Summary of "Reinforcement Learning" by Sutton & Barto

Second edition, summarized by Pierre Enel

This document contains a summary of the book *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto.

The original material was modified in the following way: The content is shortened, and summarized, either by merely copying portions of the original text or by paraphrasing the content into a summarized paragraph. All formulas and algorithms are identical to the originals found in the book. This is still work in progress, see table of contents.

Contents

1 Introduction

Introduction chapter not included. It's probably better to read it if you're not familiar with reinforcement learning.

Tabular Solution Methods (discrete environment)

2 Multi-armed Bandits

2.1 A k-armed Bandit Problem

A k-armed bandit is a problem in which there are k different actions that lead to a reward drawn from a stationary probability distribution conditional on the chosen action. The goal is to maximize total reward over time.

We denote the action selected on time step t as A_t , and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a] \tag{1}$$

We assume that you do not know the action values with certainty, although you may have estimates. We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q_*(a)$

A **greedy** action is the action with the highest estimated value with current knowledge. The policy that always chooses **greedy** actions is the **greedy policy**. greedy actions correspond to a **exploiting** the current knowledge of the values of actions while non greedy actions correspond to an **exploring** behavior. With limited knowledge of the environment, reward can be lower in the short term with exploration but will be higher in the long run. It is not possible to explore and exploit simultaneously, so there is a conflict or trade-off between the two.

2.2 Action value Methods

The **sample-average** method estimates the Q values from the history of actions and rewards:

$$Q_t(a) \doteq \frac{\text{sum of rewards when a taken prior to t}}{\text{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \tag{2}$$

where $1_{predicate}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not. $Q_t(a)$ converges to $q_*(a)$.

Greedy actions as defined in the previous section are selected as follow, breaking ties randomly:

$$A_t \doteq \underset{a}{\operatorname{argmax}}\, Q_t(a) \tag{3}$$

where $argmax_a$ denotes the action a for which the expression that follows is maximized. The greedy policy is not exploring, to remediate this, we can use the **ϵ-greedy** policy where we explore by randomly selecting an action with a small probability ϵ.

2.3 The 10-armed Testbed

This section is about an example of ϵ-greedy policies with varying ϵ values.

2.4 Incremental Implementation

q value can be estimated with:

$$Q_n \doteq \frac{R_1 + R_2 + ... + R_{n-1}}{n-1} \tag{4}$$

However an incremental approach is more practical:

$$Q_{n+1} = Q_n + \frac{1}{n}[r_n - q_n] \tag{5}$$

Algorithm 1: A simple bandit algorithm

Initialize, for $a = 1$ to k: $Q(a) \leftarrow 0$ $N(a) \leftarrow 0$

while *True* **do**

$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\epsilon \text{ (breaking ties randomly)} \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$

end

2.5 Tracking a non-stationary problem

When we face a non-stationary problem (the environment and thus rewards can change), it's better to weigh more recent rewards:

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n] \tag{6}$$

where the step-size parameter $\alpha \in (0,\, 1]$ is constant. This results in Q_{n+1} being a weighted average of past rewards and the initial Q_1. This is sometimes called the **exponential recency-weighted average**.

2.6 Optimistic Initial Values

All the methods discussed so far are dependent on the initial action-value estimates, $Q_1(a)$, which makes them **biased**. In practice this is generally not a problem, but the initial values become another set of parameters that must be picked by the user, but can be viewed as some prior knowledge.

2.7 Upper-Confidence-Bound Action Selection

Upper confidence bound action selection is a way to explore actions that are not greedy but have the most potential to be optimal:

$$A_t \doteq \underset{a}{\text{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{7}$$

where $\ln t$ denotes the natural logarithm of t, $N_t(a)$ denotes the number of times that action a has been selected prior to time t, and the number $x > 0$ controls the degree of exploration. If $N_t(a) = 0$, then a is considere to be a maximizing action.

2.8 Gradient Bandit Algorithms

Actions can be selected according to a **preference**, independent of the associated values.

$$\Pr\{A_t = a\} \doteq \frac{\exp^{H_t(a)}}{\sum_{b=1}^{k} \exp^{H_t(b)}} \doteq \pi_t(a) \tag{8}$$

where $\pi_t(a)$ is the probability of taking action a at time t. Note that the **soft-max distribution** is introduced in this equation.

