Reinforcement Learning: Model-free

Chris R. Sims
Department of Brain & Cognitive Sciences
University of Rochester
Rochester, NY 14627, USA

July 24, 2012

Reference: Much of the material in this note is from Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press. An electronic copy of the book is freely available at `http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html`.

1. What is reinforcement learning?

Simply put, reinforcement learning (RL) is a term used to indicate a large family of different algorithms that all share two key properties. First, the objective of RL is to *learn* appropriate behavior through trial-and-error experience in a task. Second, in RL, the feedback available to the learning agent is restricted to a *reward signal* that indicates how well the agent is behaving, but does not indicate specifically how the agent could improve its behavior. For example, consider writing an essay for a course and receiving a numerical score in the range 0–100. If your score is less-than-perfect, you know that your performance could be improved upon, but the feedback itself doesn't indicate specifically *how* your essay should have been different. In more complex cases, optimal behavior will generally require numerous separate decisions, and there may be delayed or missing reward signals. For example, one can imagine trying to teach a computer to play checkers by providing a positive reward signal every time it wins a game, and a negative (penalty) signal every time it loses. In this case, each individual action (moving a piece) is not rewarded, and the only reward signal is provided at the end of a game. Learning how to improve behavior given this limited type of feedback is both the goal and challenge facing all RL algorithms.

RL

reward signal

2. Formal elements of reinforcement learning

More formally, the important elements of RL can be described using the framework of Markov decision processes (MDPs). An MDP is described by four components:

Markov decision process

- $\mathcal{S}$: The set of distinct states, s, that describe the environment. In the game of checkers, the state could be defined as the current configuration of pieces on the board.
- $\mathcal{A}$: The set of actions, a, available to the agent in each state.
- $\mathcal{P}^a_{s,s'} = \mathrm{P}(s' \mid s, a)$: The *transition probability* of moving from state s to s', given that action a is selected.
- $\mathcal{R}^a_{s,s'} = \mathrm{E}[r \mid s, a]$: The expected immediate reward associated with taking action a in state s and transitioning to state s'.

With these four components defined, the goal of RL is to learn an optimal behavioral *policy* function, $\pi(s, a)$, which specifies the probability of selecting action a in state s, for all states and actions. An optimal policy is one that maximizes the expected total *return*. In one-step decision tasks, the return is simply the immediate reward signal. In more complex tasks, the return is defined as the sum of individual reward signals obtained over the course of behavior. In some cases, one might value immediate rewards more than rewards far in the future. To incorporate this possibility, one can define a *discounted return*, starting from time step t as a weighted sum of future reward signals:

policy

return

discounted return

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=1}^{\infty} \gamma^k r_{t+k}, \tag{1}$$

where $0 \leq \gamma \leq 1$ determines the rate at which future rewards are discounted relative to immediate outcomes. Discounted returns are especially useful in continuing tasks that do not have a natural end-point, as in this case the total (un-discounted) return may be infinite.

One challenge in RL is that it is often not obvious how best to define the state of the environment. In simple games like checkers, the configuration of pieces on the board seems like an obvious choice. But this is not the only possibility. One could also include the entire history of previous moves in the definition of the state of the game. This more elaborate definition would distinguish between two board configurations that have an identical arrangement of pieces, but were arrived at via different sequences of moves. An important concept in RL is known as the *Markov property*. A particular definition of the state $\mathcal{S}$ satisfies the Markov property if and only if the transition and reward probabilities, $\mathcal{P}$ and $\mathcal{R}$, depend on the state s and action a at time t, but are independent of the past sequence of observed states and previous actions. In other words, if including the entire history of past states and actions does not provide any additional useful information, then the state definition meets the Markov property.

