probability of going to another state only depends on the current state, which is why we call this a *Markov* Decision Process. Since the ultimate goal is to maximize the expected utility, we will need to learn a *policy* function π that maps states to actions. Finally, while not strictly necessary, it is common to "spice up" the MDP problem by assuming that actions are *non-deterministic*. In the canonical grid-world example described in the book (and in a lot of undergraduate AI classes, for that matter), we assume if we move *North*, then that has an 80 percent chance of succeeding. Hence, the transition probability $P(s' \mid s, a) = P(s, a, s')$ is nontrivial, where s and s' are states, and a is the action we took at state s.

In order to understand how to get an optimal policy, we first need to realize how exactly to define the *value* $V^*(s)$ of a state, which in other words, *is the expected utility we will get if we are at state s and play optimally*. (The asterisk here is to indicate optimality.) There are, sadly, *two* common formulations for $V^*(s)$. The first, from R & N, assumes that the *reward* aspect of the formula is defined in terms of a state only:

$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} P(s, a, s') V^*(s')$$

The second formulation, which Berkeley's CS 188 class uses[1], defines reward in terms of a state-action-successor triple:

$$V^*(s) = \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

Both of these equation sets can be called the **Bellman Equations**, which characterize the optimal values (but we will generally need some other way of computing them, as we show shortly). In general, I will utilize the second formulation, but the formulations are not fundamentally different.

It is also common to define a new quantity called a *Q-value* with respect to state-action pairs:

$$Q^*(s, a) = \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

In words, $Q(s, a)$ is the expected utility starting at state s, *taking action* a, and hereafter, playing optimally. We can therefore relate the Vs and Qs with the following equation:

$$V^*(s) = \max_a Q^*(s, a)$$

I find it easiest to think of these in terms of expectimax trees with chance nodes. The "normal" nodes correspond to Vs, and the "chance" nodes correspond to Qs. Both nodes have multiple successors: the Vs because we have *choices of actions*, and Qs because, even if we commit to an action, the actual *outcome* is generally non-deterministic, so we will not know what state we end up in. (I guess we can also view states as the "normal" nodes here.)

Given these definitions, how do we figure out the optimal policy? There are two common tactics:

- **Value Iteration** is an iterative algorithm that computes values of states indexed by a k, as in $V_k(s)$, which can also be thought of the best value of state s assuming the game ends in k time-steps. This is not the actual policy itself, but these values *are used* to determine the optimal policy[2].

  Starting with $V_0(s) = 0$ for all s, value iteration performs the following update:

  $$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

  and it repeats until we tell it to stop.

  To make this intuitive, during each step, we have a vector of $V_k(s)$ values, and then we do a *one-ply expectimax computation* to get the next vector. Note that expectimax *could* compute all of the values we need, but it would take too long due to repeated and infinite depth state trees.

To prove value iteration converges to V^* (and uniquely!), appeal to *contraction* and the fact that γ^k, so long as $\gamma \neq 1$, will go down to zero. Said another way, the "V_k tree" and the "V_{k+1} tree" are different only in their last layer, but that last layer's contribution is reduced exponentially.

- **Policy Iteration** is generally an improvement over Value Iteration because policies often converge long before the values do, so we alternate between *policy evaluation* and *policy improvement* steps. In the first step, we assume we are given a policy π and have to figure out the expected utilities of each state when executing π. These values are characterized by the following equations:

  $$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

  Note that the MAX operator is gone because $\pi(s)$ gives us our action. Thus, we can solve these equations in $O(n^3)$ time with standard matrix multiplication methods[3].

  It is actually rather easy to do policy improvement *from Q-values*:

  $$\pi^*(s) = \arg_a \max Q^*(s, a)$$

  Alternatively, we can use the more complicated expectimax form:

  $$\pi^*(s) = \arg_a \max \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

  The two steps would iterate until convergence, and like value iteration, policy iteration is optimal. Also, these equations are really policy *extraction*, in the sense that given values, we know how to get a policy.

In general, policy iteration seems to be a slightly better bet than value iteration, though I guess the latter could be simpler to implement in some cases since it only involves the value updating part?

