Elementary Reinforcement Learning Techniques Illustrated through the 8-Puzzle

Trevor Bekolay, 6796723
umbekol0@cc.umanitoba.ca

Abstract

In machine learning, novel reinforcement learning techniques are based on three sets of elementary reinforcement learning techniques: dynamic programming, Monte Carlo methods, and temporal difference learning. In an effort to understand the internals of these techniques, and through that their strengths and weaknesses, this paper will select a representative algorithm from each set of techniques and illustrate its mechanism through the example of the 8-puzzle.

Table of Contents

1. Introduction

Recall, if you are able, a moment from an American television show, *The Simpsons*, in which a young Bart Simpson is tested alongside a hamster. Upon reaching for a cupcake and being shocked, the hamster recoils and avoids the cupcake, while the Simpson lad displays his tenacity by repeatedly reaching for said cupcake. The hamster receives negative reinforcement from his environment and, like any semi-intelligent being, chooses not to reach for the cupcake again, and in effect, learns from the experience. In computer science, reinforcement learning is a type of machine learning that deals with this same phenomenon: an agent learns how to act in an environment by reward and punishment. [Kaelbling et al., 1996] This is in contrast to supervised learning, in which an external supervisor provides examples to learn by; in general, agents in reinforcement learning learn through their own experience, and can do so in unknown environments. [Sutton and Barto, 1998]

The standard reinforcement learning (RL) model is concerned with an individual agent who is connected via perception and action to an environment. On each step of interaction, the agent perceives the current state of the environment and chooses an action. Based on that action, the environment changes to a new state and the agent receives a reinforcement signal, or reward. An agent's job is to maximize the sum of all rewards received in a set of interactions with the environment. [Kaelbling et al., 1996]

An agent that is so shortsighted that it greedily chooses the action with the highest immediate reward will not perform well in the long run. [Kaelbling et al., 1996] There are three methods of incorporating future rewards into an agent's decision. The first is finite horizon, in which an agent looks a certain number of steps in the future. The second is average-reward, in which the agent chooses rewards that will its long run average reward. The last is discounted infinite horizon, which looks arbitrarily far into the future, but weighs more immediate rewards higher by "discounting" future expected rewards. One can vary the discounting constant to control how near or far-sighted an agent is. [Kaelbling et al., 1996] The algorithms that will be discussed in this paper all use the discounted infinite horizon model; and indeed, this is typical of most RL problems. [Tsitsiklis and Van Roy, 2002] Notice that the reward for a particular state is not simply the immediate reward given for taking a certain action; this is called delayed reinforcement. [Sutton and Barto, 1998]

A complicated issue that is unique to RL is the trade-off between exploration and exploitation. Values that are known to produce relatively more reward must be exploited to obtain reward. However, agents must explore other actions to find those that will produce even more reward. [Sutton and Barto, 1998] A further discussion into this issue, as well as an example of one solution can be found in the section on Monte Carlo techniques.

The easiest way to model RL problems is to define the environment as a set of states, with a set of actions that can be performed on those states, and the rewards associated with those actions. This leads us to a model called the Markov Decision Process. [Mansour, 1999]

1-1. Markov Decision Process

A Markov Decision Process (MDP) consists of

- A set of states, S
- A set of actions, A
- A set of probabilities P, describing the probability of each possible next state. These are called transition probabilities.
- A set of rewards R, desciribing the expected reward returned from going from one state to the next by taking a certain action. [Sutton and Barto, 1998]

