Summary of Reinforcement Learning

Sutton and Barto (2018)

Roger

2021-02-12

Introduction

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is a branch of machine learning, and distinct from supervised learning and unsupervised learning. In supervised learning, we have the correct label, and the task for the algorithm is to learn general patterns in the data in order to predict the correct label. During the training, we are able to tell exactly whether or not inference is correct. In this sense, the training information *instructs* the learner. Unsupervised learning on the other hand is about extracting (hidden) structure from unlabeled data.

With reinforcement learning, the objective is for an *agent* to take (repeated) *actions* with the goal of maximizing a (long term) total reward. During learning, we do not and cannot tell the agent exactly whether or not the action is correct. Instead, we can give the agent some (stochastic) reward for taking a particular action. The agent is thus not being instructed, but rather is *evaluating* its actions.

All reinforcement learning agents have explicit goals, can sense aspects of their environment, and can choose actions to influence their environment. A correct choice by the agent requires taking into account indirect, delayed consequences of actions, and thus may require foresight and planning.

Every reinforcement learning system has a number of required elements:

- **The agent**: This is the entity making the decisions
- **The environment**: The collection of choices and states in which the agent operates
- **The policy**: A mapping from perceived states of the environment to actions to be taken when in those states
- **The reward signal**: (Stochastic) functions of the state of the environment and the actions taken. It determines the immediate and intrinsic desirability of environmental states
- **The value function**: The total amount of reward an agent can expect to accumulate overt the future. It is the long term brother of rewards
- **The environment model** (optional): Allowing inferences to be made about how the environment behaves.

The agent seeks actions that bring about states of highest value, *not* highest reward, because these actions obtain the greatest amount of reward for user over the long run.

Chapter 2 - the multi-armed bandit

Studying the multi-armed bandit problem is a useful first step towards understanding reinforcement learning (RL), because it strips away some of the complexities of the "full" RL problem. In particular, the multi-armed bandit requires the agent to learn only in one situation or environmental *state*, with the relevant actions not depending on different states. This is called learning in a *nonassociative* setting. Further, the bandit problem is only concerned with maximizing (expected) direct reward. In contrast, a typical ML problem is concerned with maximizing total (long-term) reward.

Action values

The *k-armed bandit* problem is formulated as follows: You are faced with a repeated choice between k actions (A), each of which gives you some stochastic reward (R). Your objective is to maximize the *long term total* reward. How do you go about choosing the actions in every round?

Every action k has a "true" reward, which is called the *value* of that action, and is written as:

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

where a is the selected action at time t (A_t) and R_t is the corresponding reward. As is clear from the notation, the true value of the action is unknown, and its estimate at time t is denoted by $Q_t(a)$.

One natural way to estimate the true value of an action is to take the sample average of (1):

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_{i,a}}{N_a} \tag{2}$$

which is simply the sum of all rewards of action a until time *t-1*, divided by the number of times action a was taken up until time *t-1*. It is called the *sample average* for estimating action values. A computationally more efficient way of calculating it (without the need for storing all past rewards in memory) is given by:[1]

$$\begin{aligned} Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\ &= \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n}(R_n + (n-1)\frac{1}{(n-1)}\sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n}(R_n + (n-1)Q_n) \\ &= Q_n + \frac{1}{n}(R_n - Q_n) \end{aligned} \tag{3}$$

This action value estimation method is appropriate for problems with stationary action value distributions, where these distributions do not change over time. To see this, note that *all* past rewards contribute equally to the current value estimate of an action in (3). However, when distributions change over time, this approach is no longer valid.

A simple way around it is to interpret *1/n* in (3) as a stepsize parameter α which in this case is time-dependent. When we make it a constant which no longer depends on time (n), we are essentially computing Q_{n+1} as a

[1] For readability we drop the action identifier (a)

weighted average of its initial value (Q_1) and past rewards, with the weights exhibiting exponential decay (i.e. smaller weights furter back in time). The estimated action value then becomes:[2]

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i \tag{4}$$

Action selection

The simplest action selection rule at every time step is to select the action with the highest estimated value, also called a *greedy* action selection method:

$$A_t = \arg\max_a Q_t(a) \tag{5}$$

However, uncertainty around the best action to choose – after all, recall that were *estimating* its true value – leads to an *exploitation-exploration* trade-off. Exploitation dictates that the agent should always follow the selection rule in (5). This will maximize direct (i.e. short-term) reward. However, it also means that the agent might be missing potentially more valuable alternatives, that just happened (by chance) to yield smaller current estimated values. In order to find out and maximize total (long-term) reward, the agent should *explore* alternatives that do no have the highest current estimated value.

A simple way around this dilemma is to follow a "near greedy" action selection method, also called an ϵ-greedy method, because it involves not making the greedy choice ϵ percent of the time, with ϵ typically being small. This changes the selection method to:

$$A_t = \begin{cases} \arg\max_a Q_t(a) \text{ with probability } 1-\epsilon \\ a \sim \text{ Uniform}(\{a_1, .., a_k\}) \text{ with probability } \epsilon \end{cases} \tag{6}$$

This method will make sure that the agent chooses to explore rather than exploit once in a while, throughout all the runs. Note that this might not be the most efficient however, since over time presumably uncertainty around the true action value decreases, and the action with the highest estimated value is also the one with the true highest value, and hence exploitation becomes optimal.

An alternative method is to choose optimistic starting values. The intuition is clear: By choosing (unreasonably) high starting values, we are tricking the agent into exploring all alternatives in the first few timesteps. Whichever the action value estimation method, the (weighted or unweighted) average of a high starting value and a more realstic subsequent value will always be lower than the optimistic value. Hence, a greedy selection method will then proceed to select an action that has not been chosen before, since this still has an optimstic (initial) value.

This method can be quite useful for stationary problems. However, as can bee seen in equation (4), in non-stationary settings, the impact of the initial value on the estimated current value is reduced over time. This implies that in such a setting, optimistic initial values as a way to promote exploration will lose traction over time.

A final method is called *Upper Confidence Bound* (UCB) selection. Unlike ϵ-greedy selection, is does not explore at random, but takes into account the uncertainty of value estimates, by also accounting for their variance. The selection rule in this case changes to:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{7}$$

[2] See equation (2.6) on p.32 for a proof

The latter term in square brackets is a measure of variance, with the control parameter $c > 0$ denoting the degree of exploration. Hence, this method favors selection the action with the highest combination of estimated value (exploitation) and variance (exploration). This has a dual advantage: It increases the probability of finding a higher value action, and it increases the precision of the estimate of the chosen action by adding more evidence.

Evaluation

The book introduces a 10-armed testbed, which is a simulation of a 10-armed bandit in which the agent choses in 1,000 timesteps, acros 2000 different configuration (i.e. action value distributions) of the problem (see Figure 2.1 on p.28).

In terms of performance, ϵ-greedy algorithms perform better than (pure) greedy ones, and smaller values of ϵ generally converge more slowly but more steadily to an optimum long term value. However, optimistic initial values in turn seem to perform better in the long run than ϵ-greedy methods. However, a "parameter test" (reported in Figure 2.6 on p.42) shows that UCB methods tend to outperform the others at the right parameter values.

Chapter 3 - Finite Markov decision processes

(Finite) Markov decision processes (MDP) are an idealized form of the RL problem for which precise theoretical statements can be made. They are similar to the bandit problem in that they involve evaluative feedback, but they are different because (1) they are *associative* (choosing different actions in different situations), and (2) they involve delayed rewards.

Agent and environment

The agent senses (the state of) its environment, and takes particular actions based on that state, as well as the reward is has received from a potential previous action. The environment in turn responds to that action by changing its state, and giving rise to a new reward.

More formally, at every time step t the agent receives some information of the environment's state $S_t \in \mathcal{S}$ and on that basis selects an action $A_t \in \mathcal{A}(s)$. One time step later, and in part as a consequence of its action, the agent receives a reward $R_t \in \mathcal{R}$, and finds itself in a new state S_{t+1}.

In a *finite* MDP, the sets $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions, dependent only on the preceding state and action:

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

In an MDP, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible state S_t and reward R_t only depend on the immediately preceding state S_{t-1} and action A_{t-1}, and given them, not at all on earlier states and actions. The state must include information on all the past agent-environment interactions that make a difference for the future. In this case it is said to have the *Markov property.*

From (8) we can derive a number of other metrics of interest, such as **state transition probabilities**:

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

We can compute **expected rewards for state-action pairs**:

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

The MDP framework is abstract and flexible and can be applied in many different ways. It is useful to realize that the boundary between the agent and the environment is typically not the same as the physical boundary of a robot's arm or an animal's body. Usually, the boundary is drawn closer to the agent than that. In general, anything that cannot be changed by the agent arbitrarily is outside of it, and thus part of its environment.

Goals and rewards

A key concept in RL is sometimes called the *reward hypothesis* and stated as:

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

Hence, if we want the agent to do something for us, we must provide rewards in such a way that in maximizing them, the agent will also achieve our goals. The rewards we setup therefore truly have to indicate what we want to accomplish. The reward signal is your way of communicating to the agent *what* you want it to achieve, not *how* you want it achieved.

Returns and episodes

In general, we seek to maximize the *execpted* return, with the return (G_t) defined as some specific function of the reward sequence. In the simplest case the return is just the sum of the rewards:

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

with T being the final time step. This approach makes sense in applications in which there is a natural notion of a final time step, i.e. when the agent-environment interaction breaks naturally into subsequences, called *episodes*.

Each episode ends in a *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Note that the actual ending (and reward) in the terminal state can different between episodes. Tasks of this kind are called *episodic tasks* (e.g. plays of a game, runs through a maze). In these tasks, it is sometimes useful to distinguish all non-terminal states $\mathcal{S}$, from the set of all states (including the terminal one) $\mathcal{S}^+$. T is a random variable that typically differs between episodes.

In contrast to episodic tasks are *continuing tasks*, that don't break naturally into identifiable episodes. The return formulation (11) is problematic for these tasks, because $T = \infty$, making the return infinite as well. To deal with this, we introduce discounting to (11):

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

with $0 \leq \gamma < 1$ as the discount factor. Another, computationally more efficient way of writing (12) is:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{13}$$

However, it is not convenient to have two separate return formulations for episodic and continuous tasks. Therefore, (12) is generally rewritten, with the convention of omitting episode numbers k when they are not needed, and including the possibility that $\gamma = 1$ if the sum remains defined (i.e. when all episodes terminate):

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

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

Policies and value functions

Value functions are functions of states that estimate how good it is for the agent to be in a given state. Action-value functions are functions of state-action pairs that estimate how good it is for the agent to perform a given action in a given state. Since the future rewards (i.e. expected returns) determine how good a state

is, and since these in turn depend on the actions the agent will take, (action-)value functions are defined with particular ways of acting, called *policies*.