2.9 Associative Search (Contextual Bandits)

In **associative search** or **contextual bandits** tasks, the probability of reward for each action changes when the context changes. On way to solve this problem is to learn different values for each context.

3 Finite Markov Decision Processes (MDP)

3.1 The Agent-Environment Interface

In MDPs there is a finite number of states from a set $\mathcal{S}$ from which the agent is at any moment t. We have $S_t \in (S)$ for $t = 0, 1, 2, 3, ...$. Let's define $A_t \in \mathcal{A}(s)$, the set of possible actions in state s. In addition, each state has a reward $R_t \in \mathcal{R}$ associated to it.

For particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{9}$$

and

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \tag{10}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$.

3.2 Goals and Rewards

The goal of an RL agent is to maximize a signal called **reward**, a number $R_t \in \mathbb{R}$.

The **reward hypothesis**:
That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

3.3 Returns and Episodes

We seek to maximize the expected return, where the return is defined as follow:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{11}$$

where T is the final time step. This makes sense if there is indeed a final time step. If there is then the agent-environment interaction breaks into independent subsequences called **episodes**, and

the last state of an episode is the **terminal state**. These are **episodic tasks**, in contrast with **continuing tasks** in which we tend to discount future rewards. Here is the **discounted return**:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{12}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the **discount rate**.
Returns at successive time steps are related to each other:

$$G_t \doteq R_{t+1} + \gamma G_{t+1} \tag{13}$$

3.4 Unified Notation for Episodic and Continuing Tasks

In episodic tasks, there is an additional index i to the variables to refer to a specific episode, e.g. $S_{t,i}$ for the state at time t of episode i, however it's rarely useful.

Episodic and continuing tasks can be unified by adding an **absorbing state** to episodic tasks that always leads to itself and delivers no reward. Here is a unified notation for return:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{14}$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

3.5 Policies and Value Functions

Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Like p, π is an ordinary function; the "|" in the middle of $\pi(a|s)$ merely reminds that it defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in (S)$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

The value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right], \text{ for all } s \in (S) \tag{15}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. v_π is the **state-value function for policy** π.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a \right] \tag{16}$$

we call q_π the **action-value function for policy** π.

v_π and q_π can be estimated by experience, for example by averaging the returns in each action and state, this kind of method are **Monte Carlo methods** because they involve averaging over many random samples of actual returns.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already

established for the return. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S} \end{aligned} \tag{17}$$

where $a \in \mathcal{A}(s)$, the next states $s' \in \mathcal{S}$ and $r \in \mathcal{R}$. This is the **Bellman equation for** v_π.

3.6 Optimal Policies and Optimal Value Functions

π_* is the optimal policy that is better or equal to all other policies. Although there may be more than one, they share the same state-value function, called the optimal state-value function, denoted v_*, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{18}$$

for all $s \in \mathcal{S}$

Optimal policies also share the same optimal action-value function, denoted q_*, and defined as

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{19}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

We can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{20}$$

3.7 Optimality and Approximation

For many problems, optimal policies are rarely reached because their computational cost is too high. But for tasks with a limited number of states, it is possible to represent all the states in a table, they are the **tabular** cases. Though it is rarely the case that a problem can be expressed with only a few states. But in many problems, like the game backgammon, many states have a low probability of occurrence so it doesn't matter if the policy is suboptimal for these states because they are not likely to occur.

4 Dynamic Programming

Dynamic programming (DP) is a collection of recursive algorithms to compute optimal policies given a perfect model of the environment as a MDP. It's not practical for the problems that RL wants to save because of its assumption of a perfect model, but it is to inform RL theory and models which follow similar principles. The key idea of DP, and RL generally, is the use of value functions to organize and structure the search for good policies. We can easily obtain optimal policies once we have found the optimal value functions which satisfy the Bellman optimality equations:

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]
\end{aligned} \tag{21}
$$

or

$$
\begin{aligned}
q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{q_*}(s', a')]
\end{aligned} \tag{22}
$$

In this chapter, we assume finite MDPs.

