Notes Reinforcement Learning: An Introduction

Contents

1 Introduction

Reinforcement learning is about how an agent can learn to interact with its environment. Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards.

1.3 Elements of Reinforcement Learning

Policy defines the way that an agent acts, it is a mapping from perceived states of the world to actions. It may be stochastic.

Reward defines the goal of the problem. A number given to the agent as a (possibly stochastic) function of the state of the environment and the action taken.

Value function specifies what is good in the long run, essentially to maximise the expected reward. The central role of value estimation is arguably the most important thing that has been learned about reinforcement learning over the last six decades.

Model mimics the environment to facilitate planning. Not all reinforcement learning algorithms have a model (if they don't then they can't plan, i.e. must use trial and error, and are called model free).

2 Multi-armed Bandits

Reinforcement learning involves evaluative feedback rather than instructive feedback. We get told whether our actions are good ones or not, rather than what the single best action to take is. This is a key distinction between reinforcement learning and supervised learning.

2.1 A k-armed Bandit Problem

In the k-armed bandit problem there are k possible actions, each of which yields a numerical reward drawn from a stationary probability distribution for that action. We want to maximise the expected total reward, taking an action at each *time step*. Some notation:

- Index timesteps by t
- Action A_t
- Corresponding reward R_t
- *Value* of action a is $q_*(a) = \mathbb{E}[R_t|A_t = a]$
- Estimate of value of action a at t is denoted $Q_t(a)$

We therefore want to choose $\{a_1, \ldots, a_T\}$ to maximise $\sum_{t=1}^{T} q_*(a_t)$.

At each timestep, the actions with the highest estimated reward are called the *greedy* actions. If we take this action, we say that we are *exploiting* our understanding of the values of actions. The other actions are known as *non-greedy* actions, sometimes we might want to take one of these to improve our estimate of their value. This is called *exploration*. The balance between exploration and exploitation is a key concept in reinforcement learning.

2.2 Action-value Methods

We may like to form estimates of the values of possible actions and then choose actions according to these estimates. Methods such as this are known as *action-value methods*. There are, of course, many ways of generating the estimates $Q_t(a)$.

An ε-greedy method is one in which with probability ε we take a random draw from all of the actions (choosing each action with equal probability), providing some exploration.

2.5 Tracking a Non-stationary Problem

If we decide to implement the sample average method, then at each iteration that we choose the given action we update our estimate by

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n] \tag{1}$$

Note that this has the (soon to be familiar) form

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} \times [\text{Target} - \text{OldEstimate}]. \tag{2}$$

If the problem was non-stationary, we might like to use a time weighted exponential average for our estimates (*exponential recency-weighted average*). This corresponds to a constant step-size $\alpha \in (0, 1]$ (you can check).

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]. \tag{3}$$

We might like to vary the step-size parameter. Write $\alpha_n(a)$ for the step-size after the n^{th} reward from action a. Of course, not all choices of $\alpha_n(a)$ will give convergent estimates of the values of a. To converge with probability 1 we must have

$$\sum_n \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_n \alpha_n(a)^2 < \infty. \tag{4}$$

Meaning that the coefficients must be large enough to recover from initial fluctuations, but not so large that they don't converge in the long run. Although these conditions are used in theoretical work, they are seldom used in empirical work or applications. (Most reinforcement learning problems have non-stationary rewards, in which case convergence is undesirable.)

2.6 Optimistic Initial Values

The exponential recency weighted method is biased by the initial value one gives. If we like, we may set initial value estimates artificially high to encourage exploration in the short run – this is called *optimistic initial values*. This is a useful trick for stationary problems, but does not apply so well to non-stationary problems as the added exploration is only temporary.

2.7 Upper-Confidence Bound Action Selection

We might like to discriminate between potential explorative actions. Note that ε-greedy does not do this. We define the *upper-confidence bound* action at t as follows

$$A_t \doteq \operatorname*{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln(t)}{N_t(a)}} \right] \tag{5}$$

where $Q_t(a)$ is the value estimate for the action a at time t, $c > 0$ is a parameter that controls the degree of exploration and $N_t(a)$ is the number of times that a has been selected by time t. If $N_t(a) = 0$ then we consider a a maximal action.

This approach favours actions with a higher estimated rewards but also favours actions with uncertain estimates (more precisely, actions that have been chosen few times).

2.8 Gradient Bandit Algorithms

Suppose that we choose actions probabilistically based on a preference for each action, $H_t(a)$. Let the action at t be denoted by A_t. We then define the probability of choosing action a via the softmax

$$\pi_t(a) \doteq \mathbb{P}(A_t = a) = \frac{e^{H_t(a)}}{\sum_i e^{H_t(i)}}. \tag{6}$$

We then iteratively perform updates according to

$$H_{t+1}(a) = H_t(a) + (R_t - \bar{R}_t)(\mathbb{1}_{A_t=a} - \pi_t(a)), \tag{7}$$

where $\bar{R}_t$ is the mean of previous rewards. The box in the notes shows that this is an instance of stochastic gradient ascent since the expected value of the update is equal to the update when doing gradient ascent on the (total) expected reward.

3 Finite Markov Decision Processes

We say that a system has the *Markov property* if each state includes all information about the previous states and actions that makes a difference to the future.

The MDP provides an abstraction of the problem of goal-directed learning from interaction by modelling the whole thing as three signals: action, state, reward.

Together, the MDP and agent give rise to the *trajectory* $S_0,\ A_0,\ R_1,\ S_1,\ A_1,\ S_2,\ R_2,\ \ldots$. The action choice in a state gives rise (stochastically) to a state and corresponding reward.

3.1 The Agent–Environment Interface

We consider finite Markov Decision Processes (MDPs). The word finite refers to the fact that the states, rewards and actions form a finite set. This framework is useful for many reinforcement learning problems.

We call the learner or decision making component of a system the *agent*. Everything else is the *environment*. General rule is that anything that the agent does not have absolute control over forms part of the environment. For a robot the environment would include it's physical machinery. The boundary is the limit of absolute control of the agent, not of its knowledge.

The MDP formulation is as follows. Index time-steps by $t \in \mathbb{N}$. Then actions, rewards, states at t represented by $A_t \in \mathcal{A}(s)$, $R_t \in \mathcal{R} \subset \mathbb{R}$, $S_t \in \mathcal{S}$. Note that the set of available actions is dependent on the current state.

A key quantity in an MDP is the following function, which defines the *dynamics* of the system.

$$p(s', r|s, a) \doteq \mathbb{P}(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{8}$$

From this quantity we can get other useful functions. In particular we have the following:

state-transition probabilities

$$p(s'|s, a) \doteq \mathbb{P}(S_t = s'|S_{t-1} = s, A_{t-1} = A) = \sum_{r\in\mathcal{R}} p(s', r|s, a) \tag{9}$$

note the abuse of notation using p again; and,

expected reward

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a). \tag{10}$$

3.2 Goals and rewards

We have the *reward hypothesis*, which is a central assumption in reinforcement learning:

> All of what we mean by goals and purposes can be well thought of as the maximisation of the expected value of the cumulative sum of a received scalar signal (called reward).

3.3 Returns and Episodes

Denote the sequence of rewards from time t as R_{t+1}, R_{t+2}, R_{t+3}, ... We seek to maximise the *expected return* G_t which is some function of the rewards. The simplest case is where $G_t = \sum_{\tau>t} R_\tau$.

In some applications there is a natural final time-step which we denote T. The final time-step corresponds to a *terminal state* that breaks the agent-environment interaction into subsequences called *episodes*. Each episode ends in the same terminal state, possibly with a different reward. Each starts independently of the last, with some distribution of starting states. We denote the set of states including the terminal state as $\mathcal{S}^+$

Sequences of interaction without a terminal state are called *continuing tasks*.

We define G_t using the notion of *discounting*, incorporating the *discount rate* $0 \leq \gamma \leq 1$. In this approach the agent chooses A_t to maximise

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{11}$$

This sum converges wherever the sequence R_t is bounded. If $\gamma = 0$ the agent is said to be myopic. We define $G_T = 0$. Note that

$$G_t = R_{t+1} + \gamma G_{t+1}. \tag{12}$$

Note that in the case of finite time steps or an episodic problem, then the return for each episode is just the sum (or whatever function) of the returns in that episode.

3.4 Unified Notation for Episodic and Continuing Tasks

We want to unify the notation for episodic and continuing learning.

We introduce the concept of an *absorbing state*. This state transitions only to itself and gives reward of zero.

To incorporate the (disjoint) possibilites that $T = \infty$ or $\gamma = 1$ in our formulation of the return, we might like to write

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k. \tag{13}$$

3.5 Policies & Value Functions

Policy

A *policy* $\pi(a|s)$ is a mapping from states to the probability of selecting actions in that state. If an agent is following policy π and at time t is in state S_t, then the probability of taking action A_t is $\pi(a|s)$. Reinforcement learning is about altering the policy from experience.

Value Functions

As we have seen, a central notion is the value of a state. The *state-value function* of state s under policy π is the expected return starting in s and following π thereafter. For MDPs this is

$$v_\pi \doteq \mathbb{E}_\pi[G_t | S_t = s], \tag{14}$$

where the subscript π denotes that this is an expectation taken conditional on the agent following policy π.

Similarly, we define the *action-value function* for policy π to be the expected return from taking action a in state s and following π thereafter

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{15}$$

The value functions v_π and q_π can be estimated from experience.

Bellman Equation

The Bellman equations express the value of a state in terms of the value of its successor states. They are a consistency condition on the value of states.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] && (16)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] && (17)\\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] && (18)\\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] && (19) \end{aligned}$$

The value function v_π is the unique solution to its Bellman equation.

3.6 Optimal Policies & Optimal Value Functions

We say that $\pi \geq \pi'$ iff $v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$. The policies that are optimal in this sense are called optimal policies. There may be multiple optimal policies. We denote all of them by π_*.

The optimal policies share the same optimal value function $v_*(s)$

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}. \tag{20}$$

They also share the same optimal action-value function $q_*(s,a)$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s), \tag{21}$$

this is the expected return from taking action a in state s and thereafter following the optimal policy.

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]. \tag{22}$$

Since v_* is a value function, it must satisfy a Bellman equation (since it is simply a consistency condition). However, v_* corresponds to a policy that always selects the maximal action. Hence

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]. \tag{23}$$

Similarly,

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a] && (24)\\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]. && (25) \end{aligned}$$

Note that once one identifies an optimal value function v_*, then it is simple to find an optimal policy. All that is needed is for the policy to act greedily with respect to v_*. Since v_* encodes all information on future rewards, we can act greedily and still make the long term optimal decision (according to our definition of returns).

Having q_* is even better since we don't need to check $v_*(s')$ in the succeeding states s', we just find $a_* = \operatorname{argmax}_a q_*(s, a)$ when in state s.

4 Dynamic Programming

The term Dynamic Programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given perfect model of the environment as a Markov Decision Process (MDP). DP methods tend to be computationally expensive and we often don't have a perfect model of the environment, so they aren't used in practice. However, they provide useful theoretical basis for the rest of reinforcement learning.

Unless stated otherwise, will assume that the environment is a finite MDP. If the state or action space is continuous, then we will generally discretise it and apply finite MDP methods to the approximated problem.