Formally, a *policy* is a mapping from states to probabilities of selecting each possible action. $\pi(a|s)$ then is the probability that the agent will take action $A_t = a$ if $S_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in state s and following policy π thereafter. For MDPs, it is called the *state-value function for policy* π and can be written as:

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

Note that the value of the terminal state – if any – is always zero, since there is not reward after the final state.

A similar formulation can be derived for the *action-value function of policy* π $q_\pi(s, a)$. It captures the expected return starting from state s, taking the action a, and following policy π thereafter:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \text{ for all } s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{16}$$

A fundamental property of both value functions and action-value functions is that they satisfy recursive relationships, similar to that for the return formulation in (13). That is, the value of any state s can be expressed in terms of the value of future states s'. This is the definition of a *Bellman equation* and written as follows:

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

Note how the random reward variable R_t in the second line is replaced by the actual realized reward r in line three. Also note how the final expression is actually an expected value. It is a sum over all values of the three variables a, s' and r. For each triple, we compute its probability $\pi(a, s)p(s', r|s, a)$, weight the quantity in brackets by that probability, and sum over all possible values to get an expected values. The Bellman equation is useful because it reduces an unmanageable (possibly infinite) sum over possible futures, to a simple linear algebra problem.

Figure 1 illustrates this graphically. Starting from state s in the root node, the agent can take three possible actions based on the policy π. The function $\pi(a, s)$ specifies the probability distribution across these three possible actions. Following each of these actions, the environment can respond with one of several next states s' (in this case, two are shown for each action), along with a reward r. These probabilities are in turn specified by $p(s', r|s, a)$. The Bellman equation in (17), averages over all these possibilities, weighting each by its probability of occurring. It states that the value of the start state s must equal the (discounted) value of the expected next state, plus the reward expected along the way.

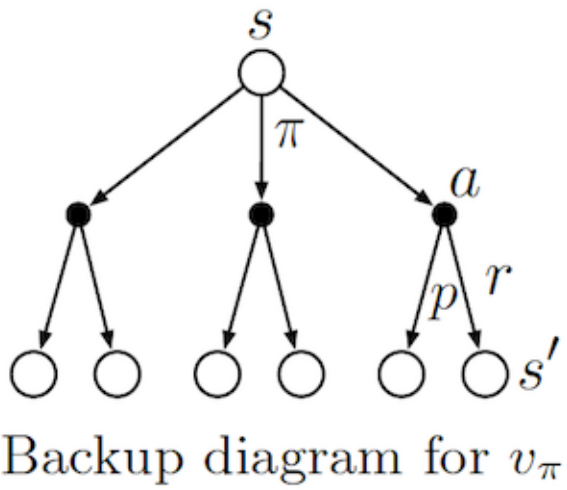

Figure 1: Backup diagram

Figure 2 shows an example of simple finite MDP. The cells of the grid correspond to the states of the environment. In each cell, the four actions are possible: north, east, south or west. The deterministic new state s' is the next cell in that direction. Actions that take the agent off the grid leave the location unchanged. There are two special states: A and B. From state A, all four actions lead to state A', with a reward of +10. From state B, all four actions lead to a state B', with reward of +5. All other rewards are 0, except when bumping into the wall (i.e. stepping off the grid), which has reward -1.

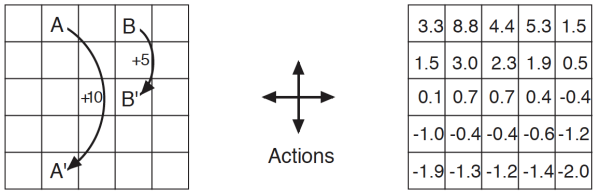

Figure 2: Gridworld

The right part of the figure shows the state-values computed from (17) (rounded to one decimal) for a uniform policy π (i.e. $\pi(a, s) = 0.25$ for all $a \in \mathcal{A}$), and with a discount of $\gamma = 0.9$. Note that this is a simplified case, because following an action a, subsequent states s' and rewards r are deterministic, i.e. $p(s', r|s, a) = 1$. For example, starting from the center cell on the grid, and using the Bellman value function (17), we obtain:

$$\begin{aligned} v_\pi(s) =& 0.25 \times (0 + 0.9 * 2.3) + \\ & 0.25 \times (0 + 0.9 \times 0.4) + \\ & 0.25 \times (0 + 0.9 \times -0.4) + \\ & 0.25 \times (0 + 0.9 \times 0.7) \\ =& 0.675 \end{aligned} \tag{18}$$

Notice the negative values near the edge of the grid. These are the result of the high probability of hitting the edge of the grid under the uniform policy. Further, state A is the best state to be in under this policy, but its expected return is less than 10 (its immediate reward), because from state A the agent is taken directly to state A', from which it is more likely to bump into the edge of the grid. For a similar (but opposite) reason, state B actually yields a higher expect return than its immediate reward.

Optimal policies and value functions

Solving a reinforcement learning problem mean, roughly, finding a policy that achieves a lot of reward over the long run. Value functions define a partial ordering over policies. A policy π is defined to be better or equal than a policy π' if its expected return is greater than or equal to that of π' for all states. That is, $\pi \geq \pi'$ if and only if $\upsilon_\pi(s) \geq \upsilon_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better or equal than all other policies. This is called an optimal policy, denoted by π_*.

To see why there always must be a least one optimal policy, consider a situation in which policy π_1 yields the highest expected return in states $s \in \mathcal{S}_\infty$, and policy π_2 yields the highest expected return in states $s \in \mathcal{S}_\in$, such that $\mathcal{S} = \mathcal{S}_\infty \cup \mathcal{S}_\in$. Then we can define an optimal policy π_* that mixes these two policies to achieve the highest expected return in all states.

All optimal policies share the same state-value function, called the *optimal state value function* which is defined as:

$$\upsilon_*(s) = \max_\pi \upsilon_\pi(s) \text{ for all } s \in \mathcal{S} \tag{19}$$

Optimal policies naturally also share the same *optimal action value function* defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \text{ for all } s \in \mathcal{S}, a \in \mathcal{A} \tag{20}$$

For the state-action pair (s,a), this function gives the expected return for taking action a in state s and then following the optimal policy π_*. That is, it gives the values after comitting to a particular *first* action, but afterward choosing whichever actions are best. As such, we can write q_* in terms of υ_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \upsilon_*(S_{t+1}) | S_t = s, A_t = a] \tag{21}$$

We can write the Bellman equation in (17) for the optimal value function as well, which is then called the *Bellman optimality equation*. One obvious way to do that would be to simply replace π in (17) with π_*:

$$\upsilon_*(s) = \sum_a \pi_*(a|s) \sum_{s',a} p(s', r|s, a) \left[r + \gamma \upsilon_\pi(s') \right] \text{ for all } s \in \mathcal{S} \tag{22}$$

Alternatively, we can replace the sum over π with maximizing everything after the double sum with respect to action a:

$$\begin{aligned}
\upsilon_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma \upsilon_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma \upsilon_*(s')]
\end{aligned} \tag{23}$$

Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state. The cost of redefining the Bellman optimality equation in this way is that it no longer represents a *linear* system of equations, since taking the maximum is a non-linear operation. The benefit is that the Bellman optimality equation no longer depends on π_*, which is unknown to us. After all, it is exactly π_* that we are trying to find in our RL task.

The Bellman optimality equation for the action-value function is given by:

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

The expression in 23) is actually a system of (nonlinear) equations, one for each state. If there are n states, there are n unknowns (i.e. best actions). If the dynamics of p are known, this system can be solved using any one of the different methods for solving systems of nonlinear equations. For finite MDPs, there will be a unique solution.

The beauty of the Bellman optimality equation in (23) is that it simplifies the problem to a one-step search problem. If you have the optimal value function v_*, then the actions that appear best after a one-step search will be optimal actions. Stated different, any policy that is *greedy* with respect to optimal value function will be an optimal policy.

To rephrase this benefit, if one uses v_* to evaluate the short-term consequences of actions, then you are also optimizing for the long run, because v_* takes into account the reward consequences of all possible future behavior. By means of v_*, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state.

Having q_* makes choosing optimal actions even easier. In this case, the agent does not even have to do one-step ahead search. For any state s, the agent can simply find an action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step ahead searches, by already working out the dynamics of p belong to every action. This makes it possible to choose optimal actions without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Figure 3 shows the optimal value and policy outcome for the example in Figure 2. Note that now, under an optimal policy rather than a uniform one, state A indeed has the highest expected return, and the policies in the right diagram indeed move to agent towards state A.

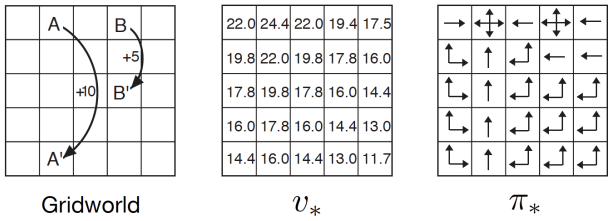

Figure 3: Gridworld - optimal values and policy

Explicitly solving the Bellman optimality equation in (23) is rarely ever useful in practice. First, it assumes we accurately know the dynamics of the environment. Second, it assumes that we have enough computational resources to complete the computation of the solution. Finally, it assumes the Markov property, i.e. that all states include information on all the past agent-environment interactions that make a difference for the future. In practice, one or a varying combination of these conditions is often not met, and we have to settle for approximate solutions.

Chapter 4 - Dynamic programming

Chapter 3 discussed MDPs, and the equations that govern optimal value functions and action-value functions (i.e. the Bellman equations). We have not yet touched on the question of how to go about solving the (set of) optimality equation(s) in order to find the optimal policy. This chapter will start a discussion of (approximate) methods of doing so.

The term *Dynamic Programming* (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. The key of idea of DP, and of RL in general, is the user of value functions to organize and structure the search for good policies. Specifically, DP algorithm are obtained by turining Bellman equations into assignments, or (iterative) update rules for improving approximations of the desired value functions.

Policy evaluation (prediction)

We often talk about to distinct tasks: *Policy evaluation* and *control*. Policy evaluation is the task of determining the value function for a specific policy. Control is the task of finding a policy to obtain as much reward as possible. In other words, finding a policy which maximizes the value function. Although *control* is the ultimate goal of RL, the task of policy evaluation is a necessary first step. After all, it's hard to improve our policy if we don't have a way to assess how good it is.

Recall the Bellman value function from chapter 3:

$$\begin{aligned} \upsilon_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+t}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s',a} p(s', r|s, a) \left[r + \gamma \upsilon_\pi(s')\right] \text{ for all } s \in \mathcal{S} \end{aligned} \tag{25}$$

If the environment's dynamics $p(.)$ are completely known, this system of $|\mathcal{S}|$ linear equations in $|\mathcal{S}|$ unknowns (i.e. the $\upsilon_\pi(s)$, $s \in \mathcal{S}$) can solved with straightforward (yet tedious) algebra. Instead, we consider an iterative method here, which works more generally.

