Introduction to Reinforcement Learning

Sunghee Yun

October 5, 2019

Contents

List of Tables

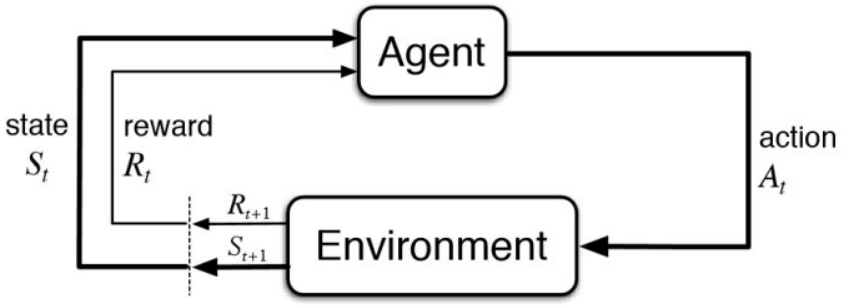

Figure 1: The agent-environment interaction in a Markov decision process.

The reinforcement learning is a machine learning where an agent learns how to take actions to achieve a goal by maximizing cumulative reward while interacting with environment. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

It differs from supervised learning in that labeled input and output pairs need not be presented (and sub-optimal actions need not be explicitly corrected). Instead the focus is finding a balance between exploration of uncharted territory and exploitation of current knowledge. It is much more focused on goal-directed learning from interaction than other approaches to machine learning.

In the following sections, we introduce finite Markov decision process (MDP) and three typical methods to solve reinforcement learning problems. All these methods are called tabular solution methods because they need to store values for all the states or all the state-action pairs (that have been visited).

1 Finite Markov decision processes

We introduce the formal problem of finite Markov decision processes (MDPs), which we try to solve. This problem involves evaluative feedback, but also an associative aspect, *i.e.*, choosing different actions in different situations. MDP is a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent states through those future rewards. Thus MDPs involve delayed reward and the need to trade-off immediate and delayed reward.

Figure 1 depicts MDP where an agent interacts with environment. The current state of the environment is known to the agent. With knowledge of state, the agent makes a decision as to which action to take. This action, in turn, will change the state of the environment and the agent will receive a reward at the same time. The agent remembers this reward in some indirect way and uses it for making future decisions.

1.1 Markov property

Suppose that the agent is in state S_t takes action A_t at time t. Then the agent receives reward R_{t+1} (from the environment) and the environment transitions to state S_{t+1}. MDP assumes that all these quantities are random variables.

Let $\mathcal{S}$ and $\mathcal{A}$ be the set of all the states and that of all the actions the agent can take respectively.

Now suppose that the environment is in state $S_0 \in \mathcal{S}$ the agent takes action $A_0 \in \mathcal{A}$ at $t = 0$. Then the state of the environment becomes $S_1 \in \mathcal{S}$ giving the agent $R_1 \in \mathbf{R}$ as reward. Suppose that the agent repeat taking actions.

Then we have a sequence of random variables

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, \ldots \tag{1}$$

We assume that these random variables satisfy the Markov Property (as assumed by the name) in the following sense.

$$S_{t+1}, R_{t+1}|S_t, A_t, R_t, S_{t-1}, A_{t-1}, R_{t-1}, \ldots = S_{t+1}, R_{t+1}|S_t, A_t \tag{2}$$

i.e., two random variables, S_{t+1} and R_{t+1}, conditioned on every state, action, and reward before $t+1$ are the same as those conditioned on S_t and R_t only.

This can be formally expressed using the probability density function (PDF) as follows.

$$p\left(S_{t+1}, R_{t+1} \middle| S_t, A_t, R_t, S_{t-1}, A_{t-1}, R_{t-1}, \ldots\right) = p\left(S_{t+1}, R_{t+1} \middle| S_t, A_t\right). \tag{3}$$

This is the reason that the process is called *Markov* decision process.

1.2 Policy

The *policy* is defined by the conditional probability of A_t given S_t, *i.e.*,

$$\pi(A|S) = p(A_t|S_t), \tag{4}$$

which implies the probability of taking certain action depends only on the current state, not the time. The policy decides which actions the agent takes in each state.

Let Π be the set of all the policies.

1.3 Return

The *return* at t is defined by

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \tag{5}$$

where $\gamma \in [0, 1]$ is called the *discount factor*. If $\gamma = 0$, the agent is myopic, *i.e.*, it only cares the immediate reward. If $\gamma = 1$, the agent is truly far-sighted, *i.e.*, it cares all the future rewards without discounting. If γ is somewhere between 0 and 1, it considers near-future rewards more importantly than those in far future.

1.4 State value function and action value function

The state value function (which is sometimes referred to as just value function) is defined by

$$v_\pi(s) = \mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s\right\} = \mathbf{E}_{\pi,p}\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k} \middle| S_t = s\right\}. \tag{6}$$

In other words, the state value function is a function of a state representing the expected return the agent will get from the state when following the policy π.

The action value function (which is sometimes referred to as just action function) is defined by

$$q_\pi(s, a) = \mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t = a\right\} = \mathbf{E}_{\pi,p}\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k} \middle| S_t = s, A_t = a\right\}. \tag{7}$$

In other words, the action value function is a function of a state and an action representing the expected return the agent will get from the state when the agent takes a certain action and follows the policy π.

As mentioned above, most reinforcement learning algorithms try to maximize either one of these functions, *i.e.*, not maximizing the immediate reward, but the long-term return.

2 Bellman equation

Richard E. Bellman, who introduced dynamic programming in 1953, proposed an equation as a necessary condition for optimality associated with dynamic programming, which is called Bellman equation. One of the properties that Markov property implies is that the value functions only depend on the current state (and the action taken) and that the function value is closely related to the function values of the next states. These facts are cleverly used to derive the Bellman equation. Here we introduce two Bellman equations; one for the state value function and the other for action value function.