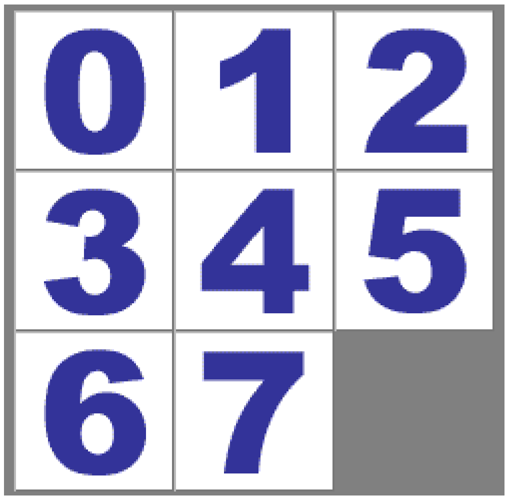
MDP's allow us to model the uncertainty in the outcome of actions, in the immediate rewards, and in the knowledge of the agent of the environment. Further, since MDP’s have been used in fields such as Operation Research and Control Theory, there already exists a rich literature to reference. [Mansour, 1999]

In general, RL deals with finite MDP's, where all states and actions are known, and states conform to the Markov Property. [Sutton and Barto, 1998] The Markov Property is upheld if each state accurately describes the environment and future possibilities, regardless of how that state was arrived at. This property be described as follows:

$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$ [Sutton and Barto, 1998] where t are time-steps.

So, with finite MDP’s, we can generate a list of states and the actions that can be performed at each state. Given any state s and action a, the probability and reward for the next state s' can be learned over time; this is how RL interacts with MDP’s. RL algorithms are developed to determine the optimal or near-optimal values for P and R. [Sutton, 1997]

1-2. MDP for the 8-Puzzle



The easiest way to visualize an MDP is to consider an example. The 8-puzzle is a sliding puzzle that consists of a grid of numbered squares with one square missing. We can see that there will be a finite number of states (configurations) for this problem, and at each state we have two to four possible actions (moves). It is easiest to visualize the actions from the perspective of the blank space. We will represent each state as three lists of three numbers, each list representing a row, with an underscore representing the blank space. Note that the 8-puzzle is a deterministic example, meaning that at state s, an action a will always produce the same next state s'. RL techniques work equally well for non-deterministic environments, in which the result of an action in a certain state is not known.

Fig. 1: A very small subsection of the 8-puzzle MDP

State	Action	Probability	Reward
(1, 4, 3) (6, _, 2) (7, 5, 0)	Up	0.3	23
	Down	0.15	7
	Left	0.25	14
	Right	0.3	25
(0, 1, 2) (3, 4, 5) (6, _, 7)	Up	0	-20
	Left	0	-20
	Right	1	100
(_, 3, 5) (1, 7, 6) (2, 4, 0)	Down	0.6	15
	Right	0.4	10

2. Reinforcement Learning Fundamentals

Regardless of which algorithm is being examined, there are some ideas and formulas that are essential to understanding how the elementary RL techniques operate.

2-1. Policies

Recall from the introduction that RL is concerned with agents, and environments. To further this, we add in time; an agent interacts with its environment at defined time-steps (t). For each time step, an agent perceives its state (s_t ε S), and selects from a set of actions (a_t ε A) that, on the next time step, produces some reward (r_{t+1}) and leaves the agent in some state s_{t+1}. [Sutton and Barto, 1998] An agent's job is to find a policy π that, given the state, selects actions to maximize reward. A policy is simply a mapping of states and actions to the probability that the agent will take that action in that state. [Kaelbling et al., 1996] We can see a policy in Fig.1. It is the set of all state-action pairs and the corresponding probability, normally represented as π(s,a).

2-2. Value Functions

RL algorithms depend on the ability to evaluate a given state. In general, a state's worth is based on the actions it can perform, and rewards that can be gained from taking those actions in that state. [Kaelbling et al., 1996] Functions for calculating a state's value are done with respect to a policy. [Sutton and Barto, 1998]

Value is represented as $V^\pi(s)$, which stands for the value of state s under policy π. In other words, what is the 'goodness' of being in state s given that we follow policy π. This function is represented formally as:

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s\right\},$$

where $E_\pi\{\}$ is the expected value given that the agent follows policy π, and t is any time step. R_t is the estimated reward that can be attained from the state, and when expanded, is the sum of the rewards from all future states, discounted by γ. [Sutton and Barto, 1998] γ is the discount constant, and lies in [0, 1]. [Kaelbling et al., 1996] This function, V^π, is called the state-value function for policy π. [Sutton and Barto, 1998]

Extending from V^π, we can also take actions into account and calculate the value of taking an action in a particular state. This is denoted as $Q^\pi(s, a)$. This function is represented formally as:

$$Q^\pi(s,a) = E_\pi\{R_t | s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s, a_t = a\right\}.$$

This function, Q^π, is called the action-value function for policy π. [Sutton and Barto, 1998]

These value functions lead to an important equation in RL called the Bellman equation. It is a recursive function that uses the values of successor states to calculate the value of the current state. Note that the value of any terminal state, if one exists, is always zero. [Peters, et al., 2003] The equation is represented formally as:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \right],$$ [Sutton and Barto, 1998]

Though this looks intimidating, if we examine each portion of the equation, it becomes easier to digest.

- $\gamma V^{\pi}(s')$ is the discounted value of the next state, s'.
- $R^{a}_{ss'}$ is the immediate reward obtained by taking action a at state s and moving to s'.
- $P^{a}_{ss'}$ is the transition probability of s' given s and a. This is multiplied by the inner brackets to determine the weighted value for a particular next state s'.
- $\Sigma_{s'}$ gives the sum of all weighted values for all possible successor states from state s, given action a. In other words, gives the total value of taking action a.
- $\pi(s,a)$ takes the previously explained sum and weighs it based on the probability of taking action a in state s based on policy π
- Σ_{a} sums up all possible actions in state s

Though slightly complicated, we see that it is possible to determine the value of any state based on the values of its possible successor states. Further, understanding this equation is essential for understanding the algorithms described in this paper. [Peters, et al., 2003] There also exist optimal state-value and action-value functions, called V*(s) and

Q*(s, a) respectively. [Sutton and Barto, 1998] They will be discussed in the dynamic programming section.

2-3. Policy Evaluation and Policy Improvement

Policy evaluation refers to how a RL algorithm evaluates the policy it has assumed at a certain point in time. This is usually done by calculating the state-value function ($V^{\pi}(s)$) for all states ε S, for a given policy π, though this is not necessarily the only way to evaluate a policy. [Peters, et al., 2003] This is sometimes known as the *prediction problem*. [Sutton and Barto, 1998]

Policy improvement poses the question; how do you optimize a policy, based on a policy evaluation? Greedy techniques are common, selecting the action for each state that maximizes return, but many different methods exist. The problem of finding an optimal policy is called the *control problem*. [Sutton and Barto, 1998]

Peters, et al. claim that all RL techniques can be described by these two steps, policy evaluation and policy improvement. [Peters, et al., 2003] We will examine these steps through techniques from three categories of RL techniques: dynamic programming, Monte Carlo methods and temporal difference learning.

3. Dynamic programming

Dynamic programming (DP) refers to a problem solving technique that has been used for decades to address an issue in naturally recursive problems, such as the Fibonacci sequence. The reason recursive solutions are often very inefficient is that any given input will result in many identical recursive calls. [Wagner, 1995]

Based on what we already know about the Bellman equation, we can see the usefulness of DP in the Bellman equation, which recursively calls $V^{\pi}(s)$. [Kaelbling et al., 1996] Using value estimates to compute other value estimates is called *bootstrapping*, and this plays an important role in differentiating and evaluating different RL algorithms. [Sutton and Barto, 1998] To examine this, however, we must first cover the Bellman optimality equations.