If we do not know exactly what state we are in, we do not have a notion of s here, so we need to change our representation of the problem. This is the **Partially Observable Markov Decision Process** (POMDP) case. We augment the MDP with a *sensor model* $P(e \mid s)$ and treat states as *belief* states. In a discrete MDP with n states, the belief state vector b would be an n-dimensional vector with components representing the probabilities of being in a particular state. The belief state update for a particular component (state) s' looks a lot like an HMM update:

$$b(s') \propto P(e \mid s') \sum_{s'} P(s' \mid s, a) b(s)$$

The cycle is: we compute an action from b, see evidence e, compute a new belief state (using the above formula), then repeat. Fortunately, the optimal action for a POMDP only depends on the current belief state, allowing us to have our policy π be a mapping from a belief state to an action.

We can also map POMDPs to an observable MDP formula, by appropriately defining transition $P(b' \mid b, a)$ and reward $\rho(b)$ functions. For instance, $\rho(b) = \sum_s b(s)R(s)$. The optimal policy for this new MDP over the *belief* states is the *same* optimal policy over the original POMDP. This is good, since we've reduced this problem to an MDP, but we have to modify our earlier algorithms to handle continuous-valued state spaces. A key observation that will help: the value function $V(b)$ over belief states is piecewise linear and convex, because it is the maximum of a collection of hyperplanes. What a modified value iteration algorithm needs to do is to accumulate a set of possible plans, and for a given belief state, compute which of those has the optimal hyperplane. But this is too slow so in practice, people use online algorithms based on one-step lookahead, inspired by Dynamic Bayesian Networks.

To understand POMDPs well, consider simple MDPs with two states, so we can represent b as a scalar.

Reinforcement Learning

Now we switch to the *reinforcement learning* case. Before diving into the details, it is easiest to think of us in the same MDP framework, *except that* we have to learn our policy *online*, not offline. This is harder because we typically assume we do not know the transition probabilities ahead of time, or even the rewards. Unfortunately, this means we will try out different paths, and will sometimes fail (e.g., falling into the -1 pit in the gridworld example).

First, consider the **passive reinforcement** case, where we are given a fixed (possibly garbage) policy π and the only goal is to learn the values at each state, according to the Bellman equations.

Here's one way to get the values: learn the transitions P and the rewards R through trials, then immediately apply the policy evaluation step in policy iteration. As a reminder, this computes values (as in value iteration) but since the policy is fixed, we can solve the set of equations quickly. This technique, which Russell and Norvig seem to call **adaptive dynamic programming**, is part of the class of *model-based* learning techniques, because that assumes we have to compute a *model* of the world (i.e., the transitions and rewards). Here, our agent could play a series of actions from a variety of states ("restarting" when necessary). At the end, it will have data about the rewards it's gotten, along with the number of times it transitioned from state i to state j. To compute the transition function, we would convert the counts to the averages in a "Maximum Likelihood Estimate"-manner. The rewards are easier because we can see $R(s, a, s')$ immediately when we transition from s to s'.

In *model-free* learning, we do not bother to compute the transition model, but instead only compute the actual *values* of each state. There is a basic technique called **direct utility estimation** that falls in this category of methods. The idea here is that for each trial, we need to record the sum of discounted rewards for each state visited. After a series of trials, these values will average out and converge, albeit slowly since direct utility estimation does not take into account the Bellman Equations.

We need a better solution. Since we already have a policy, we would like to do some form of policy evaluation (explained earlier), but we do not have the transitions and rewards. We can approximate those with our samples instead, which leads to **Temporal Difference learning**. This is a model-free tactic where, after each transition (s, a, s', r), we update the values:

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\underbrace{[R(s, \pi(s), s') + \gamma V^{\pi}(s)]}_{\text{sample}}$$

In short, TD learning works by adjusting the utility estimates towards the ideal equilibrium as stated by the Bellman equations. But what if we want to get a new *policy*? TD learning does not learn the transition probabilities, so we switch to learning *Q-values*, since it is easier to extract actions from Q-values.

This now brings us to **active reinforcement** learning, where we have to *learn* an optimal policy by *choosing* actions. Clearly, there will be some tradeoffs between exploration and exploitation.

The solution here is an algorithm called **Q-Learning**, which iteratively computes Q-values:

$$Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha\underbrace{[R(s, a, s') + \gamma \max_{a'} Q(s', a')]}_{\text{sample}}$$

Notice how the sample here is slightly different than in TD learning. Each sample is computed from a (s, a, s', r) tuple.

The amazing thing is, so long as certain obvious assumptions hold, such as that we visit states sufficiently often, Q-Learning will converge to an optimal solution *even if* the actions we take are repeatedly suboptimal[4]! In the fire pit example in the CS 188 class, we could keep jumping in the pit, but so long as our actions are somewhat stochastic, we will eventually have some paths that end up in the lone beneficial spot, which will result in higher Q-values for those states and actions that lead towards that spot. (Think of each grid as having four Q-values.)