2.1 Bellman equations

To derive Bellman equations, we use some basic statistics facts regarding conditional expectations. (Refer to §12.)

Since the definitions of state value function and action value function together with (44) imply

$$\begin{aligned} v_\pi(s) &= \mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s\right\} \\ &= \mathbf{E}_{A_t|S_t=s}\mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t\right\} \\ &= \sum_a p(A_t = a|S_t = s)\mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t = a\right\} \\ &= \sum_a \pi(a|s)\mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t = a\right\} \\ &= \sum_a \pi(a|s) q_\pi(s,a) \end{aligned}$$

and

$$\begin{aligned} q_\pi(s,a) &= \mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t = a\right\} \\ &= \mathbf{E}_{S_{t+1},R_{t+1}|S_t=s,A_t=a}\mathbf{E}_{\pi,p}\left\{G_t \middle| S_t = s, A_t = a, S_{t+1}, R_{t+1}\right\} \\ &= \mathbf{E}_{S_{t+1},R_{t+1}|S_t=s,A_t=a}\mathbf{E}_{\pi,p}\left\{\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \middle| S_t = s, A_t = a, S_{t+1}, R_{t+1}\right\} \\ &= \mathbf{E}_{S_{t+1},R_{t+1}|S_t=s,A_t=a}\mathbf{E}_{\pi,p}\left\{R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \middle| S_t = s, A_t = a, S_{t+1}, R_{t+1}\right\} \\ &= \sum_{s',r} p_{S_{t+1},R_{t+1}|S_t,A_t}(s',r|s,a)\mathbf{E}_{\pi,p}\left\{R_{t+1} + \gamma G_{t+1} \middle| S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r\right\} \\ &= \sum_{s',r} p_{S_{t+1},R_{t+1}|S_t,A_t}(s',r|s,a)\left(r + \gamma\mathbf{E}_{\pi,p}\left\{G_{t+1} \middle| S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r\right\}\right) \\ &= \sum_{s',r} p_{S_{t+1},R_{t+1}|S_t,A_t}(s',r|s,a)\left(r + \gamma\mathbf{E}_{\pi,p}\left\{G_{t+1} \middle| S_{t+1} = s'\right\}\right) \\ &= \sum_{s',r} p_{S_{t+1},R_{t+1}|S_t,A_t}(s',r|s,a)\left(r + \gamma v_\pi(s')\right), \end{aligned}$$

we have the following two equations relating state value function to action value function and vise versa.

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{8}$$

$$q_\pi(s,a) = \sum_{s',r} p_{S_{t+1},R_{t+1}|S_t,A_t}(s',r|s,a)\left(r + \gamma v_\pi(s')\right). \tag{9}$$

Now (8) and (9) imply that

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left(r + \gamma v_\pi(s')\right) \tag{10}$$

and

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left(r + \gamma v_\pi(s')\right) = \sum_{s',r} p(s',r|s,a)\left(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\right). \tag{11}$$

The equation (10) is called *Bellman equation for state value function* and the equation (11) is called *Bellman equation for action value function.*

2.2 Bellman optimality equations

Now suppose that the policy π_* is the optimal policy. Then we define the *optimal state-value function* as that of π_*, *i.e.*,

$$v_*(s) = v_{\pi_*}(s) = \max_{\pi \in \Pi} v_\pi(s). \tag{12}$$

Likewise, we define the *optimal action-value function* as that of π_*, *i.e.*,

$$q_*(s,a) = q_{\pi_*}(s,a) = \max_{\pi \in \Pi} q_\pi(s,a). \tag{13}$$

Then (8) and (9) imply that

$$v_*(s) = v_{\pi_*}(a) = \max_{a \in \mathcal{A}} q_{\pi_*}(s,a) = \max_{a \in \mathcal{A}} \sum_{s',r} p(s',r|s,a)\left(r + \gamma v_\pi(s')\right). \tag{14}$$

and

$$q_*(s,a) = q_{\pi_*}(s,a) = \sum_{s',r} p(s',r|s,a)\left(r + \gamma v_{\pi_*}(s')\right). = \sum_{s',r} p(s',r|s,a)\left(r + \gamma \max_{a' \in \mathcal{A}} q_{\pi_*}(s',a')\right). \tag{15}$$

The equation (14) is called *Bellman optimality equation for state value function* and the equation (15) is called *Bellman optimality equation for action value function.*

3 Dynamic programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). DP provides an essential foundation for the understanding of the methods presented in the rest of this chapter. All of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies.

3.1 Policy evaluation (prediction)

We consider how to compute the state-value function v_π for an arbitrary policy π. This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem.* The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

Inputs: π, MDP
Algorithm parameters: $\theta > 0$ (small threshold determining accuracy of estimation)

Initialize $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$ except that $V(\text{terminal}) = 0$

Loop:
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma V(s')\right)$
        $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
until $\Delta < \theta$

Table 1: Iterative Policy Evaluation for estimating $V \sim v_\pi$

The policy evaluation algorithm uses the fact that all the state value functions satisfy the Bellman equation for state value function. We use this equation, but in an iterative manner.

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_k(s')\right). \tag{16}$$

This equation resembles (10), but different because now we put subscript k in place of the policy π. Indeed, the sequence v_k can be shown in general to converge to v_π as k goes to ∞ the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*.

We can consider *in-place* version of this algorithm, *i.e.*, we replace the values for v_k for each state without waiting until we sweep all the states for one iteration. This in-place algorithm also converges to v_π. In fact, it usually converges faster. This in-place algorithm is described in Table 1.

3.2 Policy iteration

The policy iteration is the iterative process of improving policy as to maximize the value functions. The algorithm is described in Table 2.