3.1 Bellman Optimality Equations

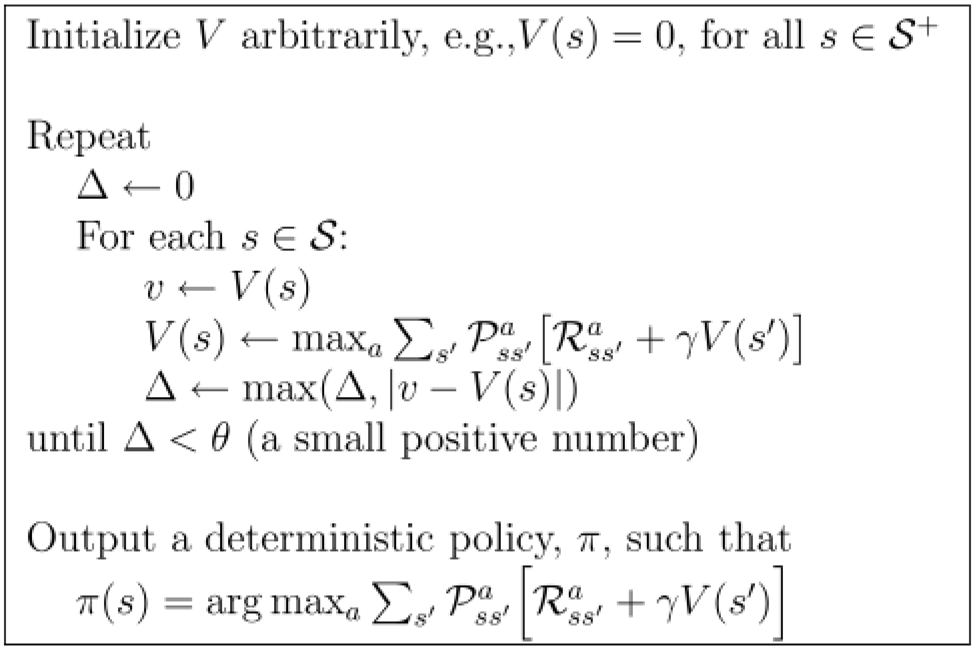
We will examine one of the two optimality functions, the optimal state-value function V*(s). It is closely related to the original Bellman's equation, except that is only interested in the best results in state s, not all results. So, $V^{\pi}(s)$ becomes:

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s') \right]$$ [Sutton and Barto, 1998]

This reduces the value of each state to the overall return of the best action for that state.

3.2 Value Iteration

Value iteration uses the optimal state-value formula as an update rule to determine the optimal value for each state. In each iteration of the value iteration algorithm, the currently policy is evaluated once, and updated. [Sutton and Barto, 1998]



```
Initialize $V$ arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
```

Fig 2. Pseudocode for Value Iteration from [Sutton and Barto, 1998]

Being the first RL algorithm we examine, it may seem overwhelming. Again, by looking at it piece-by-piece, we can fully understand the algorithm with little difficulty.

- After initialization, the main repeat-until loop iterates over every state in S.
- A new optimal value V(s) is calculated, as described in 3.1, for each state, and updates each state with its new optimal value.
- Determines the difference between its previous value and its current value.
- Iterates until the largest change in the value of a state in S is below some threshold θ.
- A policy π is created, mapping the optimal action a in A to each state s in S.

3.2 Value Iteration on the 8-puzzle

Since the 8-puzzle is a finite MDP, we can iterate over each state, and eventually determine the optimal policy. Let us examine the algorithm for one state. Recall Fig.1, a small subsection of the MDP for the 8-puzzle. Let's use one of the states examined there to go through part of the value iteration algorithm. Suppose this is the first iteration (i.e., all V(s) are initialized to 0)

State	**Old V(s)**	**Possible action-values**	**New V(s)**
(0, 1, 2) (3, 4, 5) (6, _, 7)	0	Up : R = -20	100
		Left: R = -20	
		Right: R = 100	

This is an obvious example; moving to any state other than the goal state results in negative reinforcement. However, it serves as an overly simple example of what value iteration must for each state in S.