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. We use DP and the Bellman equations to find optimal value functions.

4.1 Policy Evaluation (Prediction)

We can use the Bellman equation for the state-value function v_π to construct an iterative updating procedure.

Iterative Policy Evaluation

Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$ each mapping $\mathcal{S}^+$ to $\mathbb{R}$. The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π as an update rule:

$$v_{k+1} \doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \tag{26}$$

$$= \sum_a \pi(s|a) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s') \right] \tag{27}$$

Clearly, $v_k = v_\pi$ is a fixed point. The sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*. This update rule is an instance of an *expected update* because it performs the updates by taking an expectation over all possible next states rather than by taking a sample next state.

4.2 Policy Improvement

Policy Improvement Theorem

Let π, π' be any pair of deterministic policies, such that

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \forall s \in \mathcal{S}. \tag{28}$$

That is, π' is as least as good as π. Then we have (shown below)

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \in \mathcal{S} \tag{29}$$

so π' gives at least as good (expected) return as π.

The argument below also shows that if $q_\pi(s, \pi'(s)) > v_\pi(s)$ at any s, then there is at least one s for which $v_{\pi'}(s) > v_\pi(s)$.

proof:

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s] \\
&= v_{\pi'}(s)
\end{aligned}$$

Policy Improvement Algorithm

Now consider a policy that is greedy with respect to $q_\pi(s, a)$. Define

$$\pi'(s) = \operatorname*{argmax}_a q_\pi(s, a) \tag{30}$$

$$= \operatorname*{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{31}$$

$$= \operatorname*{argmax}_a \sum_{s', r} p(s', r | s, a)[r + \gamma v_\pi(s')]. \tag{32}$$

Now we can use v_π to get $\pi' \geq \pi$, then use $v_{\pi'}$ to get *another* policy. (In the above, ties are broken arbitrarily when the policy is deterministic. If the policy is stochastic, we accept any policy that assigns zero probability to sub-optimal actions.)

Note that by construction

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

therefore

$$v_{\pi'} \geq v_\pi$$

so we get from this process a monotonically increasing sequence of policies.

Note also that if π' is as good as π then $v_{\pi'} = v_\pi$ and $\forall s \in \mathcal{S}$

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

which is the Bellman optimality condition for v_*, so both π and π' are optimal. This means that policy improvement gives a strictly better policy unless the policy is already optimal.

The policy improvement theorem holds for stochastic policies too, but we don't go into that here.

4.3 Policy Iteration

We can exploit policy improvement iteratively to get the policy iteration algorithm.

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
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

A finite MDP has only a finite number of policies (as long as they are deterministic, of course) so this process is guaranteed to converge.

4.4 Value Iteration

Policy iteration can be slow because each iteration involves running the entire policy evaluation until convergence.

It turns out that one can truncate the policy evaluation step of policy iteration in many ways without losing convergence guarantees. One special case of this is *value iteration*, where we truncate policy evaluation after only one update of each state. This algorithm converges to v_* under the same conditions that guarantee the existence of v_*.

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 $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

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

Note the $\max_a$ in the assignment of $V(s)$, since we only one sweep of the state space and then choose the greedy policy.

It may be more efficient to interpose multiple policy evaluation steps in between policy improvement iterations, all of these algorithms converge to an optimal policy for discounted finite MDPs.

4.5 Asynchronous Dynamic Programming

The DP methods that we have described so far all involve a full sweep of the state space on each iteration. This is potentially a very costly procedure.

Asynchronous DP algorithms update the values in-place and cover states in any order whatsoever. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it can't ignore any state after some point in the computation.

Asynchronous DPs give a great increase in flexibility, meaning that we can choose the updates we want to make (even stochastically) based on the interaction of the agent with the environment. This procedure might not reduce computation time in total if the algorithm is run to convergence, but it could allow for a better rate of progress for the agent.

4.6 Generalised Policy Iteration

We use the term *generalised policy iteration* (GPI) to refer to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI, including the policy iteration algorithms we have discussed in this section. GPI works via the competing but complementary nature of the two processes. In some cases it can be guaranteed to converge.

4.7 Efficiency of Dynamic Programming

If we ignore a few technical details, then the (worst case) time DP methods take to find an optimal policy is polynomial in the number of states and actions. Compare this to the searching the states directly, which is exponential.

5 Monte Carlo Methods

Monte Carlo methods learn state and action values by sampling and averaging returns (i.e. not from dynamics like DP). These methods learn from experience (real or simulated) and require no prior knowledge of the environments dynamics.

Monte Carlo methods thus require well defined returns, so we will consider them only for episodic tasks. Only on completion of an episode do values and policies change.

We still use the generalised policy iteration framework, but we adapt it so that we learn the value function from experience rather than compute it *a priori*.

5.1 Monte Carlo Prediction

The idea is to average the returns following each state to get an estimate of the state value

$$v_\pi(s) = \mathbb{E}_\pi[G_{t+1}|S_t = s].$$

Given enough observations, the sample average converges to the true state value under the policy π.

Given a policy π and a set of episodes, here are two ways in which we might estimate state values

- *First Visit MC* average returns from first visit to state s in order to estimate $v_\pi(s)$
- *Every Visit MC* average returns following every visit to state s.

First visit MC generates iid estimates of $v_\pi(s)$ with finite variance, so the sequence of estimates converges to the expected value by the law of large numbers as visits to s tend to ∞. Every visit MC does not generate independent estimates, but still converges.

An algorithm for first visit MS (what we will focus on) is below. Every visit is the same, just without the check for S_k occurring earlier in the episode.

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 $s \in \mathcal{S}$

Loop forever (for each episode):
    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, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

Monte Carlo methods are often used even when the dynamics of the environment are knowable, e.g. in Blackjack. It is often much easier to create sample games than it is to calculate environment dynamics directly.

MC estimates for different states are independent (unlike bootstrapping in DP). This means that we can use MC to calculate the value function for a subset of the states, rather than the whole state space as with DP. Along with the ability to learn from experience and simulation, this is the another advantage that MC has over DP.

5.2 Monte Carlo Estimation of Action Values

If we don't have a model for the environment, then it is more useful to estimate action-values. With a model we can use state values to find a policy by searching possible actions, as with DP (value iteration, etc.). We can't do this without knowledge of the dynamics, so one of the primary goals of MC is to estimate q_*. We start with policy evaluation for action-values.

Policy Evaluation for Action-Values

The policy evaluation problem for action-values is to estimate $q_\pi(s, a)$ for some π. This is essentially the same as for state values, only we now talk about state-action pairs being visited, i.e. taking action a in state s, rather than just states being visited.

If π is deterministic, then we will only estimate the values of actions that π dictates. We therefore need to incorporate some exploration in order to have useful action-values (since, after all, we want to use them to make informed decisions).

One consideration is to make π stochastic, e.g. ε-soft. Another is the assumption of *exploring starts*, which specifies that ever state-action pair has non-zero probability of being selected as the starting state. Of course, this is not always posible in practice.

For now we assume exploring start. Later we will come back to the issue of *maintaining exploration*

5.3 Monte Carlo Control

We make use of the GPI framework for action-values. Policy evaluation is done as described. Policy improvement is done by making the policy greedy with respect to the action-value function, so no model is needed for this step

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

We generate a sequence of policies π_k each greedy with respect to $q_{\pi_{k-1}}(s, a)$. The policy improvement theorem applies: for all $s \in \mathcal{S}$

$$\begin{aligned} q_{\pi_k}(s, a = \pi_{k+1}(s)) &= q_{\pi_k}(s, \operatorname*{argmax}_a q_\pi(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &= v_{\pi_k}(s) \end{aligned}$$

So π_{k+1} uniformly better than π_k or it is optimal.

The above procedure's convergence depends on assumptions of exploring starts and infinitely many episodes. We will relax the first later, but we will address the second now.

Two approaches to avoid infinitely many episodes:

1. Stop the algorithm once the q_{π_k} stop moving within a certain error. (In practice this is only useful on the smallest problems.)

2. Stop policy evaluation after a certain number of episodes, moving the action value towards q_{π_k}, then go to policy improvement.

For MC policy evaluation, it is natural to alternate policy evaluation and improvement on a episode by episode basis. We give such an algorithm below (with the assumption of exploring starts).

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 $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

It is easy to see that optimal policies are a fixed point of this algorithm. Whether this algorithm converges in general is still, however, an open question.

5.4 Monte Carlo Control without Exploring Starts

On Policy vs. Off Policy

On-policy methods evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve one that is different than the one used to generate the data.

On-Policy Techniques without Exploring Starts

We consider ε-greedy policies that put probability $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$ on the maximal action and $\frac{\varepsilon}{|\mathcal{A}(s)|}$ on each of the others. These are examples of ε-soft policies in which $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$.

We use this idea in the GPI framework:

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

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

We now show that an ε-greedy policy with respect to q_π, π', is an improvement over any ε-soft policy π. For any $s \in \mathcal{S}$

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) && (33) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) && (34) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) && (35) \\ &= \sum_a \pi(a|s) q_\pi(s,a) && (36) \\ &= v_\pi(s) && (37) \end{aligned}$$

(where line 3 follows because a weighted average with weights $w_i \geq 0$ and $\sum_i w_i = 1$ is $\leq$ the max term).

This satisfies the condition of the policy improvement theorem so we now know that $\pi' \geq \pi$.

Previously, with deterministic greedy policies, we would get automatically that fixed points of policy iteration are optimal policies since

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}.$$

Now our policies are not deterministically greedy, our value updates do not take this form. We note, however, that we can consider an equivalent problem where we change the environment to select state and reward transitions at random with probability ε and do what our agent asks with probability $1 - \varepsilon$. We have moved the stochasticity of the policy into the environment, creating an equivalent

problem. The optimal value function in the new problem satisfies its Bellman equation

$$\tilde{v}_\pi(s) = (1-\varepsilon)\max_a \tilde{q}_\pi(s,a) + \frac{\varepsilon}{|\mathcal{A}(s)|}\sum_a \tilde{q}_\pi(s,a) \tag{38}$$

$$= (1-\varepsilon)\max_a \sum_{s',r} p(s',r|s,a)[r+\gamma\tilde{v}_\pi(s')] + \frac{\varepsilon}{|\mathcal{A}(s)|}\sum_a\sum_{s',r} p(s',r|s,a)[r+\gamma\tilde{v}_\pi(s')]. \tag{39}$$

We also know that at fixed points of our algorithm

$$v_\pi(s) = (1-\varepsilon)\max_a q_\pi(s,a) + \frac{\varepsilon}{|\mathcal{A}(s)|}\sum_a q_\pi(s,a) \tag{40}$$

$$= (1-\varepsilon)\max_a \sum_{s',r} p(s',r|s,a)[r+\gamma v_\pi(s')] + \frac{\varepsilon}{|\mathcal{A}(s)|}\sum_a\sum_{s',r} p(s',r|s,a)[r+\gamma v_\pi(s')]. \tag{41}$$

This is the same equation as above, so by uniqueness of solutions to the Bellman equation we have that $v_\pi = \tilde{v}_\pi$ and so π is optimal.

5.5 Off-Policy Prediction via Importance Sampling

Off-policy learning uses information gained by sampling the *behaviour policy* b to learn the *target policy* π. The behaviour policy explores the environment for us during training and we update the target policy accordingly.