3.3 Value iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation. In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps.

$$v_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_k(s')\right). \tag{17}$$

Note that value iteration is obtained simply by turning the Bellman optimality equation for state value function (14) into an update rule.

The in-place version of value iteration algorithm is described in Table 3.

Inputs: MDP
Algorithm parameters: $\theta > 0$ (small threshold determining accuracy of estimation)

1. Initialization
    $V(s) \in \mathbf{R}$ and $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$

2. Policy Evaluation
    Loop:
        $\Delta \leftarrow 0$
        For each $s \in \mathcal{S}$:
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) (r + \gamma V(s'))$
            $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
    until $\Delta < \theta$

3. Policy Improvement
$u \leftarrow$ `true`
    For each $s \in \mathcal{S}$
        $b \leftarrow \pi(s)$
        $\pi(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) (r + \gamma v_\pi(s'))$
        If $b \neq \pi(s)$, then $t \leftarrow$ `false`
    If u, then stop and return $V \sim v_*$ and $\pi \sim \pi_*$; else go to 2

Table 2: Policy Iteration (using iterative policy evaluation) for estimating $\pi \sim \pi_*$.

Inputs: MDP
Algorithm parameters: $\theta > 0$ (small threshold determining accuracy of estimation)

Initialize $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$ except that $V(\text{terminal}) = 0$

Loop:
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) (r + \gamma V(s'))$
        $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
until $\Delta < \theta$

Output: deterministic policy π such that
    $\pi(s) = \text{argmax}_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) (r + \gamma V(s'))$

Table 3: Value Iteration for estimating $\pi \sim \pi_*$

4 Monte Carlo methods

Here we consider learning methods for estimating value functions and discovering optimal policies. Unlike the previous methods, we do not assume complete knowledge of the environment. Monte Carlo (MC) methods require only experience sample sequences of states, actions, and rewards from *actual or simulated interaction with an environment*. Learning from actual experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior. *Learning from simulated experience is also powerful.* Although a model is required, the model need only generate sample transitions, *not the complete probability distributions of all possible transitions* that is required for dynamic programming (DP).

MC methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks.

Monte Carlo methods sample and average returns for each state–action pair much like the bandit methods where we sample and average rewards for each action. The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode. Because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state.

To handle the nonstationarity, we adapt the idea of general policy iteration (GPI) developed for DP. Whereas there we computed value functions from knowledge of the MDP, here we learn value functions from sample returns with the MDP. The value functions and corresponding policies still interact to attain optimality in essentially the same way (GPI). As in DP, first we consider the prediction problem, then policy improvement, and, finally, the control problem and its solution by GPI. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

4.1 Monte Carlo prediction

An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. There are two Monte Carlo (MC) prediction methods; *first-visit MC method* and *every-visit MC method*. These two MC methods are very similar but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s. Every-visit MC extends more naturally to function approximation and eligibility traces.

Table 4 describes the first-visit MC prediction algorithm.

4.2 Monte Carlo control

The overall idea is to proceed according to the same pattern as in the dynamic programming, *i.e.*, according to the idea of generalized policy iteration (GPI). In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.

For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. A complete simple algorithm along these lines, which is called Monte Carlo ES for Monte Carlo with Exploring Starts, is described in Table 5.

4.3 Monte Carlo control without exploring starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call on-policy methods and off-policy methods. On-policy methods

Inputs: π

Initialize:
    $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$
    $R(s) \leftarrow \texttt{list}()$ for all $s \in \mathcal{S}$

Loop:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t + T - 1, T - 2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        If $S_t \notin \{S_0, S_1, \ldots, S_{t-1}\}$:
            $R(S_t).\texttt{append}(G)$
            $V(S_t) \leftarrow R(S_t).\texttt{average}()$
Until a certain criterion is satisfied

Table 4: First-visit MC prediction for estimating $V \sim v_\pi$

Initialize:
    $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$
    $Q(s, a) \in \mathbf{R}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
    $R(s, a) \leftarrow \texttt{list}()$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop:
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
    Generate an episode from S_0, A_0 following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t + T - 1, T - 2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        If $S_t \notin \{S_0, S_1, \ldots, S_{t-1}\}$:
            $R(S_t, A_t).\texttt{append}(G)$
            $Q(S_t, A_t) \leftarrow R(S_t, A_t).\texttt{average}()$
            $\pi(S_t) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}(S_t)} Q(S_t, a)$
Until a certain criterion is satisfied

Table 5: MC ES (exploring starts) for estimating $\pi \sim \pi_*$

Algorithm parameters: small $\epsilon > 0$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$
$Q(s, a) \in \mathbf{R}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
$R(s, a) \leftarrow \texttt{list}()$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop:
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0 following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t + T - 1, T - 2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
If $S_t \notin \{S_0, S_1, \ldots, S_{t-1}\}$:
$R(S_t, A_t).\texttt{append}(G)$
$Q(S_t, A_t) \leftarrow R(S_t, A_t).\texttt{average}()$
$A^* \leftarrow \text{argmax}_{a \in \mathcal{A}(S_t)}$
For all $a \in \mathcal{A}(S_t)$

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Until a certain criterion is satisfied

Table 6: On-policy first-visit MC control (for ϵ-soft policies) for estimating $\pi \sim \pi_*$

attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. The Monte Carlo ES method developed above is an example of an on-policy method. In this section we show how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts.

The on-policy first-visit MC control using ϵ-greedy is described in Table 6.

4.4 Off-policy prediction via important sampling

XXX

4.5 Off-policy Monte Carlo control

XXX

5 Temporal-difference learning

Temporal-difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning.

We start by focusing on the policy evaluation or prediction problem, the problem of estimating the value function v_π for a given policy π. For the control problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI).