In the 8-puzzle, there are 8!/2, or 20 160 states. There are a maximum of four actions per state, usually less. Even if there were 4 actions per state, we would only have around 80 000 state-action pairs. If around 1 000 000 state-action pairs can be considered in a second, then value iteration should complete in a few seconds. However, consider the 15-puzzle. There are 16!/2 or 1.05 x 10^{13} states, with up to 4 actions per state. Clearly, though value iteration does eventually give you an optimal solution, it does not scale well for problems with large state or action spaces.

3.3 Dynamic Programming – Pros and Cons

Dynamic programming techniques are limited in usefulness due to the requirement of a perfect model of environment as an MDP and, as we saw in examining the 15-puzzle, the requirement to calculate values for every state is prohibitively expensive in problems with large state or action spaces. [Kaelbling et al., 1996] On the plus side, it changes estimated values as it iterates through the state space, enabling it to use improved estimates quickly. Further, given the perfect model that it requires, it does converge on an optimal solution in relatively few iterations. [Sutton and Barto, 1998]

4. Monte Carlo Methods

Monte Carlo (MC) methods describe those that change values and policies by experience instead of computing values for each state. Because they learn based on past experience, these methods can learn a behavior based on very little initial information about the environment. [Sutton and Barto, 1998] However, a MC method can easily become biased by stochastically choosing a decent action near the start and sticking with it. This is where the issue of exploration vs. exploitation first rears its head. [Sutton and Barto, 1998]

4.1 Exploration vs. Exploitation

In DP, every state in S is examined on every iteration of the algorithm. MC methods, on the other hand, use experience to gather information. If a MC algorithm simply chooses the best action in a state based on previous experience, some actions will never be explored even though they may be better, or even optimal. [Sutton and Barto, 1998]

Exploration is achieved in MC algorithms by the addition of a constant, ε. ε is the probability of exploration; that is, the probability that the algorithm will choose an action other than that which is determined to be the best for a particular state. If ε is 0, there is no exploration, and if ε is 1, the algorithm will always choose a random action. [Sutton and Barto, 1998]

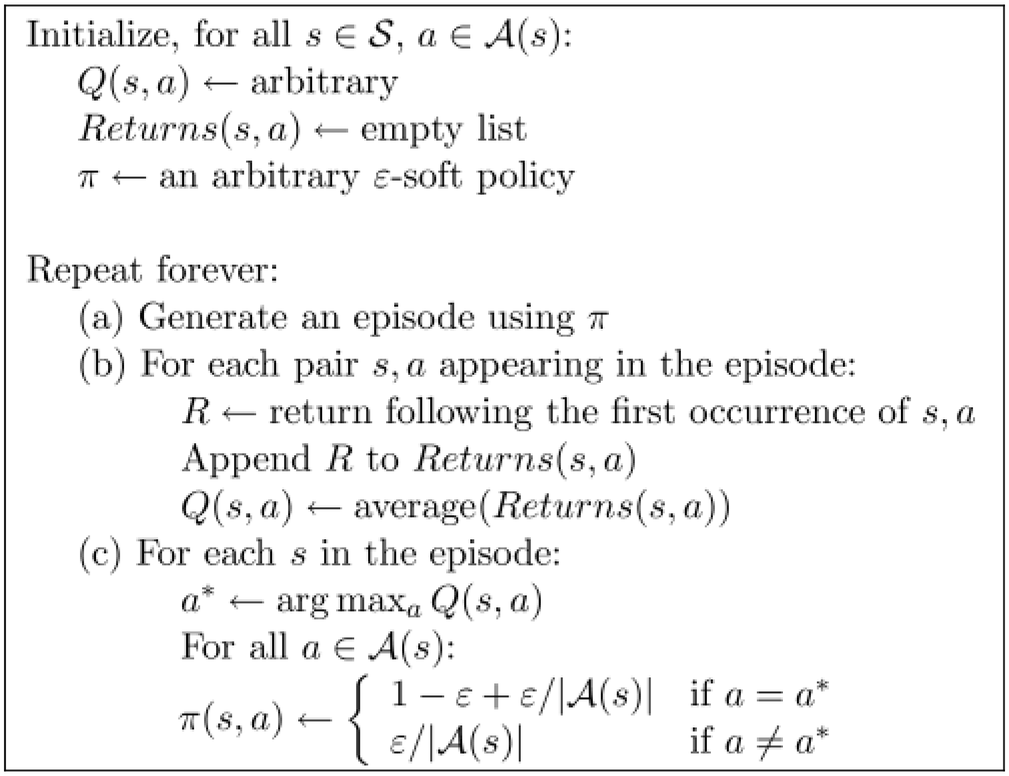
A policy is deemed ε-soft if $\pi(s, a) \geq \varepsilon$ / (# of actions for s). In ε-greedy policies, the action with the best estimated value is chosen most of the time, and the rest of time a random non-greedy action is chosen. [Sutton and Barto, 1998] In each state s in S, the action with the best estimated value is known as a*. ε-greedy policies can be formally defined as:

- if $a = a^*$, $\pi(s, a) = 1 - \varepsilon + (\varepsilon$ / (#actions in s))
- if $a \neq a^*$, $\pi(s, a) = \varepsilon$ / (#actions in s) [Sutton and Barto, 1998]

4.2 ε-soft on-policy Monte Carlo control

We will examine an explorative on-policy algorithm for simplicity's sake. On-policy refers to RL methods that improve and evaluate the same policy that it uses to gather experiential information. Conversely, off-policy methods consider experience that is obtained from policies other than the one being modified. [Kaelbling et al., 1996]

An episode is a run-through of a problem given some starting state s and using policy π. Since MC techniques are based on experience that is, actual reward values, the total reward for any given state-action pair cannot be determined until the algorithm reaches a terminal state. Therefore, MC techniques can only be used on problems that can be expressed in episodes that terminate regardless of actions taken. [Sutton and Barto, 1998] It is common practice in MC algorithms, especially elementary ones, to exclude the first 'visit' to each state that occurs in an episode. [Sutton and Barto, 1998]



```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each pair $s,a$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s,a$
            Append $R$ to $Returns(s,a)$
            $Q(s,a) \leftarrow$ average$(Returns(s,a))$
    (c) For each $s$ in the episode:
            $a^* \leftarrow \arg\max_a Q(s,a)$
            For all $a \in \mathcal{A}(s)$:
            $\pi(s,a) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
```

Fig 3. Pseudocode for ε-soft on-policy Monte Carlo control from [Sutton and Barto, 1998]

Unlike the complicated function that we used in value iteration, in MC we simply average out all of the returns of a state-action pair to get the action-value. This algorithm is generally easy to follow step-by-step, as will be shown with the 8-puzzle.

4.4 ε-soft on-policy Monte Carlo control on the 8-puzzle

Let’s try running this algorithms for ε = 0.6.

Part (a): Consider an episode for the 8-puzzle.

- Starting state: (1, 0, 3) (_, 7, 6) (2, 5, 4); Action: Up; Returns 15
- … some sequence of moves …
- State: (1, 0, 3) (_, 7, 6) (2, 5, 4); Action: Up; Returns 25
- …some sequence of moves …
- State: (1, 0, 3) (_, 7, 6) (2, 5, 4); Action: Up; Returns 35
- …some sequence of moves ending in a terminal state.

Assume that these were the only moves made from this sate.

Part (b): The first occurrence of the ([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) pair is ignored.

The second occurrence results in Returns([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) now containing an entry, 25, and Q([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) is now also 25.

The third occurrence results in Returns([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) containing the list 25, 35 and Q([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) is 30.