In this section we consider the prediction problem: estimating v_π or q_π for a fixed and known π using returns from b. In order to do this we need the assumption of coverage:

$$\pi(a|s) \geq 0 \implies b(a|s) \geq 0. \tag{42}$$

This implies that b must be stochastic wherever it is not identical to π. The target policy π may itself be deterministic, e.g. greedy with respect to action-value estimates.

Importance Sampling

We use *importance sampling* to evaluate expected returns from π given returns from b.

Define the importance sampling ratio as the relative probability of a certain trajectory from S_t

$$\rho_{t:T-1} = \frac{\mathbb{P}(A_t, S_{t+1}, A_{t+1}, \dots)|S_t, A_{t:T-1}\sim\pi}{\mathbb{P}(A_t, S_{t+1}, A_{t+1}, \dots)|S_t, A_{t:T-1}\sim b} \tag{43}$$

$$= \frac{\prod_{k=t}^{T-1}\pi(A_k|S_k)\mathbb{P}(S_{k+1}|S_k,A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)\mathbb{P}(S_{k+1}|S_k,A_k)} \tag{44}$$

$$= \prod_{k=t}^{T-1}\frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{45}$$

where the state transition dynamics $\mathbb{P}$ cancel out.

If we have returns G_t from evaluating policy b, so $v_b(s) = \mathbb{E}[G_t|S_t = s]$, then we can calculate

$$v_\pi(s) = \mathbb{E}[\rho_{t:T-1}G_t|S_t = s]$$

Estimation

Introduce new notation:

- Label all time steps in a single scheme. So maybe episode 1 is $t = 1, \ldots, 100$ and episode 2 is $t = 101, \ldots, 200$, etc.
- Denote the set times of first/every visit to s by $\mathcal{T}(s)$ (spanning episodes).
- Let $T(t)$ be the first termination after t
- Let G_t be the returns from t to $T(t)$

We can now give two methods of values for π from returns from b:
Ordinary Importance Sampling

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1} G_t}{|\mathcal{T}(s)|} \tag{46}$$

Weighted Importance Sampling

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1}} \tag{47}$$

or 0 if the denominator is 0.

Weighted importance sampling is biased (e.g. it's expectation is $v_b(s)$ after 1 episode) but has bounded variance. The ordinary importance sampling ratio is unbiased, but has possibly infinite variance, because the variance of the importance sampling ratios themselves is unbounded.

Assuming bounded returns, the variance of the weighted importance sampling estimator converges to 0 even if the variance of the importance sampling ratios is infinite. In practice, this estimator usually has dramatically lower variance and is strongly preferred.

5.6 Incremental Implementation

We look for incremental calculations of the averages that make up the estimates, as in Chapter 2.

For on-policy methods the incremental averaging is the same as in Chapter 2. For off-policy methods, but with ordinary importance sampling, we only need to multiply the returns by the importance sampling ratio and then we can average as before.

We will now consider weighted importance sampling. We have a sequence of returns G_i, all starting in the same state s and each with a random weight W_i (e.g. $W_i = \rho_{i:T(i)-1}$). We want to iteratively calculate (for $n \geq 2$)

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}.$$

We can do this with the following update rules

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n] \tag{48}$$

$$C_{n+1} = C_n + W_{n+1} \tag{49}$$

where $C_0 = 0$ and V_1 is arbitrary (notice that it cancels out as $V_2 = G_1$).

Below is an algorithm for off-policy weighted importance sampling (set $b = \pi$ for on policy). The estimator Q converges to q_π for all encountered state-action pairs.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

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)
    $C(s,a) \leftarrow 0$

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

5.7 Off-Policy Monte Carlo Control

Below is an algorithm for estimating π_* and q_* in the GPI framework. The target policy π is the greedy policy with respect to Q, which is an estimate of q_π. This algorithm converges to q_π as long as an infinite number of returns are observed for each state-action pair. This can be achieved by making b ε-soft. The policy π converges to π_* at all encountered states even if b changes (to another ε-soft policy) between or within episodes.

Off-policy MC control, for estimating $\pi \approx \pi_*$

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

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Notice that this policy only learns from episodes in which b selects only greedy actions after some timestep. This can greatly slow learning.

5.8 *Discounting Aware Importance Sampling

We present a method of importance sampling that recognises the return as a discounted sum of rewards. This can help in estimation, since if an episode is of length 100 and $\gamma = 0$ then the final 99 terms of the importance sampling ration contribute nothing to the expected value of our estimator (they have expected value of 1) but can greatly increase its variance. We therefore construct a method of importance sampling that takes into account discounting.

Introduce the *flat partial returns*

$$\bar{G}_{t:h} \doteq \sum_{i=t+1}^{h} R_i \quad 0 \leq t \leq h \leq T$$

then it can be shown (by rearranging) that

$$G_t \doteq \gamma^{i-t} R_{i+1} \tag{50}$$

$$= (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}. \tag{51}$$

Now we can scale each flat partial return by a truncated importance sampling ratio (hence reducing variance).

Ordinary Importance Sampling Ratio

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left[(1-\gamma) \sum_{h=t+1}^{T(t-1)} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right]}{|\mathcal{T}(s)|} \tag{52}$$

Weighted Importance Sampling Ratio

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)}\left[(1-\gamma)\sum_{h=t+1}^{T(t)-1}\gamma^{h-t-1}\rho_{t:h-1}\bar{G}_{t:h}+\gamma^{T(t)-t-1}\rho_{t:T(t)-1}\bar{G}_{t:T(t)}\right]}{\sum_{t\in\mathcal{T}(s)}\left[(1-\gamma)\sum_{h=t+1}^{T(t)-1}\gamma^{h-t-1}\rho_{t:h-1}+\gamma^{T(t)-t-1}\rho_{t:T(t)-1}\right]} \tag{53}$$

5.9 *Per-Decision Importance Sampling

There is another way in which we may be able to reduce variance in off-policy importance sapling, even in the absence of discounting ($\gamma = 1$). Notice that the off-policy estimators are made up of terms like

$$\rho_{t:T-1}G_t = \rho_{t:T-1}(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1}R_T)$$

and that each of these terms is of the form

$$\rho_{t:T-1}R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}\cdots\frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})}R_{t+1}.$$

Now notice that only the first and last terms here are correlated, while all the others have expected value 1 (taken with respect to b). Clearly this is also the case at each t. This means that

$$\mathbb{E}[\rho_{t:T-1}R_{t+k}] = \mathbb{E}[\rho_{t:t+k-1}R_{t+k}]$$

therefore

$$\mathbb{E}[\rho t : T-1 G_t] = \mathbb{E}[\tilde{G}_t]$$

where

$$\tilde{G}_t \doteq \sum_{i=t}^{T-1}\gamma^{i-t}\rho_{t:i}R_{i+1}.$$

Now we can write the ordinary importance sampling estimator as

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)}\tilde{G}_t}{|\mathcal{T}(s)|}$$

possibly reducing variance in the estimator.

The weighted importance sampling estimators of this form that have so far been found have been shown to not be consistent (in the statistical sense). We don't know if a consistent weighted average form of this exists.

6 Temporal-Difference Learning

We first focus on the prediction problem, that is, finding v_pi given a π. The control problem, finding π_*, is approached using the GPI framework.

6.1 TD Prediction

Connection between TD, MC & DP

Monte-Carlo methods wait until the end of an episode to update the values. A simple MC update suitable for non-stationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{54}$$

we will call this *constant-α MC*. Temporal difference learning (TD) increments the values at each timestep. The following is the TD(0) (or one-step TD) update which is made at $t+1$ (we will see TD(λ) in Chapter 12)

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{55}$$

The key difference is that MC uses G_t as the target whereas TD(0) uses $R_{t+1} + \gamma V(S_{t+1})$. TD uses an estimate in forming the target, hence is known as a *bootstrapping method*. Below is TD(0) in procedural form.

Tabular TD(0) for estimating v_π

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

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'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

The core of the similarity between MC and TD is down to the following relationship

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] && (56)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] && (57)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1}) | S_t = s] && (58) \end{aligned}$$

- MC uses an estimate of the first line, since it uses sample returns to approximate the expectation
- DP uses an estimate of the final line, because it approximates v_π by V
- TD does both, it samples the returns like MC and also uses the current value estimates in the target

TD Error

We can think of the TD(0) update as an error, measuring the difference between the estimated value for S_t and the better estimate of $R_{t+1} + \gamma V(S_{t+1})$. We define the *TD error*

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t), \tag{59}$$

now if the array V does not change within the episode we can show (by simple recursion) that the MC error can be written

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma_{k-t} \delta_k. \tag{60}$$

6.2 Advantages of TD Prediction Methods

- TD methods do not require a model of the environment
- TD methods are implements online, which can speed convergence vs. MC methods which must wait until the end of (potentially very long) episodes before learning. TD methods can be applied to continuing tasks for the same reason
- TD methods learn from all actions, whereas MC methods required that the tails of the episodes be greedy
- For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions
- It is an open question as to whether TD methods converge faster than constant-α MC methods in general, though this seems to be the case in practice

6.3 Optimality of TD(0)

Given a finite number of training steps or episodes, a common method for estimating V is to present the experience repeatedly until V converges. We call the following *batch updating*: given finite experience following a policy and an approximate value function V, calculate the increments for each t that is non-terminal and change V once by the sum of all the increments. Repeat until V converges.

Under batch updating, we can make some comments on the strengths of TD(0) relative to MC. In an online setting we can do no better than to guess that online TD is faster than constant-α MC because it is similar towards the batch updating solution.

- Under batch updating, MC methods always find estimates that minimize the mean-squared error on the training set.
- Under batch updating, TD methods always finds the estimate that would be exactly correct for the maximum-likelihood model of the Markov process. The MLE model is the one in which the estimates for the transition probabilities are the fraction of observed occurrences of each transition.
- We call the value function calculated from the MLE model the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process is exact. In general, batch TD(0) converges to the certainty equivalence estimate.

6.4 Sarsa: On-policy TD Control

We now use TD methods to attack the control problem. The *Sara* update is as follows

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{61}$$

This update is done after every transition from a non-terminal state S_t. If S_{t+1} is terminal then we set $Q(S_{t+1}, A_{t+1}) = 0$. Note that this rule uses the following elements $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which gives rise to the name Sarsa. The theoroms regarding convergene of the state-value versions of this update apply here too.

We write an on-policy control algorithm using Sarsa in the box below, at each time step we move the policy towards the greedy policy with respect to the current action-value function. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state action pairs are visited infinitely often and the policy also converges to the greedy policy in the limit (e.g. maybe π is ε-greedy with $\varepsilon = \frac{1}{t}$).

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

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

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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

6.5 Q-learning: Off-policy TD Control

The *Q-learning* update is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{62}$$

The learning function Q directly approximates q_*. All that is required for convergence is that all pairs continue to be updated. An algorithm for Q-learning is given in the box below.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

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

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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

6.6 Expected Sarsa

The update rule for *Expected Sarsa* is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \tag{63}$$

$$\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{64}$$

This algorithm moves deterministically in the same direction as Sarsa moves in *expectation*, hence the name. It is more computationally complex than Sarsa, but eliminates the variance due to random selection of A_{t+1}. Given the same amount of experiences, it generally performs slightly better than Sarsa.