Consider a sequence of value functions $\upsilon_0, \upsilon_1, \upsilon_2, ...$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$. The initial approximation υ_0 is chose arbitrarily, and each successive approximation is obtained by using the Bellman equation in (25) as an update rule:

$$\begin{aligned} \upsilon_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma \upsilon_k(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',a} p(s', r|s, a) \left[r + \gamma \upsilon_k(s')\right] \text{ for all } s \in \mathcal{S} \end{aligned} \tag{26}$$

Since $\upsilon_k = \upsilon_\pi$ is a fixed point taking us back to the Bellman equation, this approach is guaranteed to converge to the true value function (asymptotically). The algorithm is called *iterative policy evaluation*.

To produce each successive approximation υ_{k+1} from υ_k, iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value, obtained from the old value of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. This kind of operation is called an *expeced update*. They are called this because they are based on an expectation over all possible next states, rather than on a sample next state. One can think of

the updates as being done in a *sweep* through the state space (with one update sweeping through all the states once).

One subtlety in terms of implementation concerns whether we keep two value arrays – one for the old values and one for the new ones after the update (whereafter the new values replace the old, and the process repeats again) – or just one, and we replace old values with updated ones *in place*. In the latter approach, the order of updating matters, and some state value updates will already be confronted with updated values of successor states. Indeed, it turns out that convergence in this approach generally is faster.

Figure 4 presents an algorithm for iterative policy evaluation in pseudo code. Although formally it only converges asymptotically, it practice it must be stopped short of this. Specifically, the algorithm will stop when the no state value update exceeds some (small) threshold value θ anymore.

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

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

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

Figure 4: Pseudo code for iterative policy evaluation

It is useful to again use a gridworld representation to clarify some of these ideas. Figure 5 presents the setup. The nonterminal states are $\mathcal{S} = \{1, 2, ..., 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$, which deterministically cause the corresponding state transitions (i.e. we can ignore $p(.)$ in (26)), except that actions which would take the agent off the grid leave the state unchanged. It is an *undiscounted*, *episodic* task. The rewards is -1 on all actions (transitions), until the terminal state is reached, which is shaded in the grid.

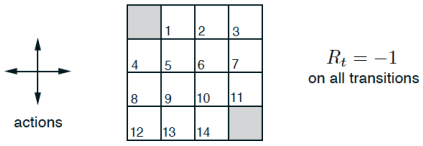

Figure 5: Gridworld setup

Suppose that we want to evaluate the value function under the uniform random policy, i.e. all actions are equally likely in every state. The left side of Figure 6 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The initial values v_0 are set to 0 for all $s \in \mathcal{S}$. This approach uses the "two arrays" alternative discussed above, where the values in each state space are updated using the values of the previous one. The final estimate in Figure 6 is in fact the v_π, which in this case gives for each state the negation of the expected number of steps from that state until termination (due to the reward scheme).

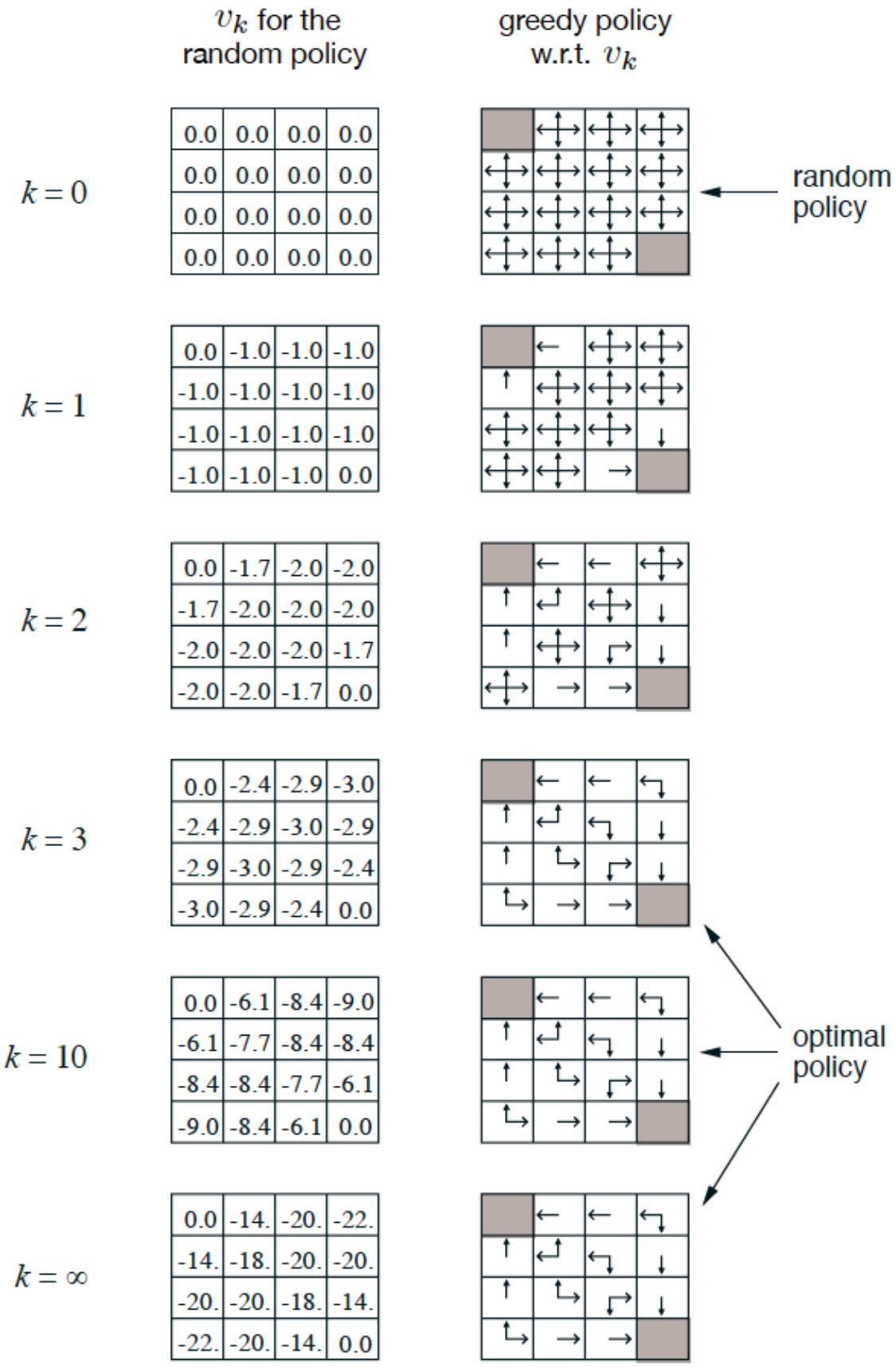

Figure 6: Gridworld - iterative policy evaluation and control

Policy improvement

How do we use policy evaluation in our goal of finding better policies? Suppose we have determined v_π for some arbitrary deterministic policy π. We can now consider state s, and try choosing $a \neq \pi(s)$, but follow policy π thereafter. To establish the value of doing so, we can use the action-value function of policy π:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r'} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{27}$$

We can compare this action-value function to v_π to evaluate whether this generates an improvement over the original policy. From this follows the *policy improvement theorem*. Consider a policy π' which is identical to policy π except that $\pi'(s) = a \neq \pi(s)$. Then it must hold that:

$$\begin{aligned} q_\pi(s, \pi'(s)) >= q_\pi(s, \pi(s)) \text{ for all } s \in \mathcal{S} &\iff \pi >= \pi' \\ q_\pi(s, \pi'(s)) > q_\pi(s, \pi(s)) \text{ for at least one } s \in \mathcal{S} &\iff \pi > \pi' \end{aligned} \tag{28}$$

It is now a natural extension to consider changes at *all* states and to *all* possible actions, selecting at each state the action that appears best according to $q_\pi(s, a)$. That is, to consider the new greedy policy π' given by:

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{29}$$

This process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*. It must always give us a policy improvement, except when the original policy is already optimal (satisfying the Bellman optimality equation in (23)).

All the ideas just described also carry over to the situation of stochastic policy $\pi(a|s)$. In addition, if there are ties in policy improvement steps such as (**??**), then in the stochastic case we need not select a single action from among them.

The bottom-right grid in Figure 6 shows an example of policy improvement over the uniform random policy in the top-right grid. It is based on the value function of the uniform random policy given in the bottom-left grid. It is straightforward to see that the policy in the bottom-right grid, by being greedy with respect to the value function of the uniform random policy, is an improvement over that policy. The value function of this new policy $v_{\pi'}(s)$ can be seen by inspection to be either -1, -2, or -3. at all states $s \in \mathcal{S}$, whereas $v_{pi}(s)$ is at most -14. In other words, $v_{\pi'}(s) > v_\pi(s)$ for all $s \in \mathcal{S}$, implying that $\pi' > \pi$.

Policy iteration

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

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0)} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} ... \pi_* \xrightarrow{\text{E}} v_{\pi_*} \tag{30}$$

where E denotes policy evaluation and I policy improvement. This way of finding an optimal policy is called *policy iteration*, and a complete algorithm is given in Figure 7 in pseudo-code.

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 $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
   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

Figure 7: Pseudo code for policy iteration

The right column in Figure 6 illustrates the steps in policy iteration. It starts at the top-left with the random initialization of the value function. Then the first policy is the random uniform policy in the top-right. We then engage in the process of *iterative policy evaluation*, eventually yielding the value function in the bottom right, as discussed before. Finally, the grid in the bottom-right shows the policy that is greedy with respect to this value function. As can be seen directly (and as was discussed in the previous section), this is the optimal policy. So here, in just one E/I step, we arrive at the optimal policy.

The other grids on the right of Figure 6 show greedy policies with respect to the different steps in the initial policy evaluation (with a small mistake in the third row). This demonstrates that even during the process of initial iterative policy evaluation, the optimal policy is already found. However, this is primarily due to the simplicity of the problem, and not a general result.

Value iteration

One drawback of policy iteration is that each of its iterations involves policy evaluation, which itself may be a protracted iterative computation requiring multiple sweeps through the state set. To visualize this, consider Figure 8, which shows how policy iteration alternatives between policy evaluation ($v = v_\pi$) and policy improvement ($\pi = \text{greedy}(v)$). Note how each step runs all the way to completion.