Part (c): For state (1, 0, 3) (_, 7, 6) (2, 5, 4), a* is Up (Right and Down had no value). Since ε = 0.6, we end up with

π([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Up) = 1 - 0.6 + 0.6 / 3 = 0.6

π([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Right) = 0.6 / 3 = 0.2

π([(1, 0, 3) (_, 7, 6) (2, 5, 4)], Down) = 0.6 / 3 = 0.2.

4.5 Monte Carlo – Pros and Cons

The main feature that makes MC techniques attractive is that they do not require a perfect model of the environment, nor do they suffer from the problems associated with large state or action spaces in DP. [Sutton and Barto, 1998] Also, as we have seen in the 8-puzzle example, the algorithm is easy to follow, and the exploration rate can easily be varied simply by changing ε.

On the other hand, MC techniques are limited to problems that can be expressed in episodes that terminate regardless of actions taken. [Sutton and Barto, 1998] Depending on how one looks at it, that limitation could make the 8-puzzle unfit for MC methods, as it is easy to get caught up in a loop and never terminate.

Monte Carlo methods are relatively new in the context of RL, and because of that, it is not yet clear whether or not they will always converge on an optimal solution. [Sutton and Barto, 1998] Whether or not it does, the episode-by-episode nature of MC techniques means that DP will almost always reach a better solution in less iterations; it just may take far more actual time to get there. [Sutton and Barto, 1998]

5. Temporal Difference

Temporal difference (TD) learning techniques strive to fuse DP and MC by taking the good points of both. Like MC, TD techniques learn from experience, and thus do not require a perfect model of the environment. Like DP, TD techniques bootstrap; that is, they update estimated values based on other estimated values, so they do not have to wait until the end of an episode to update the policy. [Sutton and Barto, 1998] The key to this (as the name indicates) is calculating the difference between time-steps.

5.1 Error

In any general problem in which one updates an estimated value, it is much faster to use the previously estimated value to calculate a new estimate than to recalculate the whole function. A general formula for this kind of performance optimization is

NewEstimate = OldEstimate + StepSize[Target - OldEstimate]

The [Target - OldEstimate] portion is known as the error of the old estimate. [Kaelbling et al., 1996] Though it was not explicitly discussed, this concept is inherent in MC. Consider a simple MC method. We can represent the updating of a state-value function at time t as:

$V(s_t) = V(s_t) + \alpha[R_t - V(s_t)]$ where R(t) is the actual return following time t, and α is a constant step-size parameter. [Kaelbling et al., 1996] Since R_t is not known until the episode reaches a terminal state, this update function only occurs at the end of an episode.

The simplest TD method, TD(0), is built on this idea, but employs bootstrapping to update the value during an episode. The update function is:

$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$ [Sutton and Barto, 1998]

The only difference in the two functions is the target, yet since the TD method bases its target on a value that can be estimated on-line, we are able to learn step-by-step instead of episode-by-episode. [Sutton and Barto, 1998]

5.2 One Step Q-learning

The moniker Q-learning encompasses the most well known set of RL algorithms. Though Q-learning deals with the action-value function (hence the name), the update function is still similar to that of TD(0): [Sutton and Barto, 1998]

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big].$$

This is considered an off-policy technique, because this function will work regardless of the policy that was used for initially determine Q(s, a), because it uses the best action, chosen by the previous policy, as the target. [Tsitsiklis, 1994]

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Fig 3. Pseudocode for One Step Q-Learning from [Sutton and Barto, 1998]

Aside from the action-value function described above, this algorithm is also relatively easy to follow, especially if one is familiar with the DP and MC algorithms. In fact, it is easy to see from this pseudo-code how closely related all three methods are, despite their different strengths and weaknesses. [Sutton and Barto, 1998]

5.3 One Step Q-Learning on the 8-Puzzle

As with the MC technique, let us examine one iteration of the algorithm using our 8-puzzle example. This is made much easier, since this algorithm iterates on a step-by-step basis instead of episode-by-episode. We will use $\alpha = 0.1$ and $\gamma = 0.9$.