4.1 Policy Evaluation (Prediction)

We use the Bellman equation to update the value of a state:

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right]
\end{aligned} \tag{23}
$$

for all $s \in \mathcal{S}$

From this we derive the **iterative policy evaluation** that allows us to determine the value function v_π for an arbitrary deterministic policy π.

Algorithm 2: Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input : π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

while $\Delta < \theta$ **do**
  $\Delta \leftarrow 0$
  for *each* $s \in (S)$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  end
end

4.2 Policy Improvement

First, here is the greedy policy:

$$
\begin{aligned}
\pi'(s) &\doteq \operatorname*{argmax}_a q_\pi(s, a) \\
&= \operatorname*{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]
\end{aligned} \tag{24}
$$

The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called **policy improvement**. Policy improvement gives us a strictly better policy except when the original policy is already optimal.

4.3 Policy Iteration

Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*} \tag{25}$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). This way of finding an optimal policy is called **policy iteration**.

Algorithm 3: Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

2. Policy Evaluation
while $\Delta < \theta$ *(a small positive number determining the accuracy of estimation)* **do**
  $\Delta \leftarrow 0$
  for *each* $s \in (S)$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
end

3. Policy Improvement
policy-stable $\leftarrow$ *true*
for *each* $s \in (S)$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  if *old-action* $\neq \pi(s)$ **then**
    policy-stable $\leftarrow false$
  end
end

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. 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 improvement is stopped after just one sweep (one update of each state). This algorithm is called **value iteration**:

$$
\begin{aligned}
v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]
\end{aligned} \tag{26}
$$

Algorithm 4: Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

while $\Delta < \theta$ **do**
  $\Delta \leftarrow 0$
  for *each* $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
end

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

4.5 Asynchronous Dynamic Programming

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once.

4.6 Generalized Policy Iteration (GPI)

It is simply the idea of alternating evaluation and policy improvement that allow convergence toward a better policy. The previous sections are already under the umbrella of the GPI.

4.7 Efficiency of Dynamic Programming

DP may not be practical for large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient.

5 Monte Carlo Methods

Here we do not assume complete knowledge of the environment, Monte Carlo methods require only experience. Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns.

5.1 Monte Carlo Prediction

To estimate the expected return – expected cumulative future discounted reward – from experience, then, is simply to average the returns observed after visits to that state. Below is a **first visit MC method** in which the returns for a state are calculated as the average of all the returns after the first visit to that state.

Algorithm 5: First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

while *True (for each episode)* **do**
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow G + R_{t+1}$
    if S_t *not in* $S_0, S_1, ..., S_{t-1}$ **then**
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$
    end
  end
end

There is also an **every visit** version of that algorithm, wherein the returns are calculated after every visit.

5.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate action values (the values of state–action pairs) rather than state values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state.

To make sure that every state-action pairs are explored, episodes can be started at different state-action pairs called **exploring starts**.

5.3 Monte Carlo Control

Here is a Monte Carlo version of the classical policy iteration:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*} \tag{27}$$

where $\xrightarrow{E}$ denotes a complete policy evaluation and $\xrightarrow{I}$ denotes a complete policy improvement. Policy evaluation is done exactly as described in the preceding section, with many episodes. Policy improvement is done by making the policy greedy with respect to the current value function. Here is the greedy policy:

$$\pi(s) \doteq \operatorname*{argmax}_a q(s, a) \tag{28}$$

Algorithm 6: Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

while *True (for each episode)* **do**
  Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0 (exploring starts)
  Generate an episode starting from S_0, A_0, following π:
  $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow G + R_{t+1}$
    if (S_t, A_t) *not in* $(S_0, A_0), (S_1, A_1)..., (S_{t-1}, A_{t-1})$ **then**
      Append G to $Returns(S_t, A_t)$
      $Q(S_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$
    end
  end
end

5.4 Monte Carlo Control without Exploring Starts

To explore all state-action values without the unlikely assumption of exploring starts, we can force all actions probabilities to be above 0: $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$.