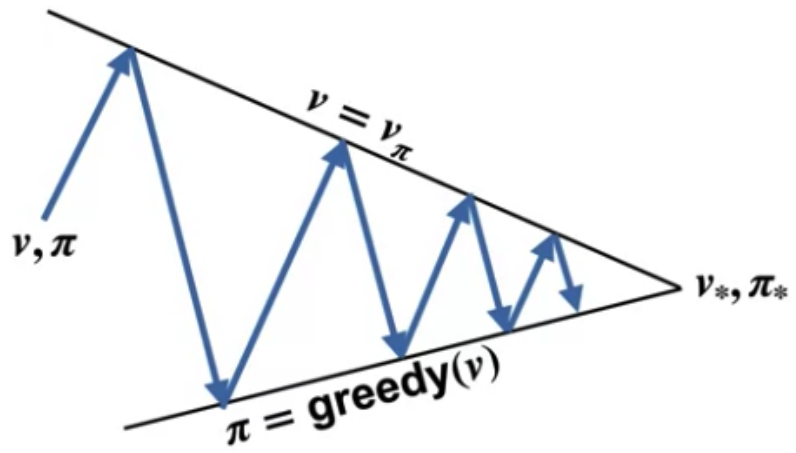

Figure 8: Complete policy iteration

However, we can relax this approach by not taking each step all the way, instead taking us only a little closer to policy evaluation and policy improvement. This is illustrated in Figure 9. As can be seen from the figure, eventually we still end up at the optimal policy π_* and value function v_*, despite taking incomplete steps. This is called *generalized policy iteration*. If refers to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. As illustrated in Figure 9, the value function only stabilizes when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.

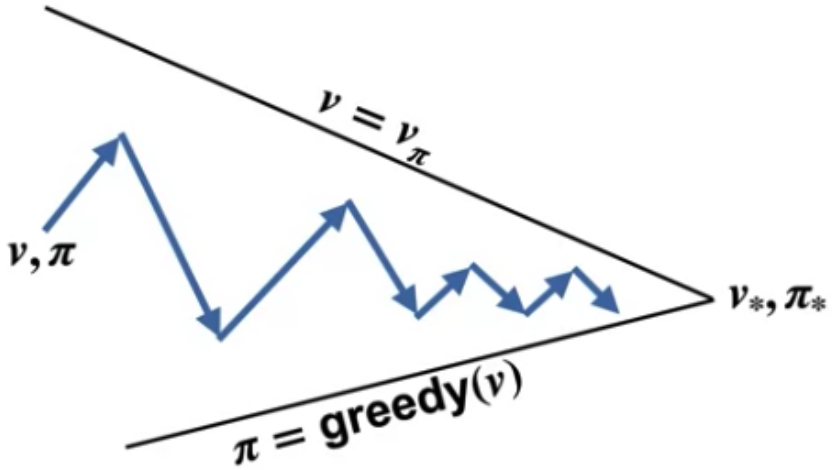

Figure 9: Generalized policy iteration

One important special case of generalized policy iteration is when policy evaluation is stopped after just one sweep (i.e. one update of *each* state). This algorithm is called *value iteration*:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \max_a \sum_{s',a} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \text{ for all } s \in \mathcal{S} \end{aligned} \tag{31}$$

Note that we are simply turning the Bellman optimality equation in (**??**) into and update rule. Also note

how this update is identical to the policy evaluation update in (26), except that it requires the maximum to be taken over all actions. In this way, value iteration combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Figure 10 provides the pseudo-code for the algorithm.

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

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

Figure 10: Pseudo code for policy iteration

Asynchronous dynamic programming

Methods like complete policy iteration and value iteration all sweep the entire state space on each iteration. This can be problematic, or even prohibitive, when the state space is large. Instead, *asynchronous* dynamic programming algorithms are in-place iterative algorithms that are not organized in terms of systematic sweeps of the entire state set. These algorithms update the value of the states in any order whatsoever, using whatever values of other states happen to be available. The value of some states may be updated several times before the values of tothers are updates once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all states: it can't ignore any state after some point in the computation.

Chapter 5 - Monte Carlo Methods

In Chapter 4, we assumed complete information of the environment. In particular, we assumed that the MDP's transition dynamics $p(s', r|s, a)$ are known. In reality, this is often not the case, and the DP methods discussed in Chapter 4 can no longer be used. Monte Carlo (MC) methods require only *experience* – i.e. sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. These methods are a way of solving the RL problem based on averaging sample returns.

There are three additional advantages of MC methods vis-a-vis DP methods. First, they can be used with simulation or sample models. Second, it is easy and efficient to *focus* MC methods on a small subset of the states. Third, they may be less harmful to violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, they do not *bootstrap*.

In this book, MC methods are defined only for episodic tasks. Only on the completion of an episode are value estimates and policies changed. MC methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense.

MC methods are a bit like the bandit problem discussed in Chapter 2. There also, the expected return was estimated as the average of a number of observed returns. However, in that case the rewards were generated by pulling the arm, whereas in this case they are generated by *policies*.

MC prediction

One way to estimate the state-value function for a given policy, is to average the returns observed after the first visit to that state per episode. The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following the first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s. Pseudo-code for an every-visit MC method is shown in Figure 11.

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, $t = T-1, T-2, \ldots, 0$:

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

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 11: Pseudo code for every-visit MC prediction

The idea is to first generate a complete episode following policy π. We then work our way backward from the penultimate terminal step to the first time step in the episode to estimate the returns (note that the return in the terminal step is 0 by definition, which we use to initialize G). Figure **??** provides a simple example of working out the return estimate in each time-step. After each of these steps, we first append G to the list (of

lists) of returns for state $s = S_t$, and then update $V(S_t)$ with the average of all collected returns for $s = S_t$ until that point.[3]

When we contrast MC learning with DP methods, we observe three notable differences. First, we do not need to keep a large model of the environment – in particular, we do not require transition probabilities. This is because MC methods learn directly from experience. Second, MC methods can estimate the value of a state independently of the values of any other states. In contrast, in DP the value of each state depends on the value of other states. Finally, the computation needed to update the value of each state along the way doesn't depend in any way on the size of the MDP. Rather, it depends on the length of the episode.

The book provides a good illustration of this method in the context of Blackjack (pp. 93-95).

MC estimation of action values

With a model ($p(s', r|s, a)$), state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state. This is so because the model allows you to compute the weighted sum of state values s' when taking action a in state s. Without a model, however, this is not possible, because the probability weights cannot be computed. Hence, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, one of our primary goals for MC methods is to estimate q_*.

Let us first consider policy evaluation for action values. The objective is to estimate $q_\pi(s, a)$, i.e. the expected return of taking action a when starting in state s, and then following policy π thereafter. The MC methods for this are the same as those in Figure 11, except now we talk about visits to a state-action pair rather than to a state.

The only complication is that many state-action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the MC estimates of the other actions will not improve with experience.

This is the general problem of *maintaining exploration*. For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes start in a state-action pair, and that every pair has a nonzero probability of being select as the start. This is called the *exploring starts* assumption. The most common alternative to this approach is to consider only policies that are stochastic with a nonzero probability of selection all actions in each state.

MC control

TO go from MC policy evaluation to control, we proceed in similar fashion as with the DP methods in Chapter 4. That is, according to the idea of generalized policy iteration. Let's first consider an MC variation of classic policy iteration:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0)} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} ... \pi_* \xrightarrow{\text{E}} q_* \tag{32}$$

Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. Policy improvement is done by making the policy greedy with respect to the current action-value function. Because in this case we are using an *action-value* function rather than a *value* function, no model is needed to construct the greedy policy. For any action-value function q, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action value:

[3]Note that it is more efficient to not keep a full list of all sample returns for S_t, but instead update incrementally using the rule (from Chapter 2): $V(S_t) = V(S_t) + \alpha[G_t - V(S_t)]$ with G_t being the actual return following time t and α being a constant step-size parameter.

$$\pi(s) = \arg\max_a q(s, a) \tag{33}$$

In this way, MC methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

Rather than using the classic version of policy iteration, we can improve efficiency by avoiding *complete* policy evaluation before returning to policy improvement, as was also discussed in the context of DP methods in Chapter 4. There, we saw an extreme version of that approach called *value iteration*, in which only one iteration of policy evaluation is performed between each step of policy improvement. For MC policy iteration, it is natural to alternate between policy evaluation and improvement on an episode-by-episode basis. Pseudo-code for this approach, in combination with exploring starts, is given in Figure 12.

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 $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

Figure 12: Pseudo code for every-visit MC policy iteration with exploring starts

MC control without exploring starts

There are two approaches as alternatives to the exploring starts assumption, resulting in either *on-policy* methods or *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. All the methods discussed so far (in this and the previous chapter) are examples of on-policy methods.

In on-policy control methods the policy is usually *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. An example of such a policy, reminiscent of what was discussed in Chapter 2, is an ε-greedy policy, meaning that most of the time it chooses an action that has maximal estimated action value, byt with probability ε it instead selects an action at random.

That is, all non-greedy actions are given the minimal probability of selection $\frac{\varepsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$ is given to the greedy action.[4]

[5]: For example, suppose $|\mathcal{A}(s)| = 10$ (one of which is the greedy action), and $\varepsilon = 0.1$. Then all nine non-greedy

[4] epsilon_greedy_note
[5] epsilon_greedy_note

actions receive probability 0.01, leaving a probability of (1 - 0.1 + 0.01) for the greedy action.

As before, we still user (every visit) MC methods to estimate the action-value function for the current policy. Without exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of non-greedy actions. In our on-policy method, we will move it only to an ε-greedy policy. For any ε-soft policy π, any ε-greedy policy with respect to q_π is guaranteed to be better than or equal to π. Figure 13 shows the pseudo-code for this approach.

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, $t = T-1, T-2, \ldots, 0$:

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

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

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

Figure 13: Pseudo code for every-visit MC policy iteration for ε-soft policies

If our policy always gives at least ε probability to each action, it's impossible to converge to a deterministic optimal policy. Exploring starts can be used to find the optimal policy. But ε-soft policies can only be used to find the optimal ε-soft policy. That is, the policy with the highest value in each state out of all the ε-soft policies. Still, optimal ε-soft policies still perform reasonably well in most cases, and allow use to get rid of the impractical assumption of exploring starts.

Off-policy prediction via importance sampling

The on-policy (ε-soft) approach discussed in the previous section is a compromise between exploitation and exploration: it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to use two policies: one that is learned about – called the *target policy* – and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior – called the *behavioral policy*. In this case, we say that learning is from data "off" the target policy, and the overall process is termed *off-policy learning*.

Because the data us due to a different policy, off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful and general. In particular, note that they include on-policy methods as a special case.

To fix ideas, suppose that we wish to estimate υ_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$. In this case, π is the target policy, b is the behavioral policy, and both policies are considered fixed and given. In order to use episodes from b to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under b. That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of *coverage*. It follows from coverage that b must be stochastic in states where it is not identical to π.

Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*.

Specifically, give as starting state S_t, the probability of the subsequent state-action trajectory A_t, S_{t+1}, $A_{t+1}, ..., S_T$ under any policy π is:

$$\begin{aligned}
&\Pr(A_t, S_{t+1}, A_{t+1}, ..., S_T | S_t, A_{t:T-1}\ \pi) \\
&\quad = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1})...p(S_T|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)
\end{aligned} \tag{34}$$

Hence, the relative probability of the trajectory under the target and behavior policies (i.e. the importance-sampling ratio) is:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{35}$$

Thus, the importance-sampling ratio ends up depending only on the two policies and the sequence, not on the MDP. We use the importance-sampling ratio to properly weight the expected returns that we get due to the behavioral policy, in order to better approximate those under the target policy. That is, the returns $\mathbb{E}[G_t|S_t = s] = \upsilon_b(s)$ yields the wrong expectation, and cannot be averaged to obtain υ_π. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value, $\mathbb{E}[\rho_{t:T-1}G_t|S_t = s] = \upsilon_\pi(s)$.