Markov property

The Markov property is an important concept in reinforcement learning for two reasons. First, including unnecessary variables in the definition of the state can severely impede learning, and the Markov property defines an important criterion for judging whether such extraneous information can be safely discarded. In checkers, the current configuration of pieces on the board does satisfy the Markov property, and so it is not necessary to remember the entire history of past moves. Second, for many algorithms developed in RL, the proof of their convergence or optimality depends on the assumption of the Markov property. These algorithms can exhibit unexpected or suboptimal behavior if applied to decision environments that do not satisfy the Markov property.

2.1 Value functions

Intuitively, it can be seen that some states of the environment are more advantageous than others. Having many pieces remaining in a game of checkers is generally preferable to only having one or two pieces left. However, a skilled player may still be able to win a game despite having fewer pieces on the board. Thus, the value of a particular state also depends on the policy, $\pi(s, a)$. Skilled players possess more successful policies than novice players, and so will generally assign higher values to each state. More formally, the value of a particular state, and given a particular behavioral policy, can be defined recursively as

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{s,s'} \left[\mathcal{R}^a_{s,s'} + \gamma V^\pi(s') \right]. \tag{2}$$

This state value function, also known as the Bellman equation after its discoverer Richard Bellman,

Bellman equation

says that the expected value of a particular state s is defined by the immediate reward that comes from acting in that state, plus the discounted value of the subsequent state s' that is reached. The equation is recursive in that the value of any one state is defined by reference to the value of other states. The two summations are due to the fact that it is necessary to average over two potential sources of variability: the stochastic nature of state transitions as encapsulated by $\mathscr{P}$, and the possibly stochastic nature of the policy $\pi(s,a)$.

It is also possible to define the *Bellman optimality equation*, which is simply the value function associated with the best-possible policy:

$$\begin{aligned} V^\star(s) &= \max_\pi V^\pi(s) \\ &= \max_a \sum_{s'} \mathscr{P}^a_{s,s'} \left[\mathscr{R}^a_{s,s'} + \gamma V^\star(s') \right]. \end{aligned} \tag{3}$$

The Bellman optimality equation (actually a system of equations, one for each state) is a key construct in all reinforcement learning algorithms, and so it is worth considering it in detail. First, if one has access to the value for each state, then it is a simple matter to derive the corresponding optimal policy: One simply selects the action that maximizes the expected value, determined via the right hand side of the equation. In essence, existence of the Bellman optimality equation turns the difficult problem of optimal decision making into a much simpler local search problem. For small state spaces, this system of equations can also be solved analytically using techniques such as dynamic programming to obtain both the value function and the optimal policy.

Additionally, the Bellman optimality equation can be viewed as a consistency requirement: the left hand side of the equation must equal the right hand side, for all states. Any behavior policy that satisfies this mathematical equation is guaranteed to be an optimal policy (optimal policies are not necessarily unique, however). Relying on this important property, equation (3) forms the basis of nearly all RL algorithms. In a nutshell, the goal of reinforcement learning is to improve performance by progressively adjusting a suboptimal policy to reduce the inconsistency between the left and right sides of this equation. Such incremental improvement algorithms are often tractable even for large state spaces where dynamic programming solutions are infeasible. In fact, all reinforcement learning algorithms can be considered as approximate solution methods to the Bellman optimality equation.

approximate solution methods

3. Reinforcement learning algorithms

This section briefly discusses several reinforcement learning algorithms. This examination is by no means exhaustive; for more details refer to (Sutton & Barto, 1998), full reference given at the beginning of this document.