An example is the ϵ-greedy, where all non greedy actions are given the minimal probability of selection, $\frac{\epsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. This is an example of ϵ-$soft$ policies for which $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $1 > \epsilon > 0$.

Algorithm 7: On-Policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\epsilon > 0$
Initialize:
$\pi \leftarrow$ an arbitrary ϵ-policy
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

while *True (for each episode)* **do**
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow G + R_{t+1}$
    if $(S_t, A_t) \notin (S_0, A_0), (S_1, A_1) ..., (S_{t-1}, A_{t-1})$ **then**
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken arbitrarily)
      for *all* $a \in \mathcal{A}(S_t)$ **do**
        $\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
      end
    end
  end
end

Any ϵ-greedy policy with respect to q_π is an improvement over any ϵ-soft policy π (or as good as).

5.5 Off-policy Prediction via Importance Sampling

Exploring actions are necessary to find an optimal policy but are not part of an optimal policy, so it is necessary to split policy into a **target policy** (converging towards the optimal policy), and a **behavior policy** to learn the values and improve the target policy. **On-policy** methods are a special case of off-policy where the target and behavior policies are one and the same. With **Off-policy**, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function, and becomes a deterministic optimal policy, while the behavior policy remains stochastic and more exploratory, for example, an ϵ-greedy policy. Note that the behavior policy must have a nonzero probability of selecting all actions that might be selected by the target policy. This is called **coverage**.

Let's consider the prediction problem in which we want to estimate v_π or q_π, and for now let's assume that the target policy π and the behavior policy b are fixed. Almost all off-policy methods utilize **importance sampling**, a general technique for estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the **importance-sampling ratio**.

Given a starting state S_t, the probability of the subsequent state–action trajectory, $A_t, S_{t+1}, A_{t+1}, ..., S_T$ occurring under any policy π is

$$Pr\{A_t, S_{t+1}, A_{t+1}, ..., S_T \mid S_t, A_{t:T-1} \sim \pi\} = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1})...p(S_T|S_{T-1}, A_{T-1})$$
$$= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k) \tag{29}$$

where p here is the state-transition probability function from the MDP. Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{30}$$

To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|} \tag{31}$$

where $\mathcal{T}(s)$ are all the time step in which state s is visited.

But a better alternative is the **weighted importance sampling**, which is a weighted average:

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}} \tag{32}$$

It is biased but has a bounded variance which produces better estimates of value with fewer episodes.

5.6 Incremental Implementation

Suppose we have a sequence of returns $G_1, G_2, ..., G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i (e.g., $W_i = \rho_{t:T(t)-1}$). We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2. \tag{33}$$

and keep it up-to-date as we obtain a single additional return G_n. In addition to keeping track of V_n, we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}[G_n - V_n], \quad n \geq 1. \tag{34}$$

and

$$C_{n+1} \doteq C_n + W_{n+1}, \tag{35}$$

where $C_0 \doteq 0$ (and V_1 is arbitrary and thus need not be specified).

Algorithm 8: Off-Policy MC prediction (policy evaluation), estimates $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$

while *True (for each episode)* **do**
  $b \leftarrow$ any policy with coverage of π
  Generate an episode following b: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
    $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    if $W = 0$ **then**
      exit **for** loop
    end
  end
end

5.7 Off-policy Monte Carlo Control

The previous section was about off-policy evaluation, now we want to also update the target policy.

Algorithm 9: Off-Policy MC prediction (policy evaluation), estimates $Q \approx q_\pi$

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(s,a)$ (with ties broken consistently)

while *True (for each episode)* **do**
  $b \leftarrow$ any soft policy with coverage of π
  Generate an episode following b: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
    $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
    if $A_t \neq \pi(S_t)$ **then**
      exit **for** loop
    end
    $W \leftarrow W \frac{1}{b(A_t|S_t)}$
  end
end

6 Temporal-Difference Learning

Temporal-difference (TD) learning uses concepts from dynamic programming and Monte Carlo (MC) methods. They are all based on the generalized policy iteration, so the main difference is on the evaluation or prediction problem.