Using the importance-sampling ratio in (35) we can adjust the pseudo-code in Figure 11 to the one shown in Figure 14. Note that the episode in this case is generated using policy b instead of π, and the returns get weighted by W. Also note that W gets updated by $W\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$. To see why, first note that ρ can also be written as:

$$\begin{aligned}
\rho_{t:T-1} &= \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \\
&= \rho_t \times \rho_{t+1} \times \rho_{t+2} ... \rho_{T-1}
\end{aligned} \tag{36}$$

Further note that in Figure 14 we loop over time steps backwards (as in all MC algorithms discussed before). This means we start with $W_1 = \rho_{T-1}$, then $W_2 = \rho_{T-1}\rho_{T-2} = W_1\rho_{T-2}$. In other words, we can compute ρ recursively, which is done in Figure 14.

Armed with this approach, we can now average returns from a batch of observed episodes following policy b to estimate $\upsilon_\pi(s)$. We define the set of all time steps in which state s is visited as $\mathcal{T}(s)$ (i such a way that we keep counting timesteps across episodes, i.e. without resetting it to 0 when starting a new episode). Further, let $T(t)$ denote the first time of termination following time t, and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertain to state s, and $\{\rho_{t:T(t)-1}\}_{t\in\mathcal{T}(s)}$ are the corresponding

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, **arbitrarily, for all** $s \in S$

$Returns(s) \leftarrow$ **an empty list, for all** $s \in S$

Loop forever (for each episode):

Generate an episode following b $S_0, A_0, R_1, S_1 \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0 \;\; W \leftarrow 1$

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

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

Append G **to** $Returns(S_t)$

$V(S_t) \leftarrow$ **average**$(Returns(S_t))$

$$W \leftarrow W \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$$

Figure 14: Pseudo code for every-visit MC policy iteration for off-policy prediction

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

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|} \tag{37}$$

When importance sampling is done like this, with a simple average, it is called *ordinary importance sampling*. An important alternative is *weighted importance sampling*, which uses a weighted average:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\rho_{t:T(t)-1}} \tag{38}$$

Ordinary importance sampling is unbiased whereas weighted importance sampling is biased. On the other hand, the variance of ordinary importance sampling is in general unbounded because the variance of the ratios can be unbounded, whereas in the weighted estimator the largest weight on any single return is one. Indeed, in practice the weighted estimates usually has a dramatically lower variance and is strongly preferred.

Chapter 6 - Temporal-Difference Learning

A big drawback of the MC approach to learning discussed in the previous chapter, is that value prediction has to wait until the episode is finished. Only then can the return (G_t) be computed. This can be clearly seen from all the algorithms in Figures 11-14, in which a full episode is generated first, and then we loop through that episode backwards in time. This approach is fine for offline learning, but problematic for online learning, where we would like to learn on the fly. Temporal-Difference (TD) learning allows for learning after each time-step, rather than after each episode.

TD prediction

Both TD and MC methods use experience to solve the prediction problem. MC methods waid until the return following a visit to state S_t is known, and then use that return as a target for $V(S_t)$:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{39}$$

Let's call this method *constant-α MC*. MC methods must wait until the end of the episode to determine the increment to $V(S_t)$, because only then G_t is known. TD methods on the other hand only wait until the next time step. At time $t+1$ they directly form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. In the simplest case, the update becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{40}$$

In other words, the *target* for the MC update is G_t, whereas the target for the TD update i $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called $TD(0)$, or *one-step TD*. Figure 15 gives pseudo code for its algorithm. Because $TD(0)$ bases its update in part on an existing esimate, we say that it is a *bootstrapping* method, like DP.

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

Figure 15: Pseudo code for TD(0) policy prediction

Recall from Chapter 3 that:

$$\begin{aligned} \upsilon_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma \upsilon_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{41}$$

MC methods use an estimate of the first line in (41) as a target, because this expected value is not known; a sample return is used in its place. DP methods use an estimate of the last line in (41), because $\upsilon_\pi(S_{t+1})$ is not known, and the current estimate $V(S_{t+1})$ is used in its place.[6] The TD target is an estimate for *both* reasons: it samples the expected values in the last line in (41) *and* it sues the current estimate V in place of the true υ_π. Thus, TD methods combine the sampling of MC with the bootstrapping of DP.

We refer to TD and MC updates as *sample* updates. They are different from *expected* updates (like for DP) in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

Note that the quantity in brackets in (40) is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity is called the *TD error*:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{42}$$

Note that δ_t is the error in $V(S_t)$, available at time $t+1$. If the array V does not change during the episode (as it does not in first-visit MC methods), then the MC error can be written as a sum of TD errors as follows:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma\delta_{t+1}\gamma^2(G_{t+2} - V(S_{t+2})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T)) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0)) \\ &= \sum_{k=t}^{T_1} \gamma^{k-1}\delta_k \end{aligned} \tag{43}$$

Suppose that, at different stages from you car drive from work to home, you are trying to predict how much your total travel time is. Figure 16 shows the differences between MC and TD methods for approaching that task. On the left, you see the MC approach (with $\alpha = 1$). The dotted line shows the actual outomce at the end of the episode, and the MC errors in each stage are the difference between the predicted total travel time at that stage *versus* the actual total travel time (i.e. the red arrows in the figure). Learning can only begin after the episode has finished, because only then the actual outcome is observed.

But is it really necessary to wait until the final outcome is known before learning can begin? Not following a TD approach, in which the prediction changes (red arrows) in each stage are with respect to the updated prediction in the next stage. That, is you predict when leaving the office that it would take you 30 minutes to get home, but half an hour later, you are still stuck in a traffic jam and estimate that it will take another 30 minutes to get home, you already know then that you initial estimate of 30 minutes was too optimistic. Indeed, following TD, you would shift you initial estimate immediately from 30 minutes to 60 minutes. Indeed, each estimate would be shifted toward the estimate that immediately follows it, as shown in the right plot of Figure 16

[6] Note that DP methods are *not* an estimate due to $\mathbb{E}$ appearing in the last line of (41), since a model of the environment is assumed to be completely known in this case.

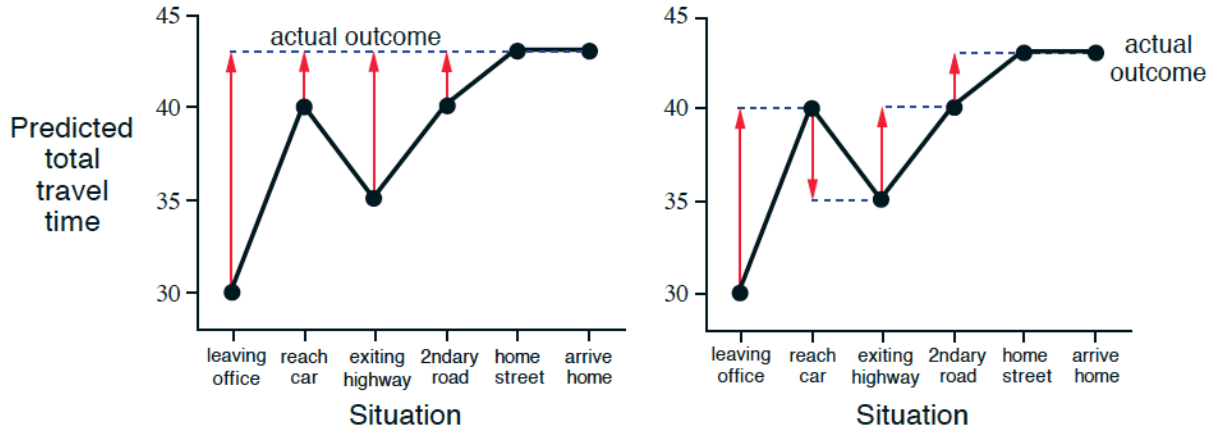

Figure 16: Learning updates in MC vs TD methods

Optimality of TD(0)

In practice, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks. Why? To understand this, we first need to understand *batch updating*. This is a method that is often applied when the is only a finite amount of experience available, say 10 episodes or 100 timesteps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function V, the increments specified in (39) or (40) are computed for every time step t at which a non-terminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on until the value function converges.

When using this approach, a general difference between MC and TD(0) methods arises. Batch MC methods always find the estimate that minimize the mean-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the *maximum-likelihood* model of the Markov process. This is called the *certainty-equivalence* estimate because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. This explains why TD(0) methods converge more quickly than MC methods.

Sarsa: On-policy TD control

As with MC methods, also for TD methods there is a challenge to trade off exploration and exploitation when doing policy control. The first step is to learn *action-value* functions. This can be done using essentially the same TD method described above for learning v_π. We consider the transition from state-action pair to state-action pair, and learn the values of state-action pairs:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{44}$$

This update rule uses every element in $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, giving rise to the name *Sarsa*.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy control methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. Figure 17 gives the general form of the control algorithm.

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 $Q(\mathit{terminal}, \cdot) = 0$

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

Figure 17: Sarsa (TD) (on) policy control

Q-learning: Off-policy TD control

To achieve an off-policy method of TD control, we have to adjust the update rule in (44) as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{45}$$

In this case, the learned action-value function Q directly approximates q_*, the optimal action-value function, regardless of the policy being followed (i.e. the behavioral policy). This (TD) method is known as *Q learning* and is described in procedural form in Figure 18.

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 $Q(\mathit{terminal}, \cdot) = 0$

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

Figure 18: Q-learning (TD) (off) policy control

Why don't we use importance sampling for then updating the action-values in Q-learning? After all, it is an off-policy method. It is because the agent is estimating action values with unknown policy. It does not need importance sampling ratios to correct for the difference in action selection. The action-value function represents the returns following each action in a given state. The agents target policy represents

the probability of taking each action in a given state. Putting these two elements together, the agent can calculate the expected return under its target policy from any given state, in particular, the next state, S_{t+1}. Q-learning uses exactly this technique to learn off-policy. Since the agents target policy is greedy with respect to its action values, all non-maximum actions have probability 0. As a result, the expected return from that state is equal to a maximal action value from that state.

Finally, note that Sarsa is a sample-based version of DP-based policy iteration as described in Chapter 4. Instead, Q-learning is a sample-based version of value iteration described in the same chapter.

Expected Sarsa

Now consider the learning algorithm that is just like Q-learning except that instead of the maximum over the next state-action pairs it uses the *expected* value, taking into account how likely each action is under the current policy:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}_\pi Q(S_{t+1}, A_{t+1}|S_{t_+1}) - Q(S_t, A_t)] \\ &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{46}$$

Given the next state S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *Expected Sarsa*. It is more complex computationally than Sarsa, but in return eliminates the variance due to the random selection of A_{t+1}.