3.1 TD learning

From the Bellman equations, we know that the value for each state is the average immediate reward plus the discounted value of the subsequent state. If we do not know the state values, then a simple approach is to start with an initial guess for $V(s)$, and gradually improve the estimate as we gain more experience. Suppose we start in state s, and select action a according to some policy $\pi(s,a)$. We then observe an immediate reward r and arrive at state s'. The observed quantity $r+\gamma V(s')$ is actually a Monte Carlo sample from the Bellman equation for policy π. If we repeated the same process many times and averaged the results, we would exactly obtain the right hand side of equation (2). Using this fact, we

can define an incremental learning rule for updating $V(s)$ so that it more closely reflects the true state value:

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]. \tag{4}$$

The quantity $r + \gamma V(s') - V(s)$ is defined as the *temporal difference error*, often abbreviated as TD error. This update equation gives rise to perhaps the simplest reinforcement learning algorithm, known as TD learning. In this equation, α is a constant *learning rate* parameter in the range $0 \leq \alpha \leq 1$. It is desirable to set $\alpha < 1$, due to the fact that rewards and state transitions are potentially stochastic, as well as the fact that we expect that the values for all states will change over the course of learning. Note that this learning rule relies on a bootstrapping process: The value of each state is updated on the basis of the estimated value of subsequent states. Perhaps surprisingly, even if we start with an incorrect guess for the value of each state, this learning rule will eventually converge on the true value function (exact convergence requires that the learning rate gradually decreases and approaches zero).

temporal difference error

learning rate

3.2 Q learning

While TD learning will eventually learn the value function for a given policy π, ideally we would like to improve a policy at the same time that we learn its associated value function. This is not easily accomplished through TD learning, because the estimated value for each state is based on the potentially suboptimal policy π. One way to get around this is to learn an *action-value function* $Q(s, a)$ rather than the value function $V(s)$. The function $Q(s, a)$ indicates the value of taking action a in state s. Consider the following update rule for the action-value function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]. \tag{5}$$

As before, the updates are based on a bootstrap process. However, the max operator in this equation implies that the update at each step is not based on the current (and potentially suboptimal) policy, but rather on the best possible action one could take in the resulting state. Note also the similarity between this update rule and the Bellman optimality equation, (3). This *Q-learning* update rule is actually based on a Monte Carlo sample from the optimal value function. At each step, behavior approximating the optimal policy can be determined by selecting the action with the highest $Q(s, a)$ value. Perhaps remarkably, Q-learning will still converge on the optimal state-action values even if the actions selected on each step follow a suboptimal policy. Conditions for convergence require that in the limit of infinite task experience, all state-action pairs are explored an infinite number of times, and that the learning rate gradually decreases over time. Because of its ability to learn the optimal policy while following another policy for action selection, Q-learning is known as an *off-policy* learning algorithm. The basic Q-learning algorithm is presented in Algorithm (1).

off-policy learning

To demonstrate the performance of Q-learning, consider the cave exploration task in Figure 1a. The goal is to navigate from the starting position at the top of the maze to the treasure (located at the bottom of the maze) while incurring as little cost as possible. Crossing each open square costs 1 point, while crossing water (blue squares) costs 100 points. The agent can move in the four cardinal directions, unless blocked by walls. In addition, with probability = 0.2, the agent becomes disoriented and takes a step in a random direction. The action-value function $Q(s, a)$ was initialized to zero for all states and actions, and on each step the agent selected actions randomly. After 1,000 training episodes, Q-learning was able to infer the optimal policy (illustrated in Figure 1b). Note that the optimal path through the maze is not

Algorithm 1 Q-learning.

Inputs:

$Q(s, a)$: State-action values, initialized arbitrarily

s_0: Initial state

Algorithm:

$s \leftarrow s_0$

Repeat

Choose a for the current state s based on Q(s,a)

Take action a, observe reward r and resulting state s'