6.1 TD Prediction

MC methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit MC method suitable for non-stationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{36}$$

where G_t is the actual return following time t, and α is a constant step-size parameter. Whereas the MC method must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{37}$$

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This method is called **TD(0)**, or **one-step TD**, because it is a special case of the TD(λ) and n-step TD methods.

Algorithm 10: Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

while *True (for each episode)* **do**
  Initialize S **for** *each step of episode* **do**
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  end
  until S is terminal
end

Because TD(0) bases its update in part on existing estimate, we say that it is a bootstrapping method, like DP. TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. We refer to TD and Monte Carlo updates as sample updates because they involve looking ahead to a sample successor state (or state–action pair).

Note that the update include the **TD error**, that arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{38}$$

The TD error, is the error in $V(S_t)$ available at time $t + 1$.

6.2 Advantages of TD Prediction Methods

TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions. In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks.

6.3 Optimality of TD(0)

With TD methods, given a finite amount of experience, increments to a specific part of the value function are computed at every time step, but the value function is changed only once, by the sum of all the increments. In **batch updating** all the available experience is processed repeatedly until the value function converges.

Whereas batch MC methods will give a lower RMS error on the training data, it does not take into account value information that is related to the Markov process nature of the environment because the value of each state is computed only from the specific sequence of states that were experienced. On the other hand, the TD algorithm takes into account value information that is dependent on the underlying Markov process. Given a MDP, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the **certainty-equivalence estimate** because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate. This is the reason why TD methods

converge faster than MC methods. Also, on tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

6.4 Sarsa: On-policy TD Control

We are now interested in learning the value of state-action pairs (action values) instead of states.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{39}$$

This update is done after every transition from a non-terminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that gives rise to the name Sarsa.

Algorithm 11: Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$

while *True (for each episode)* **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  for *each step of episode* **do**
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'; A \leftarrow A';$
  end
  until S is terminal
end

6.5 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

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{40}$$

Note: unlike Sarsa, the action value is updated with the maximum action value of the following state, which is not necessarily the value of the action chosen by the behavior policy, hence it is an off-policy method.

Algorithm 12: Q-learning (off-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$

while *True (for each episode)* **do**
  Initialize S
  for *each step of episode* **do**
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  end
  until S is terminal
end

6.6 Expected Sarsa

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\big] \\ &\leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\big] \end{aligned} \tag{41}$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1}, this algorithm moves deterministically in the same direction as Sarsa moves in expectation, and accordingly it is called Expected Sarsa.

6.7 Maximization Bias and Double Learning

Maximization is involved in the computation of target policies. For example, in Q-learning, the target policy is the greedy policy given the current action values, which is defined with a max. In some cases, for example when the reward associated with an action is randomly distributed, the value of an action will be overestimated, and this overestimation will be passed on to previous state-action, shifting the behavior towards actions that in fact have a lower return.

To prevent this, there exists a method called **Double Learning**, in which two sets of action-values Q_1 and Q_2 are learned.

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \operatorname*{argmax}_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \tag{42}$$

Algorithm 13: Q-learning (off-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$

while *True (for each episode)* **do**
  Initialize S
  for *each step of episode* **do**
    Choose A from S using the policy ϵ-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probability:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + \gamma Q_2(S', \text{argmax}_a\, Q_1(S', a)) - Q_1(S, A)]$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[R + \gamma Q_1(S', \text{argmax}_a\, Q_2(S', a)) - Q_2(S, A)]$
    $S \leftarrow S'$
  end
  until S is terminal
end

6.8 Games, Afterstates, and Other Special Cases

In some cases (e.g. the tic-tac-toe game) the environment is better represented with afterstate-value functions than with action-values functions, when different state-action pairs lead to the same afterstate (state of the environment after an action was taken in a given state). In this case, afterstate value function will assess state-action pairs that lead to the same afterstate equally immediately after an action that led to that afterstate. Generalized policy iteration applies to afterstate value functions as well, with on- or off-policy methods.

7 n-step Bootstrapping

So far the TD learning algorithm were looking one step back in the past to update the value of an action. n-step TD learning look back n steps in the past.