In general, Expected Sarsa might use a policy different from the target policy π to generate behavior, and hence becomes an off-policy algorithm. For example, suppose that π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning. In this sense, Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa.

Chapter 8 - Planning and learning with tabular methods

We have now seen two types of RL methods: *model-based* methods (such as DP), and *model-free* methods (such as MC and TD). Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. In this chapter, we will integrate the two.

Models and planning

My a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. It the model is stochastic, then there are several possible next states and next rewards. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample* models. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it's much easier to obtain sample-models than distribution models.

Models can be used to mimic or simulate experience. Given a starting state and a policy, a sample model could produce and entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated* experience.

In this book, *planning* is used to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the model environment. We will in particular focus on *state-space planning*; this is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal.

All state-space planning methods share a common structure, which is common to what we have already seen in the learning methods discussed in previous chapters. This structure is based on two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience.

The heart of both planning and learning methods thus is the estimation of value functions by backing-up updated operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment.

This means that many ideas and algorithms can be transferred between planning. Specifically, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods only require experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. Figure 19 shows a simple example of a planning method based on one-step tabular Q-learning and on random samples for a sample model.

Random-sample one-step tabular Q-planning

Loop forever:

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

Figure 19: Q-planning

Dyna: Integrated planning, acting and learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. Dyna-Q is a simple architecture integrating the major functions needed in an online planning agent.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment), and it can be used to directly improve the value function and policy using the kinds of RL methods that we have discussed in previous chapters. The former is called *model-learning* or *indirect RL*, the latter is called *direct RL*.

Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affeceted by biases in the design of the model. The possible relationships between experience, model, values and policy are illustrated in Figure 20

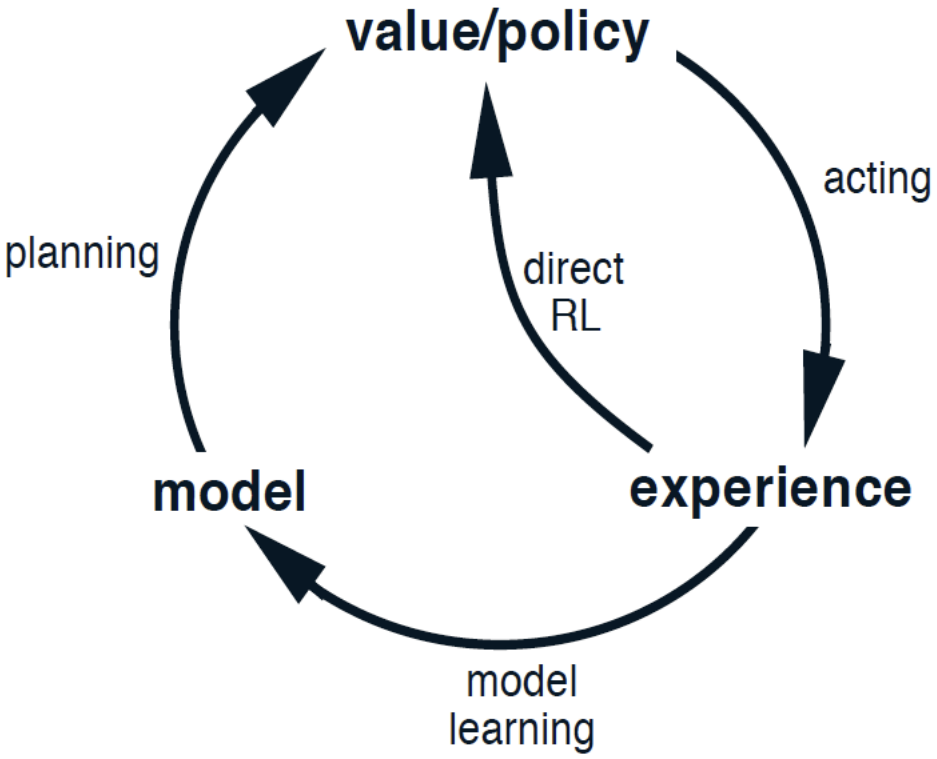

Figure 20: Schema of a planning agent

Dyna-Q includes all of the processes shown in Figure 20, all occurring continually. The planning method is the random-sample one-step tabular Q-planning method in Figure 19. The direct RL method is one-step tabular Q learning in Figure 18. The model-learning method is also table-based and assumes the environment is deterministic. After each transition $S_t, A_t \to R_{t+1}, S_{t+1}$ the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state-action pairs that have previously been experienced, so the model is never queried with a pair about which is has no information.

The overall architecture of Dyna agents – of which Dyna-Q is one example – is shown in Figure 21. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated

experiences generated by the model. Typically, as in Dyna-Q, the same RL method is used both for learning and planning. Learning and planning are deeply integrated in that they share almost all the same machinery, differing only in the source of their experience.

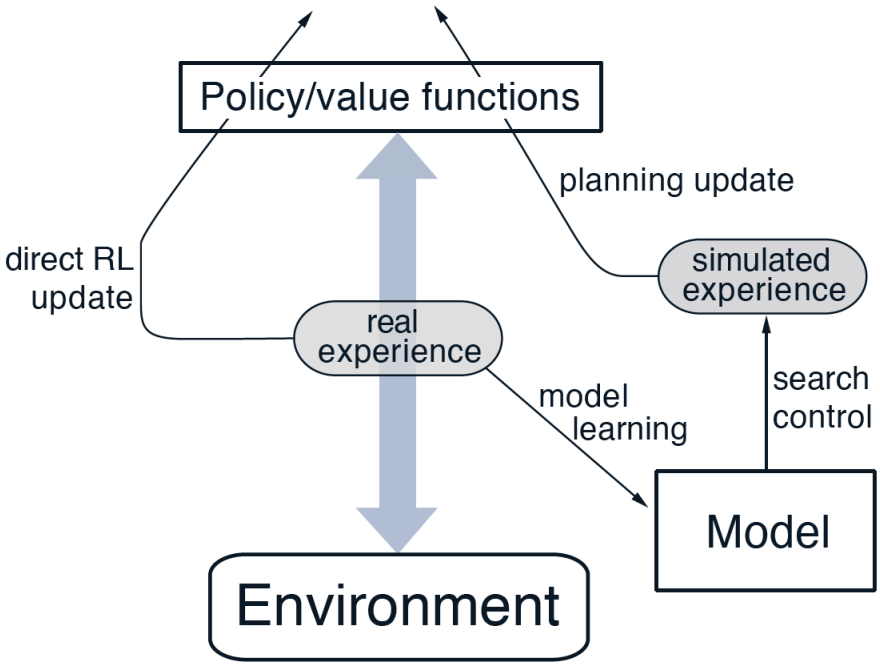

Figure 21: The general Dyna architecture

In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devote to the planning process, which is inherently computation-intensive. Figure 22 shows pseudo-code for one step of Dyna-Q. (Sub-)steps (d) and (e) are direct RL and model-learning respectively, whereas step (f) is planning, and is repeated n times (assuming this can be achieved in the remaining time in this step). Also note that if (e) and (f) were omitted, we are back to one-step tabular Q-learning from the previous chapter. In short, Dyna-Q makes better use of its limited interaction with the environment, after every episode, it runs n planning rounds from which it learns a better policy.

When the model is wrong

When the model is incorrect, the planing process is likely to compute a suboptimal policy. In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they don't exist.

Greater difficulties arise then the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these case the modeling error may not be detected for a long time, if ever. This is another version of the exploration-exploitation conflict. In a planning context, exploration means trying actions that improve the *model*, whereas exploitation means behaving in the optimal way given the current model. We want the agent to explore to find changes in the environment, but not so much that performance if greatly degraded.

The Dyna-Q+ agent uses an heuristic to solve this issue. This agent keeps track for each state-action pair of how many time steps have passed since the pair was last tried in real interaction with the environment. The

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

Figure 22: Tabular Dyna-Q

more time that has elapsed, the greater the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special bonus-reward is given on simulated experience involving these actions. In particular, if the modeled reward for a transition is r, and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$ for some small κ.

Note that taking this exploratory actions during *planning* will not yet help improve the accuracy of the model, since during planning we use a (so far inaccurate) model of the environment to obtain next states and rewards. However, as these make their way into the updated policy, eventually the agent *will* visit these exploratory states and take these exploratory actions, which then leads to an update of the model (either good or bad).

Chapter 9 - On-policy prediction with approximation

So far, we have treated approximate value functions as lookup tables. However, in many cases there are so many (sometimes even an infinite amount of) states that this approach becomes unfeasible. Instead, we are now going to represent value functions with a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s, w) \simeq v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$.

Typically, the number of weights is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. This *generalization* - i.e. the ability to apply knowledge of specific situations to draw conclusions about a wider variety of situations - makes the learning potentially more powerful but also potentially more difficult to manage and understand. In contrast to generalization stands *discrimination*, i.e. the ability to make the values of two (or more) states different.

Figure 23 plots different methods for value function computation in a generalization-discrimination framework. At the bottom-left are the tabular methods we have studied so far: they discriminate perfectly because every value update only affects one state. However, because of this, they offer no potential for generalization. At the opposite extreme (top-right) aggregate state methods, in which every update affects all states, leading to full generalization but no discrimination (not very useful). The ideal situation sits in the top-right (at *), but realistically we can typically only hope to achieve the "dot" combination.

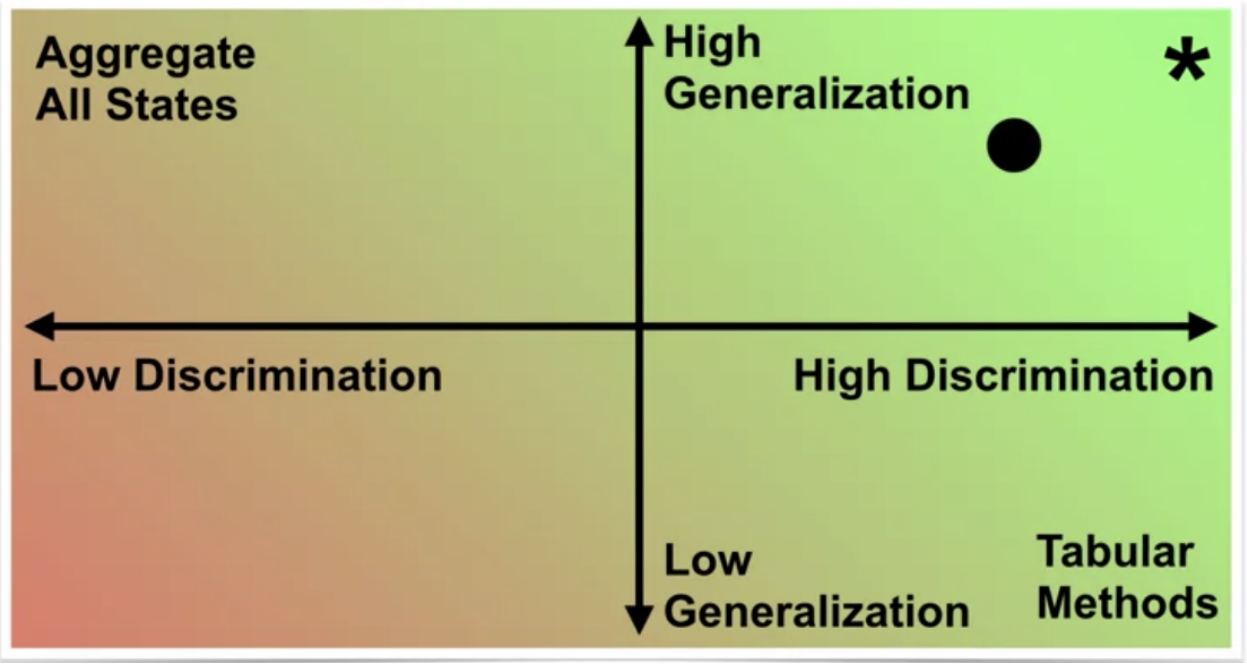

Figure 23: Generalization and discrimination

Value-function approximation

Let us refer to an individual update of a value to its target using the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted toward. It is natural to interpret each update as specifying an example (i.e. observation) of the desired input-output behavior of the value function. We permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. *Supervisedlearning* methods can be used to model this relationship, which is often called $functionapproximation$ in this context. These methods expect to receive examples of the desired input-output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of

each update as a training example. We then interpret the approximate function they produce as an estimated value function. That is, we view each update as a conventional training example, where the input is the state, and the output is the target.