$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$

$s \leftarrow s'$

Until the end of the learning episode

Return $Q(s, a)$, the updated state-action values

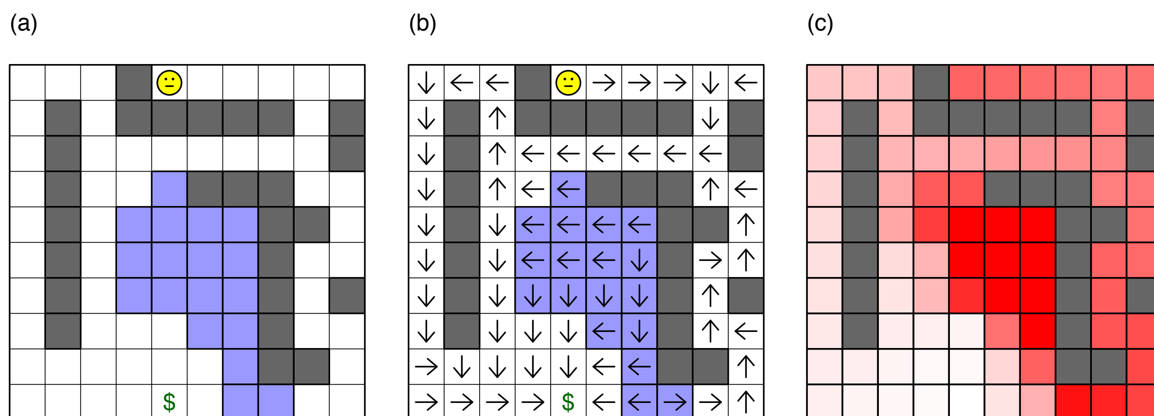

Figure 1: (a) The cave exploration task. Crossing each open square costs 1 point; crossing water (blue squares) costs 100 points. (b) Optimal policy learned by Q-learning after 1,000 training episodes. (c) Optimal value function, defined according to equation 3. Darker shading indicates states with lower value.

the shortest path, but rather one that minimizes expected cost, taking into account the probability of accidentally stepping into a water square. Figure 1c plots the optimal value function for this task, learned via Q-learning, showing the minimum expected cost-to-go for each state, assuming the optimal policy is followed.

3.3 Action selection and the exploration-exploitation dilemma

It seems intuitive that on each decision, one should always select the action that has the highest expected value. But is this really a good idea? In the face of uncertainty regarding the structure of a task, optimal performance requires balancing exploration of the task with actions that exploit current knowledge. In most cases, exploration requires selecting actions that have lower expected value according to current knowledge. In the general case of Markov decision processes, the optimal balance between exploration and exploitation is an intractable problem to solve. For this reason, much of RL relies on heuristic or

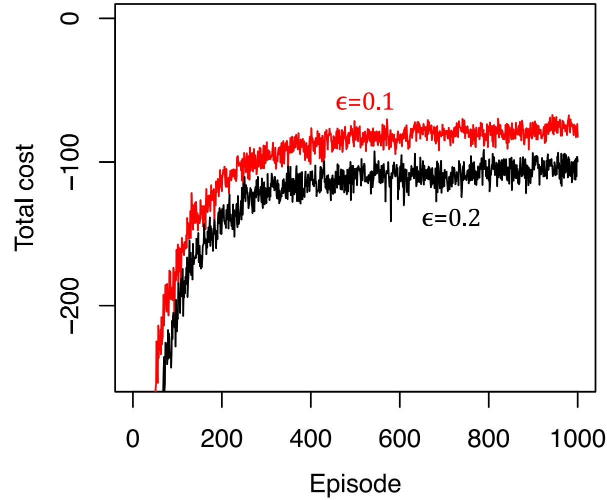

Figure 2: Average cost per episode over the course of learning. Exploration is based on ϵ-greedy action selection, using two different values of the parameter ϵ. Each curve is based on the average performance of 1,000 simulated agents.

approximate solutions to this tradeoff.