Even so, it would probably be wise to have good heuristics for deciding on what actions to take, which might make Q-Learning converge faster. The simplest (and worst) strategy: act according to a current policy, but with some small probability, we choose a random action. To improve this, use *exploration functions* that weigh the importance of actions and states based on whether

their values are “established” or not. This information propagates back so that we end up favoring actions that *lead* to unexplored areas. This means for some exploration function f, the action we pick at a step will be $\arg_{a'} \max f(Q(s', a'), N(s', a'))$ where N represents counts, and the new Q-Learning update is:

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha \underbrace{[R(s, a, s') + \gamma \max_{a'} f(Q(s', a'), N(s', a'))]}_{\text{sample}}$$

There is a close relative to Q-Learning called **SARSA**, State-Action-Reward-State-Action. The update is, following Russell and Norvig’s notation to have rewards be fixed to a state:

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha (R(s) + \gamma Q(s', a'))$$

What this means is that we already have an action a' chosen for the successor state s'. Q-Learning will choose the best action a', but in SARSA, it is fixed, and *then* we can update $Q(s, a)$. SARSA and Q-Learning (the first version here, not the second version with f) are the same for a greedy agent always picking the best action. When exploration happens, Q-Learning will attempt to pick the best action, but SARSA picks whatever is actually going to happen. This means Q-Learning does not pay attention to the policy, i.e., it is **off-policy**, but SARSA does, so it is **on-policy**. Here’s an example I thought of to clarify: the fire pit in CS 188 means that a policy that tells us to go in the pit will be bad, but if transitions are stochastic, then at *some point*, we will get to the good exit, hence Q-Learning will eventually learn that best value. But with SARSA, we do not get that opportunity because we will always have the next action fixed, so it is “more realistic” in some sense.

It is worth pointing out that all this previous discussion about learning all the $Q(s, a)$ values is really applicable only for the smallest of problems, because we cannot tabulate all those in real-life situations. Thus, we resort to **Approximate Q-Learning**. Here, we use feature-based representations by representing a state as a vector of features:

$$V(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

And the same for the Q-values:

$$Q(s, a) = w_1 f_1(s, a) + \cdots + w_n f_n(s, a)$$

We usually make things linear in terms of the features for simplicity. This has the additional benefit in that we can *generalize* to states we have not visited yet (in addition to the obvious compression benefits). Note that when we do the Q-Learning update here, we change the *weights*. Each weight update looks like the same kind of update we see in stochastic gradient descent.

Finally, the last major reinforcement learning topic is **policy search**. It actually seems like the policy iteration analogue: we only care about the policy, not the actual values themselves. Starting with an initial feature-based representation of the Q-states, we might consider tweaking the individual weights up and down and then evaluating the result of that representation to see if our rewards are higher. This may involve computing an expression for the expected reward, or running an agent in the world multiple times.

Old CS 188 Exam Questions

I found four interesting questions related to MDPs and reinforcement learning.

- Spring 2011, Question 4 ("Worst-Case Markov Decision Processes"). Now we measure the quality of a policy by its *worst-case utility*, or in other words, *what we are guaranteed to achieve*. This question is really intellectually simulating, as we have to consider the algorithm from a *minimax* perspective, rather than an *expectimax* perspective.

  For a state, we have:

  $$L^*(s) = \max_a \min_{s' \in C(s,a)} [R(s,a,s') + \gamma L^*(s')]$$

  So we pick the action that maximizes the following quantity: it is the discounted reward based on the *worst* possible state s' to which we *could* transition.

  There is a corresponding Q-value version:

  $$M^*(s,a) = R(s) + \gamma \min_{s' \in C(s,a)} \max_{a'} M^*(s',a')$$

  We are guaranteed $R(s)$, but have *already committed* to action a, so the adversary is free to send us to the *worst* possible subsequent state for us (as long as that transition has non-zero probability, of course), but then we are free to maximize over the next action.

  If we want to do M-Learning, then it is actually easier than standard Q-Learning because the environment has deterministic rewards and we only care about the minimax one, so if we ever observe a reward, we know that it's the correct one.

- Fall 2011, Question 4 (An un-named question). This one required us to actually perform some computations of value iteration and Q-learning. The only non-trivial thing about this was the jump from C, which could land in either D or E. I personally found it easiest to just run Value Iteration and try to reason about the Q-values later. The Q-learning *iterative* update is:

  $$Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

For the third part, the tricky thing was to remember that the Q-learning update will involve MAX-ing over previously computed Q-values.