In RL, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. They also require methods that are able to handle nonstationary target functions.

The prediction objective

In the earlier chapters, the learned value function could come to equal to true value function exactly. Moreover, the learned value at each state were decoupled - an update at one state affected no other. In the case of function approximation however, an update at one state affects many others, and it is not possible to get the values of all states exactly correct.

Therefore, we are obliged to say which states we care about most. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By this, we mean the square of the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $\upsilon_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, i.e. the *Mean Squared Value Error*:

$$\overline{VE}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[\upsilon_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2 \tag{47}$$

Often $\mu(s)$ is chosen to be the fraction of the time spent in s. Under on-policy training this is called the *on-policy distribtution*. In continuing tasks, the on-policy distribution is the stationary distribution under π.[7]

Stochastic-gradient and semi-gradient methods

Our objective is to minimize $\overline{VE}$ in (47). We will use gradient descent methods to do so. However, note that finding a (global) minimum of this function not necessarily corresponds to the true value function (i.e. it is typically not 0). It is limited by our choice of function parameterization, and depends on our choice of objective. Stochastic gradient descent (SDG) methods are particularly well-suited for online learning, because they update each time a new observation comes in.

The weight vector is a column vector with a fixed number of real-valued components $\mathbf{w} = (w_1, w_2, ..., wd)^T$ and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps $t = 0, 1, 2, ...$ and hence we will adjust the notation of the weight vector in each time step as $\mathbf{w}_t$.

Let's further assume that on each step, we observe a new examples $S_t \mapsto \upsilon_\pi(S_t)$. There is generally no $\mathbf{w}$ that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in the examples.

We assume that states appear in examples with the same distribution μ over which are are trying to minimize the $\overline{VE}$ as given by (47). This means we can drop the (often unknown) distritbution $\mu(s)$ from the objective function. A good strategy in this case is to try to minimize the error on the observed examples. SGD methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[\upsilon\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]^2 \\ &= \mathbf{w}_t - \alpha \left[\upsilon\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t) \end{aligned} \tag{48}$$

[7] In episodic tasks, it is the discretized version of this (i.e. the histogram of time spent in each state).

where α is a positive step-size parameter which controls how far we move in each step, and $\nabla f(\mathbf{w})$ the column vector of partial derivatives:

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, ..., \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^T \tag{49}$$

This derivative vector is the *gradient* of f with respect to $\mathbf{w}$. This gradient gives the the direction of steepest ascent in f when increasing w. SGD methods are gradient descent methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error. They are called stochastic when the update is done on only a single example.

Recall that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. The convergence results for SGD methods in this regard assume that α decreases over time.

An issue arises because we don't typically know the true value of the target $v_\pi(S_t)$. In the examples that we use for training, the target output is denoted by $U_t \in \mathbb{R}$, and is possibly some random approximation to the true value. In this case, we cannot perform the exact update in (48), but we can approximate it by substituting U_t for $v_\pi(S_t)$:

$$\mathbf{w}_{t+1} = \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{50}$$

Figure 24 gives pseudo-code when using MC methods to estimated U_t. Since this is by definition an unbiased estimate of $v_\pi(S_t)$, it is guaranteed to find a locally optimal solution.

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, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[G_t - \hat{v}(S_t, \mathbf{w}) \big] \nabla \hat{v}(S_t, \mathbf{w})$

Figure 24: Gradient Monte Carlo

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target. This is because bootstrapping targets all depend on the current value of the weight vector $\mathbf{w}_t$, which implies they will be biased and that they will not produce a true gradient-descent method.

Stated differently, when using bootstrapping methods to approximate the target value, the first term within brackets in (ref{eq:sgd_approx}) now also depends on $\mathbf{w}_t$. Applying gradient descent methods in this case incorrectly ignores the impact of changing the weights on the target. That is, they include only part of the gradient, which is why they are called *semi-gradient methods*.

Although they do not converge as robust as gradient methods, they do converge reliable in important cases, and they have other important advantages which often make them clearly preferred (e.g. enabling faster learning, and not having to wait for an episode to end). Figure 25 shows pseudo-code for implementing semi-gradient TD(0), which uses $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ as its target.

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

Figure 25: Semi-gradient TD(0)

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD in (50) in which the gradient $\nabla\hat{v}(S_t, \mathbf{w}_t)$ is 1 for S_t's group's component, and 0 for the other components.[8]

Feature construction for linear methods

Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate.

Polynomials

Various families of features commonly used for interpolation and regression can also be used in RL. Polynomials make up one of the simplest families o features used for interpolation and regressions. It is important that these polynomials can be nonlinear in features - they are part of linear methods when they are linear in the weights that have to be learned.

Higher-order polynomial bases allow for more accurate approximation of more complicated functions. But because the number of features in an order-n polynomial basis grows exponentially with the dimension k of the natural state space (if $n > 0$)[9], it is generally necessary to select a subset of them for function approximation.

[8] Alternatively, think of this as having one feature for each group of states. Each feature will be 1 if the current state belongs to the associated group, and 0 otherwise.

[9] Suppose each state s corresponds to k numbers $s_1, s_2, ..., s_k$ with each $s_i \in \mathbb{R}$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as $x(s)_i = \prod_{j=1}^{k} s_j^{c_{i,j}}$ where each $c_{i,j}$ is an integer in the set $\{0, 1, ..., n\}$ for an integer $n \geq 0$. These features make up the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features.

Coarse coding

Consider a task in which the natural representation of the state set is a continuous two-dimensional space. One kind of representation for this case is made up of features corresponding to *circles* in state space, as in Figure 26. If the state is inside the circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and said to be *absent*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

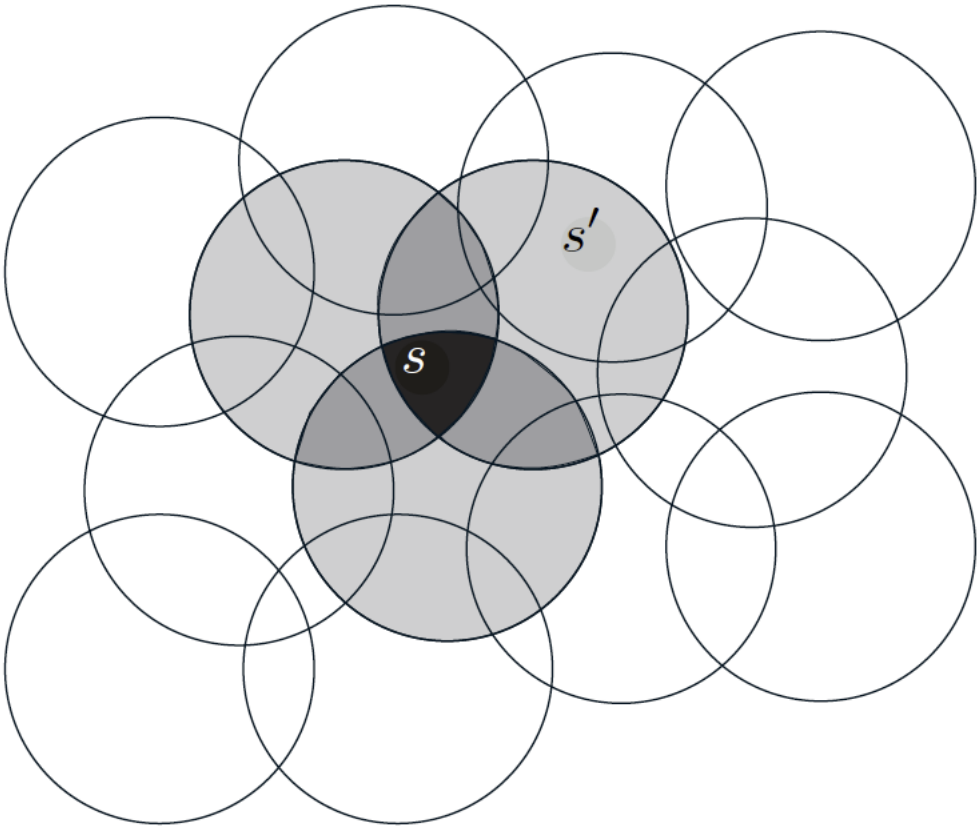

Figure 26: Coarse coding

Corresponding to each circle is a single weight (a component of $\mathbf{w}$) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersection that state will be affected. Thus, the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has "in common" with the state, as shown in Figure 26.

Note that the shapes need no be perfect circles as in Figure \ref{fig{coarse_coding_circles}, but could be elongated in one direction. Different shapes *and sizes* affect the receptive fields of the features. Initial generalization from one point to another is controlled by the size and the shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the number of features. The reason for this is that the extent of generalization is driven by the union of features, whereas discrimination by their intersection. After all, states that share an intersection will have the exact same feature representation. And, the more features we have, the smaller generally the intersection. Stated differently still, receptive field shape (and size) tends to have a strong effect on generalization but little effect on asymptotic solution quality.

Tile coding

In tile coding, the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. Figure 27 gives an example.