One simple approach to balancing exploration and exploitation is to occasionally select actions with sub-optimal value. For example, one can choose the optimal action with probability $(1 - \epsilon)$, and with probability ϵ choose a random action. This approach, known as epsilon-greedy action selection, is simple to implement and works well in many cases. Figure 2 shows the learning performance, defined in terms of the total cost incurred on each episode, for the cave exploration task using two diffent values of ϵ.

epsilon-greedy action selection

One disadvantage of ϵ-greedy action selection is that it does not take into account the utility of actions during exploration. An alternative approach, known as softmax action selection, chooses actions relative to their value, so that actions with extremely low utility are less likely to be chosen. Specifically, one can define a noise parameter τ, and choose an action a with probability

softmax action selection

$$P(a) = \frac{\exp\left(Q(s,a)/\tau\right)}{\sum_{i=1}^{n} \exp\left(Q(s,a_i)/\tau\right)}. \tag{6}$$

When the noise parameter is close to zero, this equation selects the action with the highest utility with probability 1. As the noise approaches infinity, actions are selected randomly, without regard for their utility. The name 'softmax' derives from the fact that the approach usually, but not always, selects the action with maximal utility.

3.4 Eligibility traces

One notable feature of the algorithms presented so far is that the occurrence of a reward only impacts the value of the state immediately preceding the reward. That is to say, utility updates are only based on the temporal difference error between a given state, and the next state that is reached after executing some action. This has the undesirable consequence that in large state spaces, learning can be extremely slow. Intuitively, one might imagine that receiving a reward should increase the utility of all previous actions leading up to the reward, and not just the most recent action.

One way of incorporating this intuition into the learning algorithm is to maintain a trace, or history, of each state encountered during a learning episode. When a state is encountered with better-than-anticipated value, (a positive TD error), the values of all previous states are increased. Similarly, values are decreased when the agent encounters unexpected penalties or rewards that are lower than expected (negative TD error). To increase the generality of this approach, one can also allow the 'eligibility' of past states to decay over time, so that changes in value are larger for states that were more recently encountered. In particular, one can define an *eligibility trace* $e(s)$ that indicates the extent to which the value for state s should be modified based on the current TD error. Each time a state is encountered, its eligibility is reset to 1. For all other states, the eligibility is multiplied by $0 \leq \lambda \leq 1$. When $\lambda = 0$, this approach is identical to the standard, 'one-step' update rules already encountered.

eligibility trace

Algorithm 2 TD(λ) learning

Inputs:
- $V(s)$:Value function, initialized arbitrarily
- $\pi(s, a)$: Policy to be evaluated
- s_0: Initial state

Algorithm:
- $s \leftarrow s_0$
- $e(s) \leftarrow 0$
- **Repeat**
  - Choose a defined by the policy $\pi(s, a)$
  - Take action a, observe reward r and resulting state s'
  - $\delta \leftarrow r + \gamma V(s') - V(s)$ // δ = temporal difference error
  - $e(s) \leftarrow 1$
  - **For** all s with $e(s) > 0$
    - $V(s) \leftarrow V(s) + \alpha\delta e(s)$
    - $e(s) \leftarrow \gamma\lambda e(s)$
  - $s \leftarrow s'$
- **Until** the end of the learning episode
- **Return** $Q(s, a)$, the updated state-action values

Algorithm 2 presents a basic 'TD(λ)' learning algorithm that incorporates eligibility traces. In general, there is no optimal setting of the parameter λ for all tasks. Many variants of eligibility traces have also been considered. One possibility is to increment the eligibility of each state, rather than reset its eligibility trace, each time it is encountered. Eligibility traces have also been incorporated in policy learning algorithms such as Q-learning. One complication is that for off-policy learning algorithms such as Q-learning, state values should not be influenced by future exploratory actions, but rather only by the rewards observed while following the optimal policy. A simple modification of the basic Q-learning algorithm is to define an eligibility trace $e(s, a)$ for each state-action pair, updating them as in the case of the TD(λ) algorithm. But whenever exploratory (or suboptimal) actions are selected, the eligibility traces for all states are reset to zero. This leads to the algorithm known as Watkins' Q(λ), named after the researcher who first proposed it. Note that the material presented here only touches the surface of the large field of reinforcement learning. Readers are encouraged to consult (Sutton & Barto, 1998) for a more comprehensive introduction.

Q(λ)