- Fall 2012, Question 6 ("Reinforcement Learning"). This question makes clear the distinction between Q-Learning and SARSA. For part (a), they give us the formula for the Q-Learning agent, and ask if this will converge to the optimal set of Q-values even if the policy we're following is suboptimal. The answer is yes, since they also mention in the question several necessary requirements (e.g., that we visit all state-action pairs infinitely often). For part (b), the SARSA update, answer is *no*, we do not converge to the optimal Q-values, because SARSA will have Q-values for whatever policy is actually being executed. This policy that we converge to is the policy that follows π with probability 0.5, and uniformly random otherwise.

- Fall 2012, Question 7 ("Bandits"). This is a rather long question. Part (a) gives a nice review of MEU and VPI, and the remaining parts now deal with POMDPs over the two states of having bandits versus not having bandits. Hence, we represent the belief state as a single scalar b representing the probability we believe there *are* bandits. When we have to compute $b_{t+1} = P(s_{t+1} = +b)$ in the tree, we use the following formula:

$$P(s_{t+1} = +b) = \frac{P(e = +safe \mid s_{t+1} = +b)b_t}{P(e = +safe \mid s_{t+1} = +b)b_t + P(e = +safe \mid s_{t+1} = -b)(1 - b_t)}$$

  Notice that this is precisely the formula for updating the belief state that we discussed earlier in the notes, except that we have the transition function as the identity. By the way, it took me a while to figure this out, but the normalizing constant here needs to iterate through the set of belief states. We have a scalar b_t, but we really have to consider the case of there possibly being *no* bandits at all. A bit tricky here. The transition probability is also tricky, so it's easiest to think of it in terms of the problem: what is the probability of even reaching this node?

  Finally, they discuss some utility diagrams. As we discussed earlier, to find the overall value function, we take the maximum over all the possible lines. For the discounting questions, part (i) the point was that agents with larger γ values should be more concerned about the future, so they are more willing to try out short-term suboptimal paths. Thus, Agent A has a higher discount factor because it played the sub-optimal *Land* action one more time. In part (ii), there are three plots you can easily remove. Then it's just assigning three plots to three discount factors. A higher discount factor means we need to be more certain that we have bandits before we commit to making air shipments.

---

1. Here is where I get confused. Why are we having Berkeley's AI class use a different formulation than what R & N are using? I mean, Stuart Russell *is a faculty member here*, so why are we choosing the other way? For pedagogical purposes? ↩

2. In R & N's notation, we would use $\pi(s) = \arg_a \max \sum_s P(s, a, s')V^*(s)$. ↩

3. Alternatively, if that step is still too expensive, one can perform iterative updates as we would in value iteration for some number of steps, then switch to policy improvement. In this case, we would add ks as subscripts. ↩

4. But we do need to visit states often enough. In section 21.3 of R & N, they make it clear that at each time step (i.e., after each sample of state-action-rewards-successors), we *could* compute a model with whatever information we have, and then follow the optimal policy for that. But such a greedy agent is not good because it will never try out different states whose value may not be established. ↩

0 Comments **seitasplace** **1 Login**

Recommend 6 **Share** **Sort by Oldest**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON SEITASPLACE

These Aren't Your Father's Hearing Aids

3 comments • 10 months ago

Yongyi Chen — Thanks! So it looks like having a volume control is a very important thing to have, which I didn't know. My

IPython, Jupyter Notebooks, and matplotlib

2 comments • 2 years ago

Ricardo de Azambuja — Hi, just to add a new information: instead of "%matplotlib inline", if you have Matplotlib 1.5 or above, you can

Thoughts on Dale Carnegie's "How to Win Friends and Influence People"

1 comment • 2 months ago

Mike Langevin — Daniel, I read this book in high school for the first time and have read it a dozen times since. I find it to be like my "bible"

Deep Reinforcement Learning (CS 294-112) at Berkeley, Take Two

3 comments • 6 months ago

Daniel Seita — Um ... sorry ... it's going to come out in a few weeks ...

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Privacy**

Seita's Place

Seita's Place
seita@berkeley.edu

DanielTakeshi
(Never!)

This is my blog, where I have written over 250 articles on a variety of topics, most of which are about one of two major themes. The first is computer science, which is my area of specialty as a Ph.D. student at UC Berkeley. The second can be broadly categorized as "deafness," which relates to my experience and knowledge of being deaf.