Inputs: the policy π to be evaluated
Algorithm parameters: step size $\alpha \in (0, 1]$

Initialize:
    $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$ except that $V(\text{terminal}) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow (1 - \alpha)V(S) + \alpha(R + \gamma V(S'))$
        $S \leftarrow S'$
    until S is terminal
Until a certain criterion is satisfied

Table 7: TD(0) for estimating v_π

5.1 TD prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. A simple every-visit MC method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) = (1 - \alpha)V(S_t) + \alpha G_t. \tag{18}$$

TD methods need to wait only until the next time step. At time $t + 1$, they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) = (1 - \alpha)V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1})). \tag{19}$$

This TD method is called TD(0), or one-step TD, because it is a special case of the TD(λ) and n-step TD methods. Table 7 specifies TD(0) completely in procedural form.

The quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity, called the TD error, arises in various forms throughout reinforcement learning. It can be formally defined as follows.

$$\delta_t \coloneqq R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{20}$$

It is interesting to observe that we can express Monte Carlo error in terms of modified TD errors if we defined the modified TD error as follows.

$$\delta'_t \coloneqq R_{t+1} + \gamma V_{t+1}(S_{t+1}) - V_t(S_t) \tag{21}$$

Then the Monte Carlo error is defined by $G_t - V_t(S_t)$ is

$$\begin{aligned}
G_t - V_t(S_t) &= R_{t+1} + \gamma G_{t+1} - V_t(S_t) \\
&= R_{t+1} + \gamma \left(G_{t+1} - V_{t+1}(S_{t+1}) + V_{t+1}(S_{t+1})\right) - V_t(S_t) \\
&= R_{t+1} + \gamma V_{t+1}(S_{t+1}) - V_t(S_t) + \gamma \left(G_{t+1} - V_{t+1}(S_{t+1})\right) \\
&= \delta'_t + \gamma \left(G_{t+1} - V_{t+1}(S_{t+1})\right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \left(G_{t+2} - V_{t+2}(S_{t+2})\right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{T-t-2} \delta'_{T-2} + \gamma^{T-t-1} \left(G_{T-1} - V_{T-1}(S_{T-1})\right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{T-t-2} \delta'_{T-2} + \gamma^{T-t-1} \left(R_T + \gamma V_T(S_T) - V_{T-1}(S_{T-1})\right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{T-t-2} \delta'_{T-2} + \gamma^{T-t-1} \delta'_{T-1} \\
&= \sum_{k=t}^{T-1} \gamma^{k-t} \delta'_k = \sum_{k=0}^{T-t-1} \gamma^k \delta'_{k+t} \qquad (22)
\end{aligned}$$

where the fact that the state-value function for a terminal state, $V_{T-1}(S_T)$, is 0 is used.

This means the Monte Carlo error, *i.e.*, the difference between the return along the path from t to a terminal state of the episode and the state-value function of S_t can be expressed as sum of discounted (modified) one-step TD errors. If we assume that every V_t does not change during the episode, δ_t coincides with δ'_t. Hence (22) becomes

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k = \sum_{k=0}^{T-t-1} \gamma^k \delta_{k+t}. \qquad (23)$$

5.2 Sarsa: on-policy TD Control

As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ϵ greedy or ϵ-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ϵ-greedy policies by setting $\epsilon = 1/t$).

This algorithm is described in Table 8.

5.3 Q-learning: off-policy TD control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

$$\begin{aligned}
Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right) \\
&= (1 - \alpha) Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a)\right). \qquad (24)
\end{aligned}$$

The learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*. The Q-learning algorithm is described in Table 9.

Algorithm parameters: step size $\alpha \in (0, 1]$ and small $\epsilon > 0$

Initialize:
    $Q(s, a) \in \mathbf{R}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ except $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (*e.g.*, ϵ-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (*e.g.*, ϵ-greedy)
        $Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma Q(S', A'))$
        $S \leftarrow S'$, $A \leftarrow A'$,
    until S is terminal
Until a certain criterion is satisfied

Table 8: Sarsa (on-policy TD control) for estimating $Q \sim q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$ and small $\epsilon > 0$

Initialize:
    $Q(s, a) \in \mathbf{R}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ except $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (*e.g.*, ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma \max_{a \in \mathcal{A}(S')} Q(S', a))$
        $S \leftarrow S'$
    until S is terminal
Until a certain criterion is satisfied

Table 9: Q-learning (off-policy TD control) for estimating $\pi \sim \pi_*$

5.4 Maximization bias and double learning

XXX

6 n-step bootstrapping

There exists another method which unifies the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods. Neither MC methods nor one-step TD methods are always the best. Here we present n-step TD methods that generalize both methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task. n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

Another way of looking at the benefits of n-step methods is that they free one from the tyranny of the time step. With one-step TD methods the same time step determines how often the action can be changed and the time interval over which bootstrapping is done. In many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made. n-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

The idea of n-step methods is usually used as an introduction to the algorithmic idea of eligibility traces.

6.1 n-step TD prediction

n-step TD prediction is a method lying between Monte Carlo and (one-step) TD method, *i.e.*, TD(0). Consider estimating v_π from sample episodes generated using π. Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards.

One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination.

The methods that use n-step updates are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called n-step TD methods.

Suppose that the process is episodic, *i.e.*, ever episode ends or enters a terminal state within finite number of steps. Then the *target* of Monte Carlo update is the return at time step t, *i.e.*,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \tag{25}$$

where T is the last time step of the episode.