6.7 Maximisation Bias and Double Learning

All the control algorithms we have discussed so far involve some sort of maximisation in the construction of their target policies. This introduces a positive bias to the value estimates because they form uncertain estimates of the true values. This is known as the *maximisation bias*. It is essentially down to the fact the the expectation of the max of a sample is $\geq$ the max of the expected values of the samples.

To solve this we introduce the idea of *double learning*, in which we learn two independent sets of value estimates Q_1 and Q_2, then at each time step we choose one of them at random and update it using the other as a target. This produces two unbiased estimates of the action-values (which could be averaged). Below we show an algorithm for *double Q-learning*.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

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

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

6.8 Games, Afterstates, and other Special Cases

In this book we try to present a uniform approach to solving tasks, but sometimes more specific methods can do much better.

We introduce the idea of *afterstates*. Afterstates are relevant when the agent can deterministically change some aspect of the environment. In these cases, we are better to value the resulting state of the environment, after the agent has taken action and before any stochasticity, as this can reduce computation and speed convergence.

Take chess as an example. One should choose as states the board positions *after* the agent has taken a move, rather than before. This is because there are multiple states at t than can lead to the board position that the opponent sees at $t + 1$ (assuming we move second) via deterministic actions of the agent.

7 n-step Bootstrapping

n-step methods allow us to observe multiple time-steps of returns before updating a state with the observed data and a bootstrapped estimate of the value of the nth succeeding state.

7.1 n-step TD Prediction

Define the n-step return

$$G_{t:t+n} \doteq \sum_{i=t}^{t+n-1} \gamma^{i-t} R_{i+1} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{65}$$

where $n \geq 1$, $0 \leq t < T - n$ and V_i is the estimated state-value function as of time i. If $t + n > T$ then $G_{t+n} \equiv G_t$, the standard return. The n-step return is the target for *n-step TD* methods, note that $n - 1$ rewards are observed and the succeeding value is bootstrapped with the latest estimate of the value function. The corresponding update for state-values is

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)] \qquad 0 \leq t < T. \tag{66}$$

Note that Monte-Carlo can be thought of as TD(∞)Pseudocode for n-step TD is given in the box below.

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 $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | 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 $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

The n-step return obeys the *error-reduction property*, and because of this n-step TD can be shown to converge to correct predictions (given a policy) under appropriate technical conditions. This property states that the n-step return is a better estimate than V_{t+n-1} in the sense that the error on the worst prediction is always smaller

$$\max_s |\mathbb{E}_\pi[G_{t:t+n}|S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{67}$$

7.2 n-step Sarsa

Sarsa

We develop n-step methods for control. We generalise Sarsa to n-step Sarsa, or Sarsa(n). This is done in much the same way as above, but with action-values as opposed to state-values. The n-step return in this case is defined as

$$G_{t:t+n} \doteq \sum_{i=t}^{t+n-1} \gamma^{i-t} R_{i+1} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \tag{68}$$

where $n \geq 1$, $0 \leq t < T - n$ and Q_i is the estimated action-value function as of time i. If $t + n > T$ then $G_{t+n} \equiv G_t$, the standard return. The corresponding update is

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \qquad 0 \leq t < T. \tag{69}$$

Expected Sarsa

We define n-step expected Sarsa similarly

$$G_{t:t+n} \doteq \sum_{i=t}^{t+n-1} \gamma^{i-t} R_{i+1} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}) \tag{70}$$

where $n \geq 1$, $0 \leq t < T - n$ and $\bar{V}_i$ is the *expected approximate value* of state s

$$\bar{V}_i(s) \doteq \sum_a \pi(a|s) Q_i(s, a). \tag{71}$$

As always, if $t + n > T$ then $G_{t+n} \equiv G_t$, the standard return. The corresponding update is formally the same as above

n-step Sarsa 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 ε-greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

7.3 n-step Off-policy Learning

We can learn with n-step methods off-policy using the importance sampling ratio (target policy π and behaviour policy b)

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

For state-values we have

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)]$$

and for action-values we have

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

note that for action values the importance sampling ratio starts one time-step later, because we are attempting to discriminate between actions at time t.

Off-policy n-step Sarsa for 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 as 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 $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    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 $A_{t+1} \sim b(\cdot|S_{t+1})$
    $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:t+n-1})$
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau+n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

7.4 *Per-decision Methods with Control Variates

We have the standard recursion relation for the n-step return

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}.$$

For an off-policy algorithm, one would be tempted to simply weight this target by the importance sampling ratio. This method, however, shrinks the estimated value functions when the importance sampling ratio is 0, hence increasing variance. We thus introduce the *control-variate* $(1-\rho_t)V_{h-1}(S_t)$, giving an off-policy update of

$$G_{t:h} = \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1-\rho_t)V_{h-1}(S_t)$$

where $G_{h:h} = V_{h-1}(S_h)$. Note that the control-variate has expected value 0, since the factors are uncorrelated and the expected value of the importance sampling ratio is 1.

We can do a similar thing for action-values

$$G_{t:h} \doteq R_{t+1} + \gamma\rho_{t+1:h}\left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\right) - \gamma\bar{V}_{h-1}(S_{t+1}),$$

where once again the importance sampling ratio starts one time-step later.

Control Variates in General

Suppose we want to estimate μ and assume we have an unbiased estimator for μ in m. Suppose we calculate another statistic t such that $\mathbb{E}[t] = \tau$ is a known value. Then

$$m^\star = m + c\,(t - \tau)$$

is also an unbiased estimator for μ for any c, with variance

$$\mathrm{Var}\,(m^\star) = \mathrm{Var}\,(m) + c^2\,\mathrm{Var}\,(t) + 2c\,\mathrm{Cov}\,(m, t)\,.$$

It is easy to see that taking

$$c = -\frac{\mathrm{Cov}\,(m, t)}{\mathrm{Var}\,(t)}$$

minimizes the variance of $m^\star$. With this choice

$$\mathrm{Var}(m^\star) = \mathrm{Var}(m) - \frac{[\mathrm{Cov}(m, t)]^2}{\mathrm{Var}(t)} \tag{72}$$

$$= (1 - \rho_{m,t}^2)\mathrm{Var}(m) \tag{73}$$

where $\rho_{m,t} = \mathrm{Corr}\,(m, t)$ is the Pearson correlation coefficient of m and t. The greater the value of $|\rho_{m,t}|$, the greater the variance reduction achieved.

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

We introduce the n-step *tree-backup algorithm* algorithm using the return

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \tag{74}$$

for $t < T - 1$, $n > 1$ and with $G_{i:i} = 0$ and $G_{T-1:t+n} = R_T$. This algorithm updates S_t with bootstrapped, probability weighted action-values of *all* actions that were not taken all along the trajectory and recursively includes the rewards realised, weighted by the probability of their preceding actions under the policy. Pseudocode given below.

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 as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t+1$
      else:
        Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
    $\tau \leftarrow t+1-n$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      If $t+1 \geq T$:
        $G \leftarrow R_T$
      else
        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
      Loop for $k = \min(t, T-1)$ down through $\tau + 1$:
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

7.6 *A Unifying Algorithm: n-step $Q(\sigma)$

We introduce an algorithm which, at each time step, can choose to either take an action as a sample as in Sarsa or to take an expectation over all possible actions as in tree-backup.

Define a sequence $\sigma_t \in [0,1]$ that at each time step chooses a proportion of sampling vs. expectation. This generalises Sarsa and tree-backup by allowing each update to be a linear combination of the two ideas. The corresponding return (off-policy) is

$$G_{t:h} \doteq R_{t+1} + \gamma \left(\sigma_{t+1}\rho_{t+1}(1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1})\right)\left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\right) \tag{75}$$

$$+ \gamma \bar{V}_{h-1}(S_{t+1}), \tag{76}$$

for $t < h < T$, with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$ if $h < T$ and $G_{T-1:T} \doteq R_t$ if $h = T$. Pseudocode given below.

Off-policy n-step $Q(\sigma)$ for 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 ε-greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$:
  | Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
  | If S_{t+1} is terminal:
  | $T \leftarrow t+1$
  | else:
  | Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
  | Select and store σ_{t+1}
  | Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow 0$:
  | Loop for $k = \min(t+1, T)$ down through $\tau + 1$:
  | if $k = T$:
  | $G \leftarrow R_T$
  | else:
  | $\bar{V} \leftarrow \sum_a \pi(a|S_k) Q(S_k, a)$
  | $G \leftarrow R_k + \gamma\big(\sigma_k \rho_k + (1-\sigma_k)\pi(A_k|S_k)\big)\big(G - Q(S_k, A_k)\big) + \gamma \bar{V}$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

8 Planning and Learning with Tabular Methods

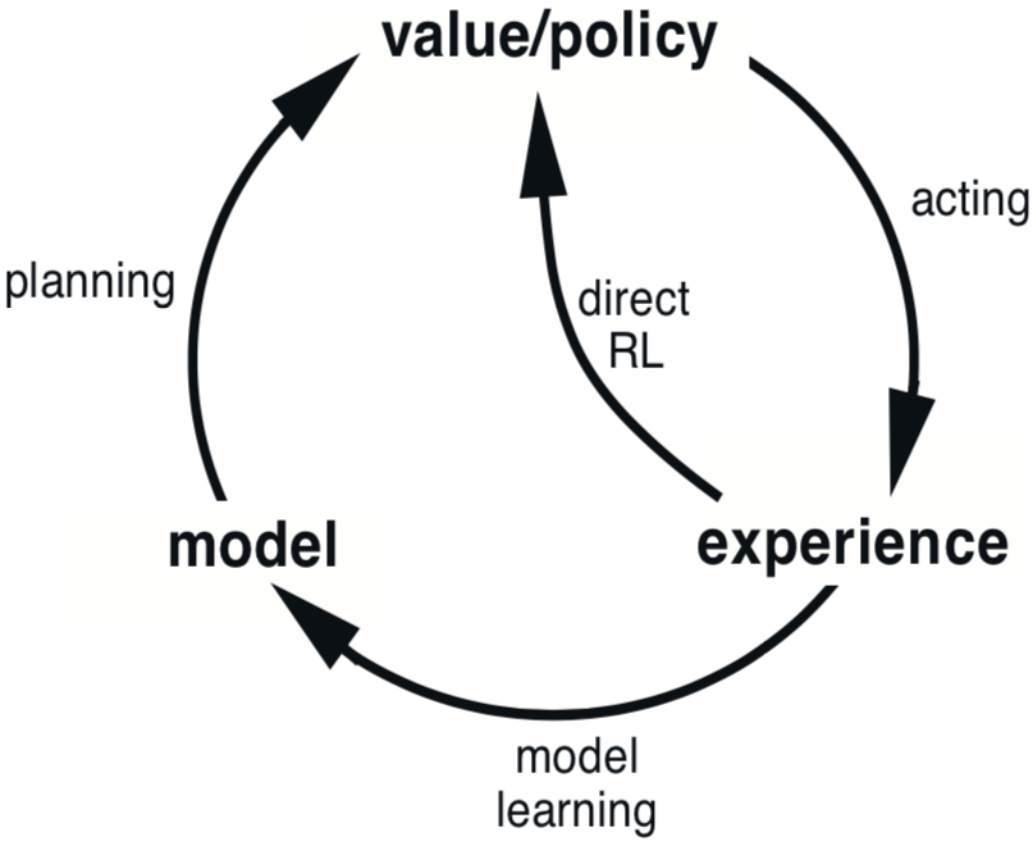
8.1 Models and Planning

A *model* of the environment is anything that an agent can use to predict how the environment will respond to its actions. A *distribution model* is one that characterises the distribution of possible environmental changes, whereas a *sample model* is one that produces sample behaviour. Distribution models are in some sense stronger, in that they can be used to produce samples of the behaviour of the environment, but it is often easier to reproduce sample responses than to model the response distribution.