- We are in state [(0, 1, 2) (3, 5, 6) (4, _, 7)]

- Our current policy instructs us to take action Up, and the value of Q([(0, 1, 2) (3, 5, 6) (4, _, 7)], Up) is currently 40. This gives us a reward of 35, and we end up in state [(0, 1, 2) (3, _, 6) (4, 5, 7)]

- We update the estimated action-value pair Q([(0, 1, 2) (3, 5, 6) (4, _, 7)], Up) based on information we already have about the next state; that is, the best action at the next state is Left, and Q([(0, 1, 2) (3, _, 6) (4, 5, 7)], Left) = 45. This gives:

- Q([(0, 1, 2) (3, 5, 6) (4, _, 7)], Up) = 40 + 0.1[35 + (0.9)(45) – 40] = 43.55

- The new value for Q([(0, 1, 2) (3, 5, 6) (4, _, 7)], Up) is 43.55, and we continue for state [(0, 1, 2) (3, _, 6) (4, 5, 7)]

5.4 Temporal Difference – Pros and Cons

As has been discussed already, TD encompasses the good points of both DP and MC: it uses bootstrapping to compute values step-by-step based on experience, which means that it can function in unknown environments. Further, unlike MC, Q-learning has been proven to converge to the optimal solution. [Tsitsiklis, 1994]

6. Conclusion

Through examining the theory behind each of the elementary RL algorithms, and reinforcing (pun intended) that knowledge with the example of the 8-puzzle, the strengths and weaknesses of each method became very obvious. While it seems all too clear that temporal difference learning is the clear choice for most general purpose RL problems, it should be noted that there exist some domains in which DP or MC techniques will present you with a solution nearer to optimal quicker, whether it be in iterations or real time. [Sutton and Barto, 1998] In general, however, TD techniques encompass the strengths of both DP and MC techniques, while getting rid of most of the drawbacks. This explains why most novel RL techniques are based on the ideas presented in TD methods.

RL is still a relatively young area of AI, and because of its ability to formulate behavior in unknown environments, it is being actively researched. Despite that, most new RL techniques are just refinements or variants of the three discussed in this paper. [Kaelbling et al., 1996] Understand these techniques fully unlocks the door to understanding far more complicated and interesting ways for machines to learn.

Bibliography

[Kaelbling et al., 1996] Kaelbling, Leslie Pack, Littman, Michael L., and Moore, Andrew W., "Reinforcement Learning: A Survey", in *Journal of Artificial Intelligence Research*, Vol. 4, Jan., 1996, pp. 237-285.

[Mansour, 1999] Mansour, Yishay, “Reinforcement Learning and Mistake Bounded Algorithms”, in *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, July, 1999, pp. 183-192

[Peters, et al., 2003] Peters, Jan, Vijayakumar, Sethu, Schaal, Stefan, "Reinforcement Learning for Humanoid Robotics", in *Humanoids2003: Third IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, Germany, Sept., 2003, 20 pp.

[Sutton, 1997] Sutton, Richard S., "On the significance of Markov decision processes", in W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud (Eds.), *Artificial Neural Networks - ICANN'97*, 1997, pp. 273-282.

[Sutton and Barto, 1998] Sutton, Richard S., and Barto, Andrew G., *Reinforcement Learning*, MIT Press, Cambridge, Mass., 1998, 322 pp.

[Tsitsiklis, 1994] Tsitsiklis, John N., "Asynchronous Stochasic Approximation and Q-Learning", in *Machine Learning*, Vol. 16, 1994, pp. 185-202

[Tsitsiklis and Van Roy, 2002] Tsitsiklis, John N., and Van Roy, Benjamin, "On Average Versus Discounted Reward Temporal-Difference Learning", in *Machine Learning*, Vol. 49, 2002, pp. 179-191

[Wagner, 1995] Wagner, David B., "Dynamic Programming", in *The Mathematica Journal*, Vol. 5, Issue 4, Fall, 1995, pp. 42-51