The target of the one-step TD method is the first reward plus the discounted estimated value of the next state, *i.e.*,

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \tag{26}$$

where $V_t : \mathcal{S} \to \mathbf{R}$ is the estimate of $v_\pi(S_{t+1})$ at time t. Note that the second term $\gamma V_t(S_{t+1})$ is the estimate for $\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$, but using the state-value function estimate of the next state, instead of using future discounted returns. Thus, this is a bootstrapping. Likewise, we can define two-step return as a target for the two-step update.

$$G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}) \tag{27}$$

where $V_{t+1} : \mathcal{S} \to \mathbf{R}$ is the estimate of $v_\pi(S_{t+2})$ at time $t+1$. Again here the third term $\gamma^2 V_{t+1}(S_{t+2})$ is the estimate for $\gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$ using bootstrapping. In general, we can define the n-step return as

Inputs: policy π to be evaluated

Algorithm parameters: step size $\alpha_t \in (0, 1]$ and $n \in \mathbf{N}$

Initialize:
    $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$ except that $V(\text{terminal}) = 0$

Loop for each episode:
    Initialize and store S_0
    $T \leftarrow \infty$
    Loop for each $t = 0, 1, 2, \ldots$:
        If $t < T$:
            $A_t \leftarrow$ action given by $\pi(\cdot|S_t)$
            Take action A_t, observe R_{t+1}, S_{t+1}
            If S_{t+1} is terminal:
                $T \leftarrow t + 1$
        $\tau \leftarrow t - n + 1$
        If $\tau \geq 0$:
            $G \leftarrow \sum_{i=\tau+1}^{\min\{\tau+n,T\}} \gamma^{i-\tau-1} R_i$
            If $\tau + n < T$:
                $G \leftarrow G + \gamma^n V(S_{\tau+n})$
            $V(S_\tau) \leftarrow (1 - \alpha_t)V(S_\tau) + \alpha_t G$
    while $\tau < T - 1$
Until a certain criterion is satisfied

Table 10: n-step TD for estimating $V \sim v_\pi$

the target for the n-step update.

$$G_{t:t+n} = \begin{cases} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) & \text{if } t + n < T \\ G_t & \text{if } t + n \geq T \end{cases} \tag{28}$$

for $t \geq 0$. An n-step return can be considered as an approximation to the full return G_t.

We can consider algorithm with the n-step update using this n-step return, *i.e.*,

$$V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha_{t+n-1}(G_{t:t+n} - V_{t+n-1}(S_t)) = (1 - \alpha_{t+n-1})V_{t+n-1}(S_t) + \alpha_{t+n-1}G_{t:t+n} \tag{29}$$

Note that this update cannot be performed before time step $t + n$ because only by then, all the rewards necessary to evaluate (28) become available. Also note that (29) is reduced to (19) when $n = 1$. Therefore one-step TD method is a special case of n-step TD method. The n-step TD prediction is described in Table 10, a Pythonian version of which is described in Table 11.

As the Monte Carlo error can be express as the sum of discounted TD errors, the n-step TD error can

Inputs: policy π to be evaluated

Algorithm parameters: step size $\alpha_t \in (0, 1]$ and $n \in \mathbf{N}$

Initialize:
    $V(s) \in \mathbf{R}$ for all $s \in \mathcal{S}$ except that $V(\text{terminal}) = 0$
    `w_list` $= [1, \gamma, \ldots, \gamma^n]$

Loop for each episode:
    `S_list` $\leftarrow$ `list()`, `R_list` $\leftarrow$ `list()`
    Initialize S
    `S_list.append`(S), `not_terminated` $\leftarrow True$
    Loop for each $t = 0, 1, 2, \ldots$:
        If `not_terminated`:
            $A \leftarrow$ action given by $\pi(\cdot|S)$
            Take action A, observe R, S'
            `R_list.append`(R)
            If S is terminal:
                `not_terminated` $\leftarrow False$
            else:
                `S_list.append`(S')
                $S \leftarrow S'$
        If $t \geq n - 1$:
            `n_R_list` $\leftarrow$ `R_list`$[t - n + 1 : t + 1]$
            $G \leftarrow$ (`n_R_list` $*$ `w_list`[: `len`(`n_R_list`)]).`sum`()
            If $t + 1 <$ `len`(`S_list`):
                $G \leftarrow G +$ `w_list`$[-1] \times V($`S_list`$[t + 1])$
            $V($`S_list`$[t - n + 1]) \leftarrow (1 - \alpha_t)V($`S_list`$[t - n + 1]) + \alpha_t G$
    while $t - n + 1 <$ `len`(`R_list`) $- 1$
Until a certain criterion is satisfied

Table 11: n-step TD for estimating $V \sim v_\pi$ (Pythonic style)

be expressed as the sum of the discounted one-step TD errors. Note that $G_{t:t+n} = R_{t+1} + \gamma G_{t+1:t+n}$. Thus,

$$\begin{aligned}
G_{t:t+n} - V_t(S_t) &= R_{t+1} + \gamma G_{t+1:t+n} - V_t(S_t) \\
&= R_{t+1} + \gamma \left(G_{t+1:t+n} + V_{t+1}(S_{t+1}) - V_{t+1}(S_{t+1}) \right) - V_t(S_t) \\
&= R_{t+1} + \gamma V_{t+1}(S_{t+1}) - V_t(S_t) + \gamma \left(G_{t+1:t+n} - V_{t+1}(S_{t+1}) \right) \\
&= \delta'_t + \gamma \left(G_{t+1:t+n} - V_{t+1}(S_{t+1}) \right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \left(G_{t+2:t+n} - V_{t+2}(S_{t+2}) \right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{n-1} \left(G_{t+n-1:t+n} - V_{t+n-1}(S_{t+n-1}) \right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{n-1} \left(R_{t+n} + \gamma V_{t+n-1}(S_{t+n}) - V_{t+n-1}(S_{t+n-1}) \right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{n-1} \left(R_{t+n} + \gamma V_{t+n}(S_{t+n}) - V_{t+n-1}(S_{t+n-1}) \right) \\
&\quad + \gamma^n \left(V_{t+n-1}(S_{t+n}) - V_{t+n}(S_{t+n}) \right) \\
&= \delta'_t + \gamma \delta'_{t+1} + \gamma^2 \delta'_{t+2} + \cdots + \gamma^{n-1} \delta'_{t+n-1} + \gamma^n \left(V_{t+n-1}(S_{t+n}) - V_{t+n}(S_{t+n}) \right) \\
&= \sum_{k=t}^{t+n-1} \gamma^{k-t} \delta'_k + \gamma^n \left(V_{t+n-1}(S_{t+n}) - V_{t+n}(S_{t+n}) \right) \\
&= \sum_{k=0}^{n-1} \gamma^k \delta'_{k+t} + \gamma^n \left(V_{t+n-1}(S_{t+n}) - V_{t+n}(S_{t+n}) \right)
\end{aligned}$$