Models can be used to simulate the environment and hence simulate experience. We use the term *planning* to refer to a computational process that uses a model for improving a policy. The kind of planning that we consider here falls under the name *state-space planning*, since it is a search through the state space for an optimal policy or path to a goal. (Planning as we consider it here is essentially just learning from simulated experience.)

8.2 Dyna: Integrated Planning, Acting and Learning

Within a planning agent, real experience can be used to improve the model or to directly improve the value function and policy. The former we call *model learning* and the latter we call *direct reinforcement learning*. The use of a learned model to improve the value function and policy is sometimes called *indirect reinforcement learning*. The figure below illustrates this duality.



Dyna-Q

Dyna-Q uses one-step tabular Q-learning to learn from both real and simulated experience. (It is typical to use the same update rule for both types of experience.) The idea is that a model and a value function are learned simultaneously from real experience, and the model is then used for further planning. An algorithm is given below. Note that, although not shown in this way, the planning and direct learning can run concurrently.

Tabular Dyna-Q

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

8.3 When the Model is Wrong

Of course, the model we are learning may be incorrect, meaning that planning results in a sub-optimal policy. If the environment's dynamics are non-stationary, then this will be an issue for the agent.

In some cases, the suboptimal policy results in discovery and correction of model error, since if the model leads to optimistic estimates for action-values the agent will take these actions and realise its modelling error. The situation can be more difficult when values are underestimated, since in this case the agent may never choose to have experience that would correct its model.

Dyna-Q+

The aforementioned issue of model error, especially in non-stationary environments, is the general problem of exploration versus exploitation. There is probably no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent keeps track of the time elapsed since it last visited each state-action pair, then increases the reward from visiting these pairs in *simulated* experience to $r + \kappa\sqrt{\tau}$, where r is the modeled reward for the transition, τ is the number of time-steps since the last time the state-action pair was visited and κ is a small constant. This increases computational complexity, but has the benefit of encouraging the agent to try actions that it hasn't taken in a long time.

8.4 Prioritised Sweeping

In the Dyna-Q algorithm given above, planning is done using uniform sweeps of the state-action space. This could be very wasteful, for instance because it is possible that there are many parts of

the state-action space that are irrelevant to the optimal policies. It is also the case that uniformly distributed planning updates could waste effort on states whose value functions have not changed recently, which is wasted computation.

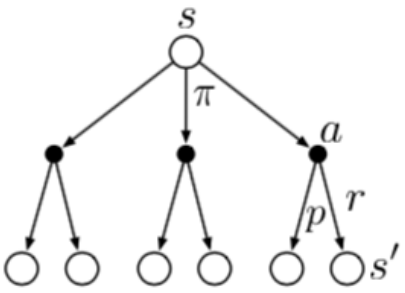
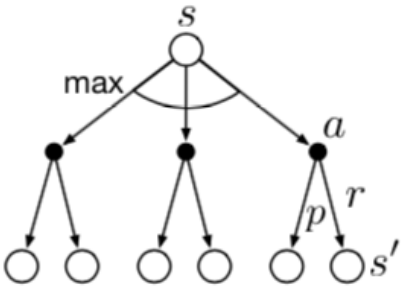
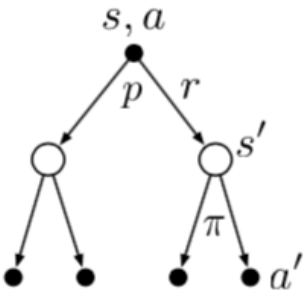
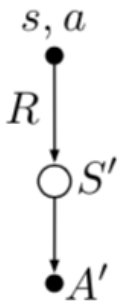
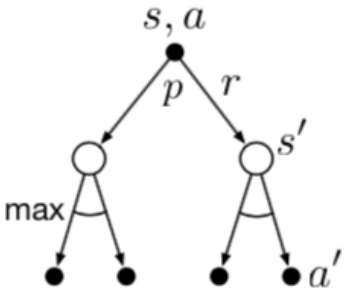
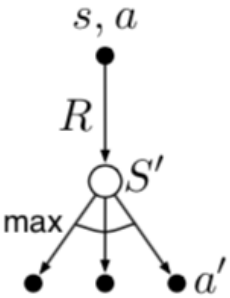
Prioritised sweeping focuses updates on the state-action pairs whose estimated values are likely to change the most from the most recent experience. Q queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritised by the size of the change. During planning, the state-action pair that is first in the queue is updated and removed from the queue first, then it's predecessors are updated and removed (if the update would be significant), and so on. An algorithm for deterministic environments is given below.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
- (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) Loop repeat n times, while $PQueue$ is not empty:
  - $S,A \leftarrow first(PQueue)$
  - $R,S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  - Loop for all $\bar{S},\bar{A}$ predicted to lead to S:
    - $\bar{R} \leftarrow$ 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

8.5 Expected vs. Sample Updates

This section is about the relative benefits of expected and sample updates. Expected updates consider all possible outcomes, while sample updates use only sample experience of particular outcomes. In the absence of a distribution model, expected updates are not possible. A summary of all one-step updates considered are given below.

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

Since expected updates do not directly suffer from sampling error (error could propagate through model estimation in planning), they are more computationally intensive. However, they are not always optimal. In problems with large state-spaces or branching factors, sample updates are often much more efficient. This means that one can do many sample updates in the same computational time as an expected update, in turn meaning that the sample updates produce more accurate value estimates in the given time.

8.6 Trajectory Sampling

As discussed previously, distributing updates uniformly during planning is often sub-optimal. This is because for many tasks, the majority of possible updates will be on irrelevant or low-probability trajectories.

We could generate experience and updates in planning by interacting the current policy with the model, then only updating the simulated trajectories. We call this *trajectory sampling*. Naturally, trajectory sampling generates updates according to the on-policy distribution.

Focusing on the on-policy distribution could be beneficial because it causes uninteresting parts of the space to be ignored, but it could be detrimental because it causes the same parts of the space to be updated repeatedly. It is often the case the distributing updates according to the on-policy distribution is preferable to using the uniform distribution for larger problems.

8.7 Real-time Dynamic Programming

Real-time dynamic programming (RTDP) is a on-policy, trajectory-sampling version of value-iteration DP. This is DP value iteration, but with the updates distributed according to the on-policy distribution. As such, it is a form of asynchronous DP.

Due to the trajectory sampling, RTDP allows us to skip portions of the state space that are not relevant to the current policy (in terms of the prediction problem). For the control problem (finding an optimal policy) all we really need is an *optimal partial policy*, which is a policy that is optimal on the relevant states and specifies arbitrary actions on the others.

In general, finding an optimal policy with on-policy trajectory-sampling control method (e.g. Sarsa) requires visiting all state action pairs infinitely many times in the limit. This is true for RTDP as well, but there are certain types of problems for which RTDP is guaranteed to find ann optimal partial policy without visiting all states infinitely often. This is an advantage for problems with very large state sets.

The particular tasks for which this is the case are *stochastic optimal path problems* (which are generally framed in terms of cost minimisation rather than reward maximisation). They are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards. For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided: 1) the initial value of every goal state is zero, 2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, 3) all rewards for transitions from non-goal states are strictly negative, and 4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to zero).

8.8 Planning at Decision Time

The type of planning we have considered so far is the improvement of a policy or value function based on simulated experience. This is not focussed on interaction with the environment and is called *background planning*.

An alternative type of planning, *decision time planning*, is the search (sometimes many actions deep) of possible future trajectories given the current state.

8.9 Heuristic Search

In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. Rollout algorithms start in a given state, then estimate the value of the state by averaging simulated returns from that state after following a given policy, called the *rollout policy*. The action with the highest estimated value is selected and the process is repeated. This is useful when one knows a policy but needs to average over some stochasticity in the environment.

8.11 Monte Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a successful example of decision time planning. It is a rollout algorithm that accumulates value estimates from the Monte Carlo simulations in order to guide the search. A variant of MCTS was used in AlphaGo.

A basic version of MCTS follows the following steps, starting at the current state:

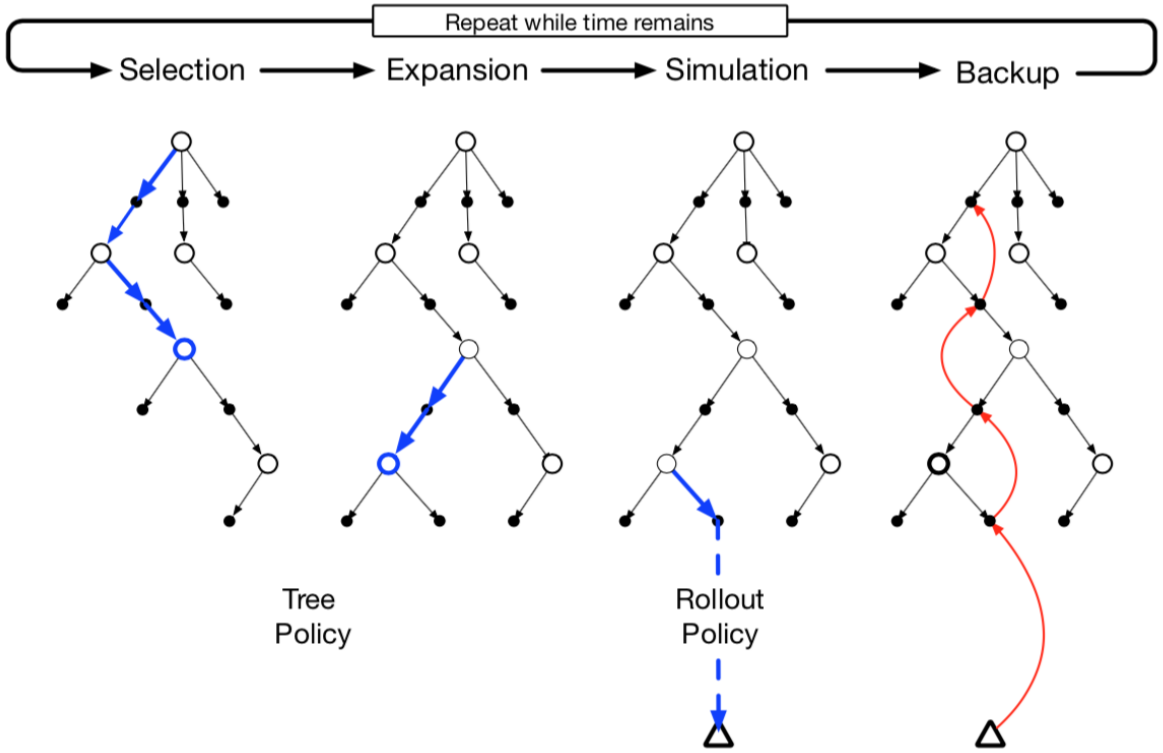
1. **Selection.** Starting at the root node, a *tree policy* based on action-values attached to the edges of the tree (that balances exploration and exploitation) traverses the tree to select a leaf node.