7.1 n-step TD Prediction

Here is the MC method (refresher):

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T \tag{43}$$

and here is what a one-step return looks like (refresher as well):

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}) \tag{44}$$

$G_{t:t+1}$ indicate that this is a truncated return and $V_t(S_{t+1})$ takes the place of the subsequent returns.

Now, here is what a two-step return looks like:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}) \tag{45}$$

and more generally a n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{46}$$

If $t + n \geq T$ (if the n-step return extends to or beyond termination), then all the missing terms are taken as zero, and the n-step return defined to be equal to the ordinary full return ($G_{t:t+n} \doteq G_t$ if $t + n \geq T$).

Of course, the update to the value function happens only n-step after a state is encountered:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T \tag{47}$$

here (as a reminder), $V_{t+n}(S_t)$ is the value at time $t + n$ of the state S_t. The values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$ for all $s \neq S_t$. We call this algorithm **n-step TD**.

Algorithm 14: n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

while *True (for each episode)* **do**
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  for *each step of episode* **do**
    if $t < T$ **then**
      Take an action according to $\pi(.|S_t)$
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      if S_{t+1} is terminal, then $T \leftarrow t + 1$
    end
    $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    if $\tau \geq 0$*:* **then**
      $G \leftarrow \sum_{i=\tau+1}^{min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      if $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
      $V(S_\tau) \leftarrow V(S_\tau) + \alpha[G - V(S_\tau)]$
    end
  end
  until $\tau = T - 1$
end

The expectation of returns with the n-step TD is guaranteed to be better than with one-step TD, which means that the n-step approach also converges.

The right number 'n' of steps in the n-step TD will vary depending on the task, and is usually intermediate between MC method and one-step TD.

7.2 n-step Sarsa

We already covered Sarsa, which was a special case that we can call the one-step Sarsa or Sarsa(0). The main idea is to simply switch states for actions (state–action pairs) and then use an ϵ-greedy policy.

We redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T - n, \quad (48)$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. The natural algorithm is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \quad (49)$$

Algorithm 15: n-step Sarsa estimating $Q \approx q_*$ or q_π

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ϵ-greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t and R_t) can take their index mod n

while *True (for each episode)* **do**
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  for *each step of episode* **do**
    if $t < T$ **then**
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      if S_{t+1} *is terminal* **then**
        $T \leftarrow t + 1$
      else
        Select and store an action $A_{t+1} \sim \pi(.|S_{t+1})$
      end
    end
    $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    if $\tau \geq 0$*:* **then**
      $G \leftarrow \sum_{i=\tau+1}^{min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      if $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $\quad (G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(.|S_\tau)$ is ϵ-greedy wrt Q
    end
  end
  Until $\tau = T - 1$
end

Expected Sarsa can also be formulated in the context of n-step TD:

$$G_{t:t+n} \doteq R_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \sum_a \pi(a|S_{t+n}) Q_{t+n-1}(S_{t+n}, a) \quad (50)$$

for all n and t such that $n \geq 1$ and $0 \geq t \geq T - n$.

7.3 n-step Off-policy Learning by Importance Sampling

To make a simple off-policy version of n-step TD, the update for time t (actually made at time $t + n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T \tag{51}$$

where $\rho_{t:t+n-1}$, called the importance sampling ratio, is the relative probability under the two policies of taking the n actions from A_t to A_{t+n-1}:

$$\rho_{t:h} \doteq \prod_{k=t}^{min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{52}$$

Note that if any one of the actions would never be taken by π (i.e., $\pi(A_k|S_k) = 0$) then the n-step return should be given zero weight and be totally ignored.

Here is an n-step Sarsa with importance sampling:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\rho_{t+1:t+n-1}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \tag{53}$$

for $0 \leq t < T$. Note that the importance sampling ratio starts one step later than for n-step TD because we are updating a state-action pair. The n-step Expected Sarsa with importance sampling

would have one less factor: $\rho_{\text{t+1:t+n-2}}$ instead of $\rho_{\text{t+1:t+n-1}}$.