where δ'_t is defined in (21). Thus, the n-step TD error is

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=0}^{n-1} \gamma^k \delta'_{k+t} + \gamma^n \left(V_{t+n-1}(S_{t+n}) - V_{t+n}(S_{t+n}) \right) + \left(V_t(S_t) - V_{t+n-1}(S_t) \right). \tag{30}$$

If V_t does not change during the episode, $\delta_t = \delta'_t$ and $V_{t+n-1}(S_{t+n}) = V_{t+n}(S_{t+n})$, hence

$$G_{t:t+n} - V(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k = \sum_{k=0}^{n-1} \gamma^k \delta_{t+k} \tag{31}$$

where δ_t is defined in (20).

The n-step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. The worst error of the expected n-step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}.

$$\max_{s \in \mathcal{S}} \left| \mathbf{E}_\pi \left\{ G_{t:t+n} \middle| S_t = s \right\} - v_\pi(s) \right| \leq \gamma^n \max_s \left| V_{t+n-1}(s) - v_\pi(s) \right| \tag{32}$$

for all $n \geq 1$.

This is called the error reduction property of n-step returns. Because of the error reduction property, one can show formally that all n-step TD methods converge to the correct predictions under appropriate technical conditions. The n-step TD methods thus form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

6.2 n-step Sarsa

The n-step Sarsa uses the previous n-step temporal difference idea for control. Like one-step Sarsa, we use a behavior policy based on Q functions that the model learns, but update Q functions using the n-step return based on Q function.

$$G^Q_{t:t+n} = \begin{cases} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) & \text{if } t+n < T \\ G_t & \text{if } t+n \geq T \end{cases} \tag{33}$$

Algorithm parameters: step size $\alpha_t \in (0, 1]$, small $\epsilon > 0$, and $n \in \mathbf{N}$

Initialize:
    $Q(s, a) \in \mathbf{R}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize and store S_0
    Select an action A_0 using policy derived from Q (*e.g.*, ϵ-greedy)
    $T \leftarrow \infty$
    Loop for each $t = 0, 1, 2, \ldots$:
        If $t < T$:
            Take action A_t, observe R_{t+1}, S_{t+1}
            If S_{t+1} is terminal:
                $T \leftarrow t + 1$
            else:
                Select an action A_{t+1} using policy derived from Q (*e.g.*, ϵ-greedy)
        $\tau \leftarrow t - n + 1$
        If $\tau \geq 0$:
            $G \leftarrow \sum_{i=\tau+1}^{\min\{\tau+n, T\}} \gamma^{i-\tau-1} R_i$
            If $\tau + n < T$:
                $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
            $Q(S_\tau, A_\tau) \leftarrow (1 - \alpha_t) Q(S_\tau, A_\tau) + \alpha_t G$
    while $\tau < T - 1$
Until a certain criterion is satisfied

Table 12: n-step Sarsa for estimating $Q \sim q_*$ or q_π

where $t \geq 0$.

Then, the n-step update using this n-step return is

$$\begin{aligned} Q_{t+n}(S_t, A_t) &= Q_{t+n-1}(S_t, A_t) + \alpha_{t+n-1} \left(G^Q_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right) \\ &= (1 - \alpha_{t+n-1}) Q_{t+n-1}(S_t, A_t) + \alpha_{t+n-1} G^Q_{t:t+n} \end{aligned}$$

The algorithm using this update rule is called *n-step Sarsa*. Table 12 describes this algorithm.

As before, the n-step return for Sarsa can be expressed as the sum of TD errors in terms of Q function.

Suppose that $t+n<T$. Then

$$\begin{aligned}
G^Q_{t:t+n} &= R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\cdots+\gamma^{n-1}R_{t+n}+\gamma^n Q_{t+n-1}(S_{t+n},A_{t+n}) \\
&= \sum_{k=t}^{t+n-1}\gamma^{k-t}R_{k+1}+\gamma^n Q_{t+n-1}(S_{t+n},A_{t+n}) \\
&= \sum_{k=t}^{t+n-1}\gamma^{k-t}R_{k+1}+\sum_{k=t}^{t+n-1}\gamma^{k-t}\left(Q_k(S_k,A_k)-Q_k(S_k,A_k)\right)+\gamma^n Q_{t+n-1}(S_{t+n},A_{t+n}) \\
&= \sum_{k=t}^{t+n-1}\gamma^{k-t}R_{k+1}+\sum_{k=t-1}^{t+n-2}\gamma^{k-t+1}Q_{k+1}(S_{k+1},A_{k+1})-\sum_{k=t}^{t+n-1}\gamma^{k-t}Q_k(S_k,A_k) \\
&\quad +\gamma^n Q_{t+n-1}(S_{t+n},A_{t+n}) \\
&= \sum_{k=t}^{t+n-1}\left(\gamma^{k-t}R_{k+1}+\gamma^{k-t+1}Q_{k+1}(S_{k+1},A_{k+1})-\gamma^{k-t}Q_k(S_k,A_k)\right) \\
&\quad +Q_t(S_t,A_t)-\gamma^n Q_{t+n}(S_{t+n},A_{t+n})+\gamma^n Q_{t+n-1}(S_{t+n},A_{t+n}) \\
&= \sum_{k=t}^{t+n-1}\gamma^{k-t}\left(R_{k+1}+\gamma Q_{k+1}(S_{k+1},A_{k+1})-Q_k(S_k,A_k)\right) \\
&\quad +Q_t(S_t,A_t)+\gamma^n\left(Q_{t+n-1}(S_{t+n},A_{t+n})-Q_{t+n}(S_{t+n},A_{t+n})\right)
\end{aligned}$$