2. **Expansion.** On some iterations (depending on the implementation), the tree is expanded from the selected leaf node by adding one of more child nodes reached from the selected node via unexplored actions.

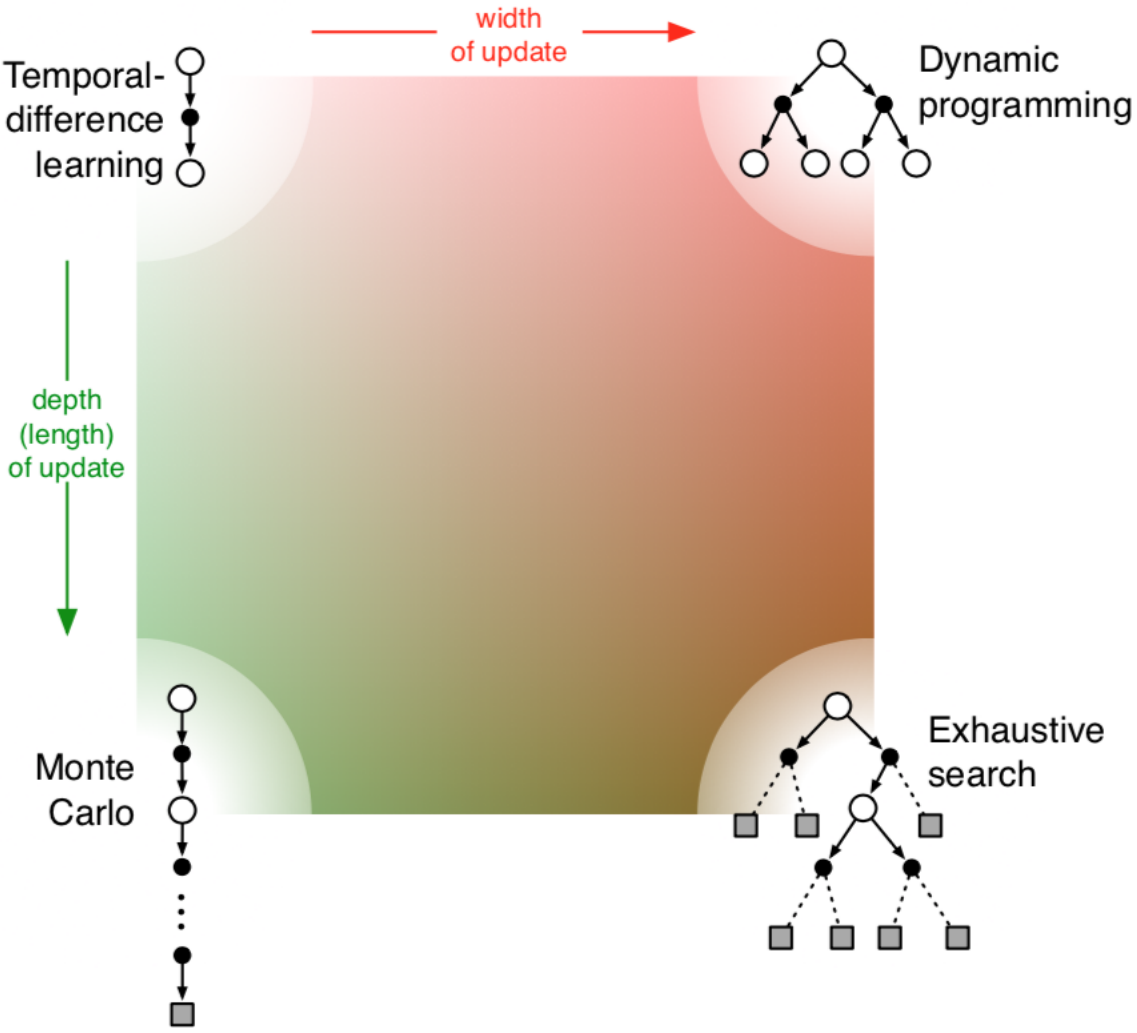
3. **Simulation.** From the selected node, or from one if its newly added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialise, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree.

The figure below illustrates this process. MCTS executes this process iteratively, starting at the current state, until no more time is left or computational resources are exhausted. An action is then taken based on some statistics in the tree (e.g. largest action-value or most visited node). After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.



Summary of Part I



9 On-policy Prediction with Approximation

In this section we consider the applications of function approximation techniques in reinforcement learning, to learn mappings from states to values. Typically we will consider parametric functional forms, in which case we can achieve a reduction in dimensionality of the problem (number of parameters smaller than state space). In this way, the function generalises between states, as the update of one state impacts the value of another.

Function approximation techniques are applicable to partially observable problems, in which the full state space is not available to the agent. A function approximation scheme which is ignores certain aspects of the space behaves just as if those aspects are unobservable.

9.1 Value-function Approximation

Many techniques from supervised learning are applicable to learning value functions from experience, but not all are equipped to deal with the non-stationarity that often occurs in RL. In RL it is also important to be able to learn online.

9.2 The Prediction Objective ($\overline{\mathrm{VE}}$)

Define a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$ that represents how much we care about each state s. Given an estimator $\hat{v}(s, \boldsymbol{w})$ of $v_\pi(s)$, parameterised by $\boldsymbol{w}$, we define our objective function as the *Mean Squared Value Error*

$$\overline{\mathrm{VE}} \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{w}) \right]^2 . \tag{77}$$

Often we choose $\mu(s)$ to be the fraction of time spent in s. Under on-policy training this is referred to as the *on-policy distribution*. In continuing tasks, this is the stationary distribution under π.

At this stage it is not clear that we have chosen the correct (or even a good) objective function, since the ultimate goal is a good policy for the task. For now, will continue with $\overline{\mathrm{VE}}$ nonetheless.

The on-policy distribution in episodic tasks

In an episodic task the on-policy distribution depends on how the initial states of the episode are chosen. Let $h(s)$ be the probability that an episode begins in state s and $\eta(s)$ be the expected time spent in s per episode. Note that you can either start in s or transition there from $\bar{s}$, so

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a) \quad \forall s \in \mathcal{S}.$$

One can solve this system for η, then take the on-policy distribution as

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \quad \forall s \in \mathcal{S}.$$

This is the natural choice without discounting. With discounting we consider it a form of termination and include a factor of γ in the second term of the recurrence relation above.

9.3 Stochastic-gradient and Semi-gradient Methods

(Stochastic) Gradient Descent

We assume that states appear in examples with the same distribution $\mu(s)$, in which case a good strategy is to minimise our loss function on observed examples. *Stochastic gradient-descent* moves the weights in the direction of decreasing $\overline{\text{VE}}$:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{2}\alpha\nabla_{\boldsymbol{w}}\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w})\right]^2 \tag{78}$$

$$= \boldsymbol{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w})\right]\nabla_{\boldsymbol{w}}\hat{v}(S_t, \boldsymbol{w}). \tag{79}$$

Of course, we might not know the true value function exactly, we will likely only have access to some approximation of it U_t, possibly corrupted by noise or got from bootstrapping with our latest estimate. In these cases we cannot perform the above computation, but we can still make the general SGD update

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\left[U_t - \hat{v}(S_t, \boldsymbol{w})\right]\nabla_{\boldsymbol{w}}\hat{v}(S_t, \boldsymbol{w}) \tag{80}$$

If U_t is an unbiased estimate of the state value for each t, then the sequence $\boldsymbol{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α.

The Monte Carlo target $U_t = G_t$ is an unbiased estimator, so locally optimal convergence is guaranteed in this case. Algorithm is given below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Semi-Gradient Descent

We don't get the same convergence guarantees if we use bootstrapping estimates of the value function in our update target, for instance if we had used the TD(0) update $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w})$. This is because the target now depends on the parameters $\boldsymbol{w}$, so the gradient is not exactly the gradient of our loss function – it only takes into account the change on our estimate with respect to $\boldsymbol{w}$. For this reason we call updates such as this *semi-gradient methods*.

Semi-gradient methods are often preferable to pure gradient methods since they can offer much faster learning, in spite of not giving the same convergence guarantees. A prototypical choice is the TD(0) update, an algorithm for which is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

State Aggregation

State aggregation is a simple form of generalising in which we group together states and fix them to have the same estimated value.

9.4 Linear Methods

As always, linear methods of function approximation are an important special case

$$\hat{v}(s, \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{x}(s) \tag{81}$$

where $\boldsymbol{x}(s)$ are feature vectors, vectors of functions (features) $x_i : \mathcal{S} \to \mathbb{R}$. The SGD update for the linear model is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\left[U_t - \hat{v}(S_t, \boldsymbol{w})\right]\boldsymbol{x}(s). \tag{82}$$

Naturally, the linear case is the most studied and the majority of convergence results for learning systems are for this case (or simpler). In particular, there is the benefit that there is a unique global optimum for our loss function (in the non-degenerate case).

Convergence of Linear TD(0)

The semi-gradient TD(0) algorithm is known to converge under linear function approximation. The point converged to is not the global optimum, but a point near the local optimum. We consider this case in more detail. First write $\boldsymbol{x}_t = \boldsymbol{x}(S_t)$ then rearrange the update

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\left(R_{t+1} + \gamma\boldsymbol{w}_t^\top\boldsymbol{x}_{t+1} - \boldsymbol{w}_t^\top\boldsymbol{x}_t\right)\boldsymbol{x}_t \tag{83}$$

$$= \boldsymbol{w}_t + \alpha\left(R_{t+1}\boldsymbol{x}_t - \boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^\top\boldsymbol{w}_t\right). \tag{84}$$

Now note that we can write

$$\mathbb{E}[\boldsymbol{w}_{t+1}|\boldsymbol{w}_t] = \boldsymbol{w}_t + \alpha(\boldsymbol{b} - \mathrm{A}\boldsymbol{w}_t)$$

where $\boldsymbol{b} = \mathbb{E}[R_{t+1}\boldsymbol{x}_t]$ and $\mathrm{A} = \mathbb{E}\left[\boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^\top\right]$. It's clear now that in a steady state we must have (can be shown that A positive definite and so invertible)

$$\boldsymbol{w}_{\mathsf{TD}} = \mathrm{A}^{-1}\boldsymbol{b}.$$

We call this point the *TD fixed point*, linear semi-gradient TD(0) converges to this point. (In the notes there is a box with some details.)

At the TD fixed point (in the continuing case) it has been proven that $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error

$$\overline{\text{VE}}(\boldsymbol{w}_{\text{TD}}) \leq \frac{1}{1-\gamma} \min_{\boldsymbol{w}} \overline{\text{VE}}(\boldsymbol{w}). \tag{85}$$

It is often the case that γ is close to 1, so this region can be quite large. The TD method jas substantial loss in asymptotic performance. Regardless of this, it still has much lower variance than MC methods and can thus be faster. The desired update method will depend on the task at hand.