Algorithm 16: Off-policy n-step Sarsa estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t, A_t and R_t) can take their index mod $n+1$

while *True (for each episode)* **do**
  Initialize and store $S_0 \neq$ terminal
  Select and store and action $A_0 \sim b(.|S_o)$
  $T \leftarrow \infty$
  for *each step of episode* **do**
    if $t < T$ **then**
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      if S_{t+1} *is terminal* **then**
        $T \leftarrow t+1$
      else
        Select and store an action $A_{t+1} \sim b(.|S_{t+1})$
      end
    end
    $\tau \leftarrow t-n+1$ (τ is the time whose state's estimate is being updated)
    if $\tau \geq 0$*:* **then**
      $\rho \leftarrow \prod_{i=\tau+1}^{min(\tau+n-1,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:t+n-1})$
      $G \leftarrow \sum_{i=\tau+1}^{min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      if $\tau+n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho[G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(.|S_\tau)$ is ϵ-greedy wrt Q
    end
  end
  Until $\tau = T-1$
end

7.4 Per-decision Off-policy Methods with Control Variates

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

So far in the n-step TD algorithms, we have always updated the estimated value of a state or action toward a target combining the rewards of successive steps and the estimated values of the last step. In the tree-backup update, the target includes all these things plus the estimated values of the unchosen actions at every step. This is why it is called a **tree-backup update**; it is an update from the entire tree of estimated action values.

The value of unchosen actions are weighted by the probability of choosing their corresponding actions obtained from the probability of the successive actions that lead to the state of the unchosen

action. Each first-level unchosen action a contributes with a weight of $\pi(a|S_{t+1})$, and each second-level unchosen action contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$, etc.

The one-step return (target) of the tree backup algorithm is the same as that of Expected Sarsa:

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a), \quad t < T-1 \tag{54}$$

and the two-step tree-backup return is:

$$\begin{aligned} G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) \left(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \right) \\ &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}, \quad t < T-2. \end{aligned} \tag{55}$$

The latter form suggest the general recursive definition of the tree-backup n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \quad t+1 < T, n > 1. \tag{56}$$

with the $n = 1$ case handled by $G_{T-1:T} \doteq R_T$. This target is then used with the usual action-value update rule from n-step Sarsa (same equation):

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \tag{57}$$

Algorithm 17: n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t, A_t and R_t) can take their index mod $n+1$

while *True (for each episode)* **do**
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  for *each step of episode* **do**
    if $t < T$ **then**
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      if S_{t+1} *is terminal* **then**
        $T \leftarrow t+1$
      else
        Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
      end
    end
    $\tau \leftarrow t-n+1$ (τ is the time whose state's estimate is being updated)
    if $\tau \geq 0$ **then**
      if $t+1 \geq T$ **then**
        $G \leftarrow R_T$
      else
        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
      end
      for $k = min(t, T-1)$ *down through* $\tau+1$ **do**
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k, S_k) G$
      end
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(.|S_\tau)$ is greedy wrt Q
    end
  end
  Until $\tau = T-1$
end

8 Planning and Learning with Tabular Methods

This chapter is about **model-based** and **model-free** RL. Dynamic programming and heuristic searches are model based and, Monte Carlo and temporal difference methods are model free. This chapter aims to unify these approaches.

8.1 Models and Planning

A model predicts the resulting state and reward of an action in a given state. **Distribution models** include all possibilities and their probabilities, like dynamic programming, and **sample**

models produce just one of the possibilities, sampled according to the probabilities. Distribution models are more accurate and complete but more complicated to obtain. In either case, the model is used to simulate the environment and produce simulated experience. **Planning** refers to using a model to produce or improve a policy. Here planning refers to **state-space planning** which is a search through the state space for an optimal policy, as opposed to **plan-space planning** which is a search through the space of plans. There are two basic ideas:

- all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy
- they compute value functions by updates or backup operations applied to simulated experience.

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value.

The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. But here is an example of their similarity: **random-sample one-step tabular Q-planning** converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment.

Algorithm 18: Random-sample one-step tabular Q-planning

while *True (forever)* **do**
1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

end

Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Dyna: Integrating Planning, Acting, and Learning

When planning is done on-line, while interacting with the environment, experience can be used to update the model. To address this approach we look at Dyna-Q, a simple architecture integrating the major functions needed in an on-line planning agent. Within a planning agent, real experience is used for **model learning** and **direct reinforcement learning**. **Indirect reinforcement learning** corresponds to planning.