Thus the n-step error for Sarsa becomes

$$\begin{aligned}
G^Q_{t:t+n}-Q_{t+n-1}(S_{t+n},A_{t+n}) &= \sum_{k=t}^{t+n-1}\gamma^{k-t}\left(R_{k+1}+\gamma Q_{k+1}(S_{k+1},A_{k+1})-Q_k(S_k,A_k)\right) \\
&\quad +\left(Q_t(S_t,A_t)-Q_{t+n-1}(S_{t+n},A_{t+n})\right)+\gamma^n\left(Q_{t+n-1}(S_{t+n},A_{t+n})-Q_{t+n}(S_{t+n},A_{t+n})\right). \qquad (34)
\end{aligned}$$

Again, if Q_t does not change during the episode, (34) becomes

$$G^Q_{t:t+n}-Q(S_{t+n},A_{t+n})=\sum_{k=t}^{t+n-1}\gamma^{k-t}\left(R_{k+1}+\gamma Q(S_{k+1},A_{k+1})-Q(S_k,A_k)\right) \qquad (35)$$

6.3 n-step off-policy learning

XXX

7 Planning and learning with tabular methods

Table 13 describes the random sample one-step tabular Q-learning.

7.1 Dyna: integrated planning, acting, and learning

The general Dyna architecture is depicted in Figure 2. The tabular Dyna-Q algorithm is described in Table 14.

8 On-policy Prediction with Approximation

XXX

Loop:
Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
Send S, A to a sample model, and obtain a sample next reward, R, and a sample next state, S'
Apply one-step tabular Q-learning to S, A, R, S':
$Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha\left(R + \gamma \max_{a \in \mathcal{A}(S)} Q(S',a)\right)$
Until a certain criterion is satisfied

Table 13: Random sample one-step tabular Q-learning

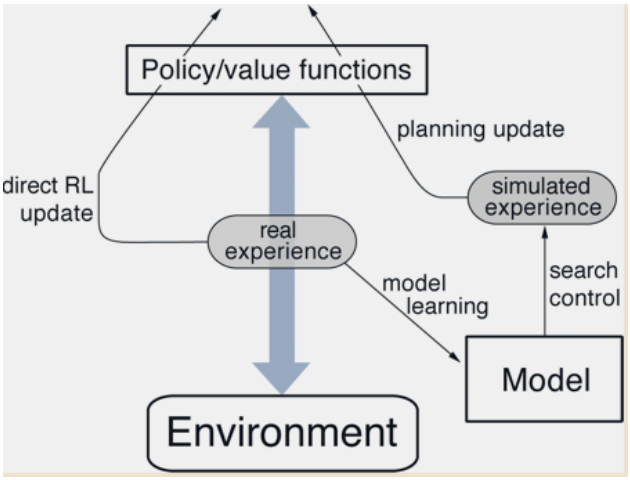

Figure 2: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(S)$

Loop:
$S \leftarrow$ current (nonterminal) state
$A \leftarrow \epsilon$-greedy(S,Q)
Take action A; observe reward R and next state S'
Update Q-function: $Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha\left(R + \gamma \max_{a \in \mathcal{A}(S)} Q(S',a)\right)$
$Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
Loop repeat
Apply one-step tabular Q-learning to S, A, R, S':
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
Update Q-function: $Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha\left(R + \gamma \max_{a \in \mathcal{A}(S)} Q(S',a)\right)$
Until a certain criterion is satisfied

Table 14: Tabular Dyna-Q

9 On-policy Control with Approximation

XXX

10 Off-policy Methods with Approximation

XXX

11 Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

We have already seen one way of unifying TD and Monte Carlo methods: the n-step TD methods §6. What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages.

The mechanism is a short-term memory vector, the *eligibility trace* $z_t \in \mathbf{R}^d$ that parallels the long-term weight vector $w_t \in \mathbf{R}^d$. The rough idea is that when a component of w_t participates in producing an estimated value, then the corresponding component of z_t is bumped up and then begins to fade away. Learning will then occur in that component of w_t if a nonzero TD error occurs before the trace falls back to zero. The *trace-decay parameter* $\lambda \in [0, 1]$ determines the rate at which the trace falls.

The primary computational advantage of eligibility traces over n-step methods is that only a single trace vector is required rather than a store of the last n feature vectors. Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode. In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

11.1 The λ-return

In §6 we defined an n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted. The general form of that equation, for any parameterized function approximator, is

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, w_{t+n-1}) \tag{36}$$

Now we note that a valid update can be done not just toward any n-step return, but toward any average of n-step returns for different ns. An example of such an average can be the average of 3-step return, 5-step return, and 7-step return, *i.e.*,

$$\frac{1}{3}(G_{t:t+3} + G_{t:t+5} + G_{t:t+7}) \tag{37}$$

This is one example, but averaging produces a substantial new range of algorithms. For example, we can average across many n-step returns from 1-step return to ∞-step return to obtain another way of interrelating TD and Monte Carlo methods. Nn principle, one could even average experience-based updates with DP updates to get a simple combination of experience-based and model-based methods.