Other Linear Updates

Linear semi-gradient DP $U_t = \sum_a \pi(a|S_t) \sum_{s',r} p(s', r|S_t, a)[r + \gamma \hat{v}(s', \boldsymbol{w}_t)]$ with updates according to the on-policy distribution also converged to the TD fixed point. There are convergence results for other step methods we have considered too. Critical to all of these is that updates are taken according to the on-policy distribution. For other update distributions, bootstrapping methods can diverge to infinity. n-step semi-gradient TD is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | 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 $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau, \mathbf{w})\right] \nabla \hat{v}(S_\tau, \mathbf{w})$
  Until $\tau = T - 1$

9.5 Feature Construction for Linear Methods

Discussed in this section

- Polynomial Basis
- Fourier Basis

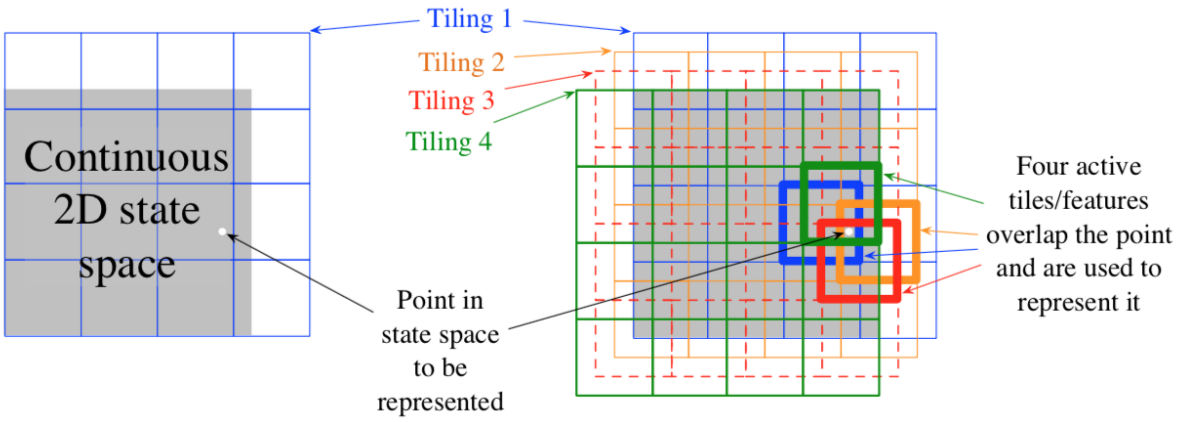
- One could use other orthogonal function bases but they are yet to see application in RL.
- Radial Basis Functions. (Offer little advantage over coarse coding with circles, but greatly increases computational complexity)

9.5.3 Coarse Coding

One way of promoting generalisation between states is to cover the state-space in overlapping regions, with each region representing a feature. If the state is being considered, then all regions that contain this state will be activated. The amount of overlap of the receptive fields (the states which can activate a feature) dictates the breadth of generalisation.

9.5.4 Tile Coding

A *tiling* of a continuous state space is form of coarse-coding that creates a partition of the state space (all of the state space is covered but elements of the tiling do not overlap). We call a sub-region of a tiling a *tile*. One might introduce multiple overlapping tilings to incorporate generalisation.



An advantage of tilings is that, because each tiling forms a partition, the total number of features active at any one time is just the total number of tilings used. So $\alpha = \frac{1}{kn}$, where n is the number of tilings, results in k-trial learning. That is, on average the learning asymptotes after k presentations of each state (assuming all updates use the same, constant target).

Tile coding is computationally efficient and may be the most practical feature representation for modern sequential digital computers.

A useful trick for reducing memory requirements is *hashing*. One can essentially hash the state space, then tile the hashed values. This means that each tile in the hashed space will represent (multiple) pseudo-randomly distributed tiles in the original space. Since only a small proportion of the state space needs to have high resolution value estimates, this can be a good way to reduce memory with little loss in performance.

9.6 Selecting Step-Size Parameters Manually

In the tabular case, taking $\alpha = \frac{1}{\tau}$ will mean that the estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, in about τ experiences.

With function approximation, there is not a clear notion of the number of visits to a state because of continuous degrees of generalisation. However, a sensible consideration for learning from τ

presentations is

$$\alpha = \frac{1}{\tau \mathbb{E}[\boldsymbol{x}^\top \boldsymbol{x}]}. \tag{86}$$

9.7 Nonlinear Function Approximation: Artificial Neural Networks

These of course see a lot of application in RL, especially with deep learning. There are some good review articles on the web.

9.8 Least-Squares TD

We saw earlier that TD(0) with linear function approximation converges to the TD fixed point

$$\boldsymbol{w}_{\mathsf{TD}} = \mathrm{A}^{-1}\boldsymbol{b},$$

where $\boldsymbol{b} = \mathbb{E}[R_{t+1}\boldsymbol{x}_t]$ and $\mathrm{A} = \mathbb{E}\left[\boldsymbol{x}_t(\boldsymbol{x}_t - \gamma \boldsymbol{x}_{t+1})^\top\right]$. Previously we computed the solution iteratively, but this is a waste of data! We could compute the MLE of A and $\boldsymbol{b}$ and then use those. This is the *Least-Squares TD Algorithm*, it uses the estimators

$$\hat{\mathrm{A}}_t = \sum_{k=0}^{t-1} x_k(x_k - \gamma x_{k+1})^\top + \epsilon \mathrm{I} \quad \text{and} \quad \hat{\boldsymbol{b}}_t = \sum_{k=0}^{t-1} R_{t+1} x_k \tag{87}$$

where we introduce $\epsilon > 0$ to ensure that the sequence of $\hat{\mathrm{A}}_t$ are each invertible. (These are estimates of $t\mathrm{A}$ and $t\boldsymbol{b}$ but the t cancel out.)

This is the most data efficient form of TD(0), but it is also more computationally intensive. Implementing incrementally and with tricks to do the matrix inverse (because of the particular form of A as sum of outer products), one can do this in $O(d^2)$ computations, where d is the number of parameters/features (note that this is independent of t). (For comparison, the semi-gradient TD(0) method needs $O(d)$ computations.) The formula for A is

$$\hat{\mathrm{A}}_t = \left(\hat{\mathrm{A}}_{t-1} + x_t(x_t - \gamma x_{t+1})^\top\right)^{-1} \tag{88}$$

$$= \hat{\mathrm{A}}_{t-1}^{-1} - \frac{\hat{\mathrm{A}}_{t-1}^{-1} x_t (x_t - \gamma x_{t+1})^\top \hat{\mathrm{A}}_{t-1}^{-1}}{1 + x_t(x_t - \gamma x_{t+1})^\top \hat{\mathrm{A}}_{t-1}^{-1} x_t} \tag{89}$$

To store $\hat{A}_{t-1}$ LSTD also needs $O(d^2)$ memory. LSTD has no step-size parameter, which means that it never forgets – this can be a blessing or a curse depending on the application. The choice between LSTD and semi-gradient TD will depend on the application, for instance the computation available and the importance of learning quickly. Pseudocode for LSTD is given below.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A d-dimensional vector
Loop for each episode:
  Initialize S; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe R, S'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma \mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top / \big(1 + \mathbf{v}^\top \mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

9.9 Memory-based Function Approximation

As an alternative to the parametric approaches discussed above, we might instead store all the training examples and execute an algorithm on the whole dataset when required, such as LOESS or nearest neighbour averaging. This approach is sometimes called *lazy learning*. The methods that go with this are non-parametric function approximation schemes. One can often evaluate the function approximation locally in the neighbourhood of the current state, which helps with the curse of dimensionality.

9.10 Kernel-based Function Approximation

Using kernels to define similarities between states for generalisation, e.g. kernel regression for state values.

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

Sometimes we are not equally interested in each state, so limited resources can be better spent than to treat every state equally. For instance, in discounted episodic problems we might be more interested in starting states because later rewards are discounted.

Introduce the scalar random variable $I_t \geq 0$ called $interest$, the degree of interest we have in accurately valuing the state at time t. If we don't cared al all about the state then $I_t = 0$, if we fully care then it might be 1 (but it is formally allowed to take any non-negative value). The interest can be set in any causal way. The distribution in our loss function $\overline{\text{VE}}$ is then defined as the distribution of states encountered when following the target policy, weighted by the interest.

We also introduce the scalar random variable $M_t \geq 0$, called the *emphasis*. The emphasis multiplies the learning update at each time-step. For general n-step learning

$$\boldsymbol{w}_{t+n} = \boldsymbol{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \boldsymbol{w}_{t+n-1})] \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}_{t+n-1}) \quad 0 \leq t < T, \tag{90}$$

with the emphasis defined recursively as

$$M_t = I_t + \gamma^n M_{t-n} \tag{91}$$

with $M_t = 0 \ \forall t < 0$.

10 On-policy Control with Approximation

We consider attempts to solve the control problem using parametrised function approximation to estimate action-values. We consider only the on-policy case for now.

10.1 Episodic Semi-gradient Control

Extension of the semi-gradient update rules to action-values is straightforward

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\right] \nabla_{\boldsymbol{w}_t} \hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{92}$$

where U_t is the update target at time t. For example, one-step Sarsa has the update target is

$$U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t).$$

We call this method *episodic semi-gradient one-step Sarsa*. For a constant policy, this method converges in the same with as TD(0), with a similar kind of error bound.

In order to form control methods, we must couple the prediction ideas developed in the previous chapter with methods for policy improvement. Policy improvement methods for continuous actions or actions from large discrete spaces are an active area of research, with no clear resolution. For actions drawn from smaller discrete sets, we can use the same idea as we have before, which is to compute action values and then take an ε-greedy action selection. Episodic semi-gradient sarsa can be used to estimate the optimal action-values as in the box below.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

10.2 Semi-gradient n-step Sarsa

We can use an n-step version of the episodic Sarsa that we defined above by incorporating the bootstrapped n-step return

$$G_{t:t+n} \doteq \sum_{i=1}^{n-1} \gamma^i R_{i+1} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, W_{t+n-1}) \tag{93}$$

where $G_{t:t+n} = G_t$ if $t + n \geq T$, as usual. This update target is used in the pseudocode in the box below. As we have seen before, performance is generally best with amn intermediate value of n.

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$

10.3 Average Reward: A New Problem Setting for Continuing Tasks

We introduce a third classical setting for formulating the goal in Markov decision problems (MDPs) (to go along with episodic and continuing). This new setting is called the *average reward setting*. This setting applies to continuing problems with no start or end state, but also no discounting. (Later we will see that the lack of a start state introduces a symmetry that makes discounting with function approximation pointless.)

In the average reward setting, the ordering of policies is (most often) defined with respect to the *average reward* while following the policy

$$r(\pi) \doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{94}$$

$$= \lim_{t\to\infty} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{95}$$

$$= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r. \tag{96}$$

We will consider policies that attain the maximal value of $r(\pi)$ to be optimal (though there are apparently some subtly distinctions here that are not gone into).