Dyna-Q is an example of a planning agent, that plans with the above random-sample one-step tabular Q-planning, uses one-step tabular Q-learning for direct RL and model learning is table-based and assumes the environment is deterministic. The table is filled in with experience. With more experience, the transitions in the environment are defined and planning improves.

We use the term **search control** to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Planning and learning from real

experience use the same method to update value and policy.

Algorithm 19: Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(a)$
while *True (forever)* **do**
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \epsilon$-greedy(S,Q)
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) **for** *n times* **do**
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  end
end

8.3 When the Model Is Wrong

The Dyna-Q algorithm assumes a static and deterministic environment. This assumption leads this agent to be rather inflexible to changes to the environment, or at least, to take many steps to react appropriately.

An improved algorithm, **Dyna-Q+**, prioritize planning associated to transitions that have not been explored in a while. Transitions (s,a) are more and more likely to be explored with time, their probability increases with $r + \kappa\sqrt{\tau}$ where r is the reward associated with this transition, κ is a constant (parameter) and τ is the number of time steps since the transition was last explored.

This algorithm reacts faster to changes to the environment, especially if the previously optimal policy is still valid but no longer optimal.

8.4 Prioritized Sweeping

Dyna agents update there value function with simulated transitions by selecting a state and action at random. It's a rather inefficient way of updating values since a lot of states have the same value, e.g. when the agent is just learning about the environment most of the states have value 0, assuming that this is the initial value given to all states. A better way is to explore the states that lead to a state whose value just changed: this is **backward focusing** of planning computations.

Prioritized sweeping is used to update first the states whose value changed the most.

Algorithm 20: Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$ and $Model(s,a)$ for all s,a, and $PQueue$ to empty
while *True (forever)* **do**
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow policy(S,Q)$
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Model(S,A) \leftarrow R, S'$
  (e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,a)|$
  (f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
  (g) **for** *n times, while PQueue is not empty* **do**
    $S,A \leftarrow first(PQueue)$
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    for *all* $\bar{S}, \bar{A}$ *predicted to lead to* S **do**
      $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
      $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$
      if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P
    end
  end
end

This is issue with prioritize sweeping is that it updates all transitions to a state whose value changed, even the transitions with low probability. Updating value based on sampling by generating sample transitions from the model is a way to save computations by focusing more on the higher probability transitions.

In contrast to backward focusing, **forward focusing** prioritize the state that are easily reached from the states that are visited frequently under the current policy.

8.5 Expected vs. Sample Updates

The one-step updates that have been reviewed so far fall into two categories: expected or sample updates.

The expected update for a state-action pair, s,a, is:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a) \left[r + \gamma \max_{a'} Q(s',a') \right] \tag{58}$$

The corresponding sample update for s,a, given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{59}$$

where α is the usual positive step-size parameter.

One-step updates also differ on the value they estimate: the state values $v_\pi(s)$, $v_*(s)$, or the action values $q_\pi(s,a)$, $q_*(s,a)$, following a given policy π or the optimal policy:

Value estimated	Expected updates (DP)	Sample updates (one-step TD)
$v_(s)$	policy evaluation	TD(0)
$v_*(s)$	value iteration	
$q_\pi(s,a)$	q-policy evaluation	Sarsa
$q_*(s,a)$	q-value iteration	Q-learning

Expected updates yield better estimates but are more computationally expensive, especially when branching is high and when the environment is stochastic.

8.6 Trajectory Sampling

Expected sample methods are inefficient because they have to produce a complete sweep of the state (or state-action) space for a single iteration which is problematic on large tasks, and because they sweep through states that are unlikely to be visited under an optimal policy. One way to sweep through relevant states with planning is to perform **trajectory sampling** where trajectories are generated with the model of state transitions following the current policy.
Experiments with different branching factors show that trajectory sampling updates the initial state value faster than exhaustive search but hurt the update in the long run because it focuses too much on on-policy states.

8.7 Real-time Dynamic Programming