An update that averages simpler component updates is called a compound update. The TD(λ) algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^n (where $\lambda \in [0, 1]$), and is normalized (to ensure that the weights sum to 1). The resulting update is toward a return, called the λ-return , defined in its state-based form by

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{38}$$

where $(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}=1$. If the episode termintes at $t=T$, then $G_{t:t+n}=G_{t:T}=G_t$ for all $n\geq T-t$, thus,

$$G_t^\lambda=(1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n}+(1-\lambda)\sum_{n=T-t}^{\infty}\lambda^{n-1}G_{t:t+n}=(1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n}+\lambda^{T-t-1}G_t. \quad (39)$$

Note that if $\lambda=1$, G_t^λ becomes the original return G_t. Thus, updating value functions according to λ-return is equivalent to Monte Carlo algorithm. On the other hand, if $\lambda=0$, G_t^λ becomes $G_{t:t+1}$, hence updating value functions according to λ-return is equivalent to the one-step temporal difference method, *i.e.*, TD(0) method.

Now we define our first learning algorithm based on the λ-return: the *off-line λ-return algorithm*. As an off-line algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of off-line updates are made according to the semi-gradient method using the λ-return as the target. The update rule of this method is

$$w_{t+1}=w_t+\alpha_t\left(G_t^\lambda-\hat{v}(S_t,w_t)\right)\nabla_w\hat{v}(S_t,w_t) \quad (40)$$

for $t=0,\ldots,T-1$. The λ-return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n-step bootstrapping.

The above approach is what can be called the *theoretical or forward* view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them.

11.2 TD(λ)

TD(λ) is one of the oldest and most widely used algorithms in reinforcement learning. It was the first algorithm for which a formal relationship was shown between a more theoretical forward view and a more computationally congenial backward view using eligibility traces. Here we will show empirically that it approximates the off-line λ-return algorithm presented in the previous section.

TD(λ) improves over the off-line λ-return algorithm in three ways.

- It updates the weight vector on every step of an episode rather than only at the end, thus its estimates is updated sooner (and may be better sooner).
- Its computations are equally distributed in time (rather than all at the end of the episode).
- It can be applied to continuing problems rather than just to episodic problems.

The semi-gradient version of TD(λ) with function approximation will be shown below.

The *eligibility trace* is a vector $z_t\in\mathbf{R}^d$ where the number of components of z_t is the same as that of w_t. Whereas the *weight vector*, w_t, is a long-term memory accumulating over the lifetime of the system, the eligibility trace is a short-term memory, which typically lasts less time than the length of an episode. Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

12 Appendix: conditional probability and expected value

Suppose that we have a sequence of random variables, X, Y, and Z with supports $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$.

Note that the definition of the conditional probability implies that for all $x \in \mathcal{X}$, $y \in \mathcal{Y}$, $z \in \mathcal{Z}$ such that $p_Y(y) \neq 0$ and $p_Z(z) \neq 0$,

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)} \Leftrightarrow p_{X,Y}(x,y) = p_{X|Y}(x|y)p_Y(y). \tag{41}$$

$$p(x,y|z) = \frac{p(x,y,z)}{p(z)} = \frac{p(x,y,z)}{p(y,z)}\frac{p(y,z)}{p(z)} = p(x|y,z)p(y|z). \tag{42}$$

Then (41) implies

$$\begin{aligned}
\mathbf{E}(X) &= \int_{\mathcal{X}} x p_X(x)dx \\
&= \int_{\mathcal{X}} x \left(\int_{\mathcal{Y}} p_{X,Y}(x,y)dy \right) dx = \int_{\mathcal{X}} x \left(\int_{\mathcal{Y}} p_{X|Y}(x|y)p_Y(y)dy \right) dx \\
&= \int_{\mathcal{Y}} \int_{\mathcal{X}} x p_{X|Y}(x|y)p_Y(y)dxdy = \int_{\mathcal{Y}} \left(\int_{\mathcal{X}} x p_{X|Y}(x|y)dx \right) p_Y(y)dy \\
&= \int_{\mathcal{Y}} \mathbf{E}(X|Y=y)p_Y(y)dy = \mathbf{E}_Y \mathbf{E}_{X|Y}(X|Y),
\end{aligned}$$

$$\mathbf{E}(X) = \mathbf{E}_{X|Y}(X|Y), \tag{43}$$

and (42) implies

$$\begin{aligned}
\mathbf{E}(X|Z=z) &= \int_{\mathcal{X}} x p_{X|Z}(x|z)dx \\
&= \int_{\mathcal{X}} x \left(\int_{\mathcal{Y}} p_{X,Y|Z}(x,y|z)dy \right) dx = \int_{\mathcal{X}} x \left(\int_{\mathcal{Y}} p_{X|Y,Z}(x|y,z)p_{Y|Z}(y|z)dy \right) dx \\
&= \int_{\mathcal{Y}} \int_{\mathcal{X}} x p_{X|Y,Z}(x|y,z)p_{Y|Z}(y|z)dxdy = \int_{\mathcal{Y}} \left(\int_{\mathcal{X}} x p_{X|Y,Z}(x|y,z)dx \right) p_{Y|Z}(y|z)dy \\
&= \int_{\mathcal{Y}} \mathbf{E}(X|Y=y,Z=z)p_{Y|Z}(y|z)dy = \mathbf{E}_{Y|Z=z}\mathbf{E}(X|Y,Z=z),
\end{aligned}$$

i.e.,

$$\mathbf{E}(X|Z=z) = \mathbf{E}_{Y|Z=z}\mathbf{E}(X|Y,Z=z). \tag{44}$$