The distribution $\mu_\pi(s)$ is the steady-state distribution defined by

$$\mu_\pi(s) \doteq \lim_{t\to\infty} \mathbb{P}(S_t = s | A_{0:t-1} \sim \pi) \tag{97}$$

which we assume to exist for any π and to be independent of the starting state S_0. This assumption is known as *ergodicity*, and it means that the long run expectation of being in a state depends only on the policy and MDP transition probabilities – not on the start state. The steady-state distribution has the property that it is invariant under actions taken by π, in the sense that the following holds

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s'|s,a) = \mu_\pi(s').$$

In the average-reward setting we define returns in terms of the difference between the reward and the expected reward for the policy

$$G_t \doteq \sum_{i\geq t} (R_{i+1} - r(\pi)) \tag{98}$$

we call this quantity the *differential return* and the corresponding value functions (defined in the same way, just with this return instead) *differential value functions*. These new value functions also have Bellman equations:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + v_\pi(s')\right] \tag{99}$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')\right] \tag{100}$$

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + v_*(s')\right] \tag{101}$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + \max_{a'} q_*(s',a')\right]. \tag{102}$$

We also have differential forms of the TD errors, where $\bar{R}_t$ is the estimate of $r(\pi)$ at t,

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v}(S_t, \boldsymbol{w}_t) \tag{103}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t). \tag{104}$$

Many of the previous algorithms and theoretical results carry over to this new setting without change. For instance, the update for the semi-gradient Sarsa is defined in the same way just with the new TD error, corresponding pseudocode given in the box below.

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

10.4 Deprecating the Discounted Setting

Suppose we want to optimise the discounted value function $v_\pi^\gamma(s)$ over the on-policy distribution, we would choose an objective $J(\pi)$ with

$$
\begin{aligned}
J(\pi) &\doteq \sum_s \mu_\pi(s) v_\pi^\gamma(s) && (105)\\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi^\gamma(s')\right] && (106)\\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\gamma v_\pi^\gamma(s') && (107)\\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s', |s, a) && (108)\\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s')\mu_\pi(s') && (109)\\
&= r(\pi) + \gamma J(\pi) && (110)\\
&\vdots && (111)\\
&= \frac{1}{1-\gamma} r(\pi) && (112)
\end{aligned}
$$

so we may as well have optimised for the *undiscounted* average reward.

The root cause (note: why *root* cause?) of the difficulties with the discounted control setting is that when we introduce function approximation we lose the policy improvement theorem. This is because when we change the discounted value of one state, we are not guaranteed to have improved the policy in any useful sense (e.g. generalisation could ruin the policy elsewhere). This is an area of open research.

10.5 Differential Semi-gradient n-step Sarsa

We generalise n-step bootstrapping by introducing an n-step version of the TD error in this new setting. In order to do that, we first introduce the differential n-step return using function approxi-

mation

$$G_{t:t+n} \doteq \sum_{i=t}^{n-1} \left(R_{i+1} - \bar{R}_{i+1}\right) + \hat{q}(S_{t+n}, A_{t+n}, \boldsymbol{w}_{t+n-1}) \tag{113}$$

with $G_{t:t+n} = G_t$ if $t + n \geq T$ as usual and where $\bar{R}_i$ are the estimates of $\bar{R}$. The n-step TD error is then defined as before just with the new n-step return

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \boldsymbol{w}_t).$$

Pseudocode for the use of this return in the Sarsa framework is given in the box below. Note that $\bar{R}$ is updated using the TD error rather than the latest reward (see Exercise 10.9).

Differential semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_\pi$ or q_*

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, a policy π
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Algorithm parameters: step size $\alpha, \beta > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Initialize and store S_0 and A_0
Loop for each step, $t = 0, 1, 2, \ldots$:
  Take action A_t
  Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$, or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  If $\tau \geq 0$:
    $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$

11 *Off-policy Methods with Approximation

11.1 Semi-gradient Methods

12 Policy Gradient Methods

In this section we take an approach that is different to the action-value methods that we have considered previously. We continue the function approximation scheme, but attempt to learn a *parameterised policy* $\pi(a|s, \boldsymbol{\theta})$ where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ is the policy's *parameter vector*. Our methods might also learn a value function, but the policy will provide a probability distribution of possible actions without directly consulting the value function as we did previously.

We will learn the policy parameter by *policy gradient methods*. These are gradient methods based on some scalar performance measure $J(\boldsymbol{\theta})$. In particular, performance is maximised by *gradient ascent* using some stochastic estimate of J, $\widehat{\nabla J(\boldsymbol{\theta}_t)}$, whose expectation approximates $\mathbb{E}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)]$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta} + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}.$$

Methods that also learn a value function are called *actor-critic* methods. Actor is in reference to the learn policy, while critic is in reference to the learned (usually state-) value function.

12.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterised by any differentiable function of the parameter $\boldsymbol{\theta}$. We generally require $\pi \in (0, 1)$ to be defined on the open interval to ensure exploration.

For action-spaces that are discrete and not too large, it is common to learn a preference function $h(s, a, \boldsymbol{\theta}) \in R$ and then take a soft-max to get the policy

$$\pi(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}.$$

We call this type of parameterisation *soft-max in action preferences*. (Note the homomorphism: preferences add, while probabilities multiply.) We can learn the preferences any way we like, be it linear or using a deep learning.

Some advantages of policy parameterisation:

- Action-value methods, such as ϵ-greedy action selection, can result give situations in which an arbitrarily small change in the action-values completely changes the policy.
- The soft-max method will approach a deterministic policy over time. If we used action-values then these would approach their (finite) true values, leading to finite probabilities (with the soft-max). Action preferences do not necessarily converge, but instead are driven to produce an optimal stochastic policy.
- In some problems, the best policy may be stochastic. Action-value methods have no natural way of approximating this, whereas it is embedded in this scheme.
- Often the most important reason for choosing a policy based learning method is that policy parameterisation provides a good way to inject prior knowledge into the system.

12.2 The Policy Gradient Theorem

The episodic and continuing cases of the policy gradient theorem have different proofs (since they use different formulations of the expected reward). Here we will first focus on the episodic case, but the results carry over the same.

In the episodic case the performance function is the true value of the start state under the current policy

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0).$$

In the following we assume no discounting ($\gamma = 1$), but this can be inserted by making the requisite changes (see exercises).

The success of the *policy gradient theorem* is that it gives a gradient of the performance function that does not include derivatives of the state distribution. The result for the episodic case is as follows and is derived in the box shown below

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}).$$

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.18)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r\,|\,s,a) \big(r + v_\pi(s')\big) \Big] \\
& && \text{(Exercise 3.19 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''\,|\,s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}
$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 199)} \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Eq. 9.3)} \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Q.E.D.)}
\end{aligned}
$$

12.3 REINFORCE: Monte Carlo Policy Gradient

We now attempt to learn a policy by stochastic gradient ascent on the performance function. To begin, the policy gradient theorem can be stated as

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla_{\boldsymbol{\theta}} \pi(a|S_t, \boldsymbol{\theta}) \right].$$

The *all-actions* method simply samples this expectation to give the update rule

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t, a, \boldsymbol{w}) \nabla_{\boldsymbol{\theta}} \pi(a|S_t, \boldsymbol{\theta}).$$

The classical REINFORCE algorithm involves only A_t, rather than a sum over all actions. We proceed

$$\nabla_{\boldsymbol{\theta}} = \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right] \tag{114}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \tag{115}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right], \tag{116}$$

which yields the REINFORCE update

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta}). \tag{117}$$

This update moves the parameter vector in the direction of increasing the probability of the action taken proportional to the return and inversely proportional to the probability of the action. It uses the complete return from time t, so in this sense is a Monte Carlo algorithm. We refer to the quantity

$$\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta})$$

as the *eligibility vector*. Pseudocode is given in the box below (complete with discounting). Convergence to a local optimum is guaranteed under the standard stochastic approximation conditions for decreasing α. However, since it is a Monte Carlo method, it will likely have high variance which will slow learning.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

12.4 REINFORCE with Baseline

The policy gradient theorem can be generalised to incorporate a comparison to a *baseline* value $b(s)$ for each state

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \left(q_\pi(s,a) - b(s)\right) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}). \tag{118}$$

The baseline can be a random variable, as long as it doesn't depend on a. The update rule then becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_t \nabla_{\boldsymbol{\theta}} - b(S_t)\right) \log \pi(A_t|S_t,\boldsymbol{\theta}). \tag{119}$$

The idea of the baseline is to reduce variance – by construction it has no impact on the expected update.

A natural choice for the baseline is a learned state-value function $\hat{v}(S_t, \boldsymbol{w})$. Pseudocode for Monte Carlo REINFORCE with this baseline (also learned by MC estimation) is given in the box below.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

This algorithm has two step sizes $\alpha^{\boldsymbol{\theta}}$ and $\alpha^{\boldsymbol{w}}$. Choosing the step size for the value estimates is relatively easy, for instance in the linear case we have the rule of thumb $\alpha^{\boldsymbol{w}} = 1/\mathbb{E}\left[||\nabla_{\boldsymbol{w}} \hat{v}(S_t,\boldsymbol{w})||_\mu^2\right]$. It is much less clear how to set the step size for the policy parameters.

12.5 Actor-Critic Methods

Although REINFORCE with baseline can use an estimated value function, it is not an actor-critic method because it does not incorporate value estimates through bootstrapping.

We present here a one-step actor-critic method that is an analog of TD(0), Sarsa(0) and Q-learning. We replace the full return of REINFORCE with a bootstrapped one-step return:

$$\boldsymbol{\theta} = \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{v}(S_t,\boldsymbol{w})\right) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{120}$$

$$= \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \gamma \hat{v}(S_{t+1},\boldsymbol{w}) - \hat{v}(S_t,\boldsymbol{w})\right) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{121}$$

$$= \boldsymbol{\theta}_t + \alpha \delta_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{122}$$

with δ_t as the one-step TD error. The natural method to learn the state-value function in this case would be semi-gradient TD(0). Pseudocode is given in the boxes below for this algorithm and a

sister algorithm using eligibility traces.

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

12.6 Policy Gradient for Continuing Problems

For continuing problems we need a different formulation. We choose as our performance measure the average rate of reward per time step:

$$J(\boldsymbol{\theta}) = r(\pi) = \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{123}$$

$$= \lim_{t\to\infty} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{124}$$

$$= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r, \tag{125}$$

where μ is the steady distribution under π, $\mu(s) = \lim_{t\to\infty} P(S_t = s | A_{0:t} \sim \pi)$ which we assume to exist and be independent of S_0 (ergodicity). Recall that this is the distribution that is invariant under action selections according to π:

$$\sum_s \mu(s) \sum_a \pi(a|s, \boldsymbol{\theta}) p(s'|s, a) = \mu(s').$$

We also define the values with respect to the differential return:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots .$$

With these changes the policy gradient theorem remains true (proof given in the book). The forward and backward view equations also remain the same. Pseudocode for the actor-critic algorithm in the continuing case is given below.

Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Algorithm parameters: $\lambda^{\mathbf{w}} \in [0, 1]$, $\lambda^{\boldsymbol{\theta}} \in [0, 1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to s_0)

$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
Loop forever (for each time step):
  $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
  Take action A, observe S', R
  $\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$
  $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S, \mathbf{w})$
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln \pi(A|S, \boldsymbol{\theta})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
  $S \leftarrow S'$

12.7 Policy Parameterisation for Continuous Actions

We simply define a continuous (parameterised) probability distribution over the actions, then do the gradient ascent as above.