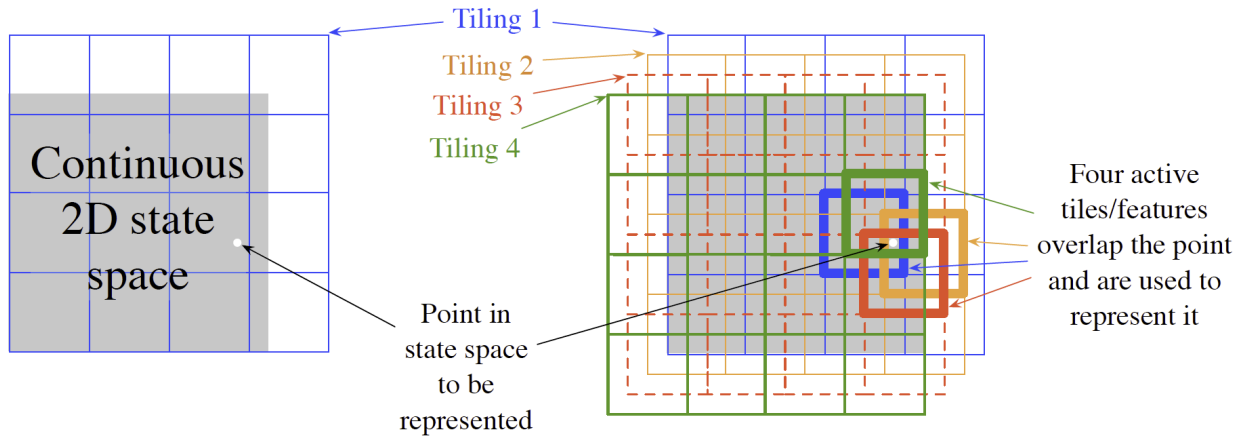

Figure 27: Tile coding

For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 27. The tiles or receptive fields here are squares rather than the circles in Figure 26. If just this single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. This is not coarse coding, but just a case of state aggregation.

To get the strengths of coarse coding requires overlapping receptive fields. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width, such as the one shown on the right in Figure 27. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the the number of tilings. This allows the step-size parameter α to be set in an easy, intuitive way.

For example, choosing $\alpha = \frac{1}{n}$, where n is the number of tilings, results in exact one-trial learning. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. In that case, one might for example choose $\alpha = \frac{1}{10n}$.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the filings from each other affects generalization. Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and n the number of tilings, then $\frac{w}{n}$ is a fundamental unit. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector (1,1), meaning that it is offset from the previous tiling by $\frac{w}{n}$ this vector.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. Another useful trick for reducing memory requirements is *hashing* - a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles.

Nonlinear function approximation: Artifical Neural Networks

Figure 28 gives a representation of an ANN with a four-unit input layer, two four-unit hidden layers, and a two-unit output layer.

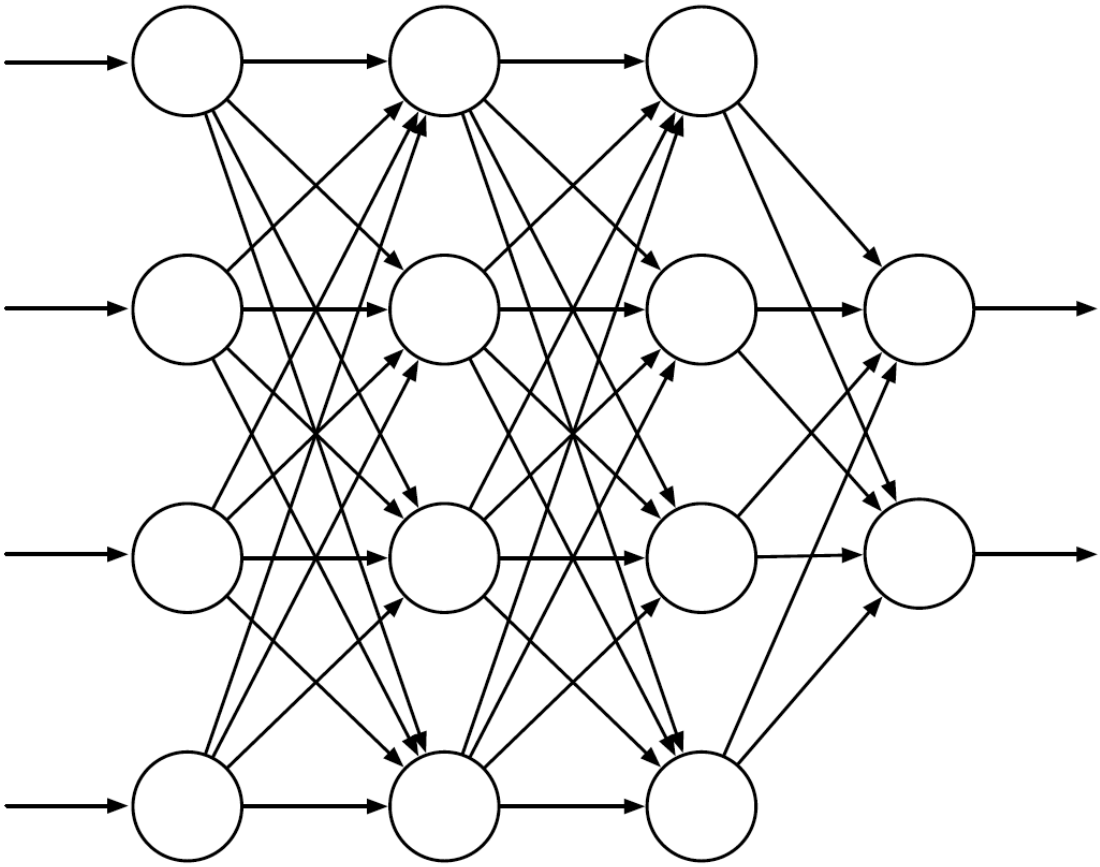

Figure 28: Feedforward ANN

The units (the circles in Figure 28) are often semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*. The units in the ANN's input layer are somewhat different in having their activations set to the externally supplied values that are the inputs to the function the ANN is approximating.

Despite the "universal approximation" property of one-hidden layer ANNs, both experience and theory show that approximating complex functions needed for many AI tasks is made easier – and may even require – abstractions that are hierarchical compositions of many layers of lower-level abstractions. The succesive layers of deep ANNs compute increasingly abstract representations of the network's raw input, with each unit providing a feaeture contributing to a hierarchical representation of the overall input-output function of the network.

For function approximation, it is necessary to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network's weights (the weights are represented by each link between to units in the network). The most successful way to do this for ANNs with hidden layers is the backpropagation algorithm, which consists of alternating forward and backward passes through the network.

However, this method may be more problematic for deep(er) ANNs. First, there is a problem of overfitting.

Second, the partial derivatives may either vanish with each backward pass, or explode. To tackle these issues, a host of solutions have been offered, such as dropout methods, using belief networks (where layers are trained one at a time, starting with the deepest), batch normalization, and deep residual learning.

A type of deep ANN that has proven to be very useful in applications, including RL, is the deep convolutional network. It is specialized for processing high-dimensional data arranged in spatial arrays, such as images. A CNN typically consists of alternating convolutional and subsampling layers, following by several fully connect layers (a regular ANN).

Each convolutional layer in a CNN produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it "sees" from the preceding layer. The subsampling layer reduces the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the fatures maps of the preceding convolutional layer. They reduce the network's sensitivity to the spatial locations of the features detected, that is, they help make the network's responses spatially invariant.

Chapter 10 - On-policy control with approximation

We now return to the control problem, in this case with parametric function approximation of the action-value function $\tilde{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector.

State-action feature construction

To move from TD (prediction) to Sarsa (control), we need action-value functions. So the feature representation has to represent actions as well. One way to do this is to have a separate function approximator for each action. This can be accomplished by so-called *feature stacking*. That is, we can use the same features for each action, but only activate the features corresponding to that action. This is illustrated in Figure 29.

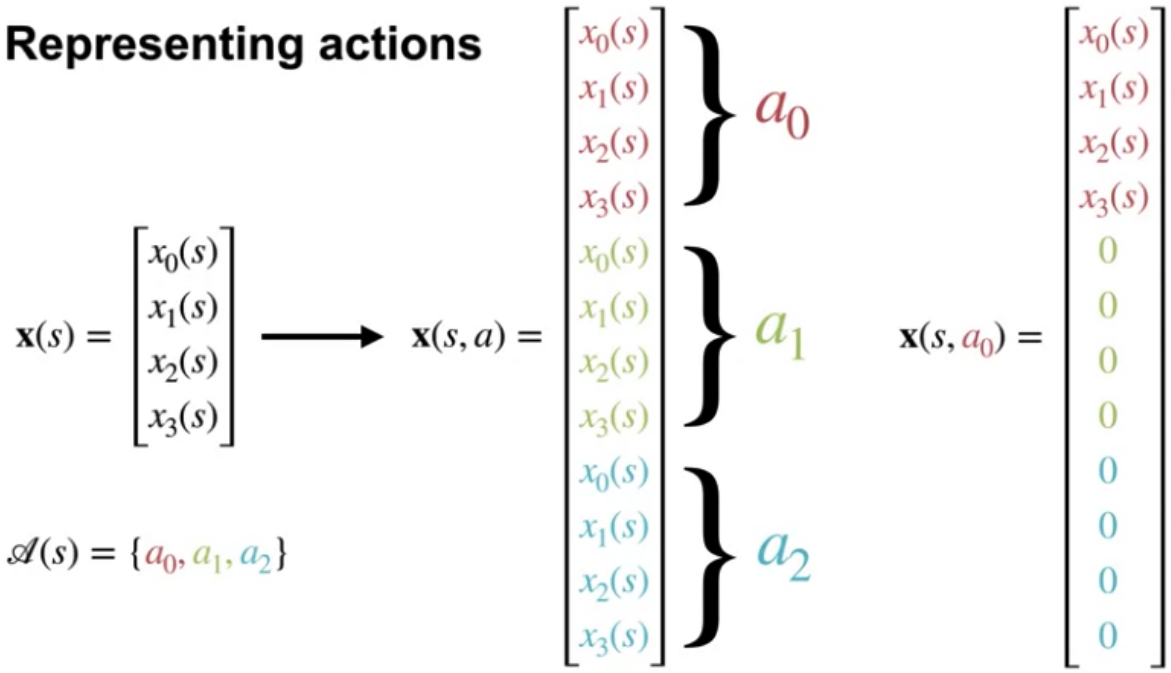

Figure 29: Feature stacking for state-action pairs

Let's say there are four features and three actions. The four features represent the state you are in, but we want to learn a function of both states and actions. We can do this by repeating the four features for each action. This results in a 12-component feature vector, with each segment of four features corresponding to one action. Thus, only the features for the specified action will be active, while those of the other actions will be set to 0.

Figure 30 gives an example of action-value computation in this case, where there features for states 0 and 3 are activated.

A similar action-value feature construction can also be achieved with NNs. The common way to represent action values with a NN is to generate multiple outputs, one for each action-value. The NN inputs are the states (as before), and the last hidden layer produces the state features. Each action-value is computed from an independent set of weights using those state features.

We might want to generalize over actions for the same reason generalizing over states can be useful. In the case of a NN, we would input *both* the states and the actions to the network, and there now would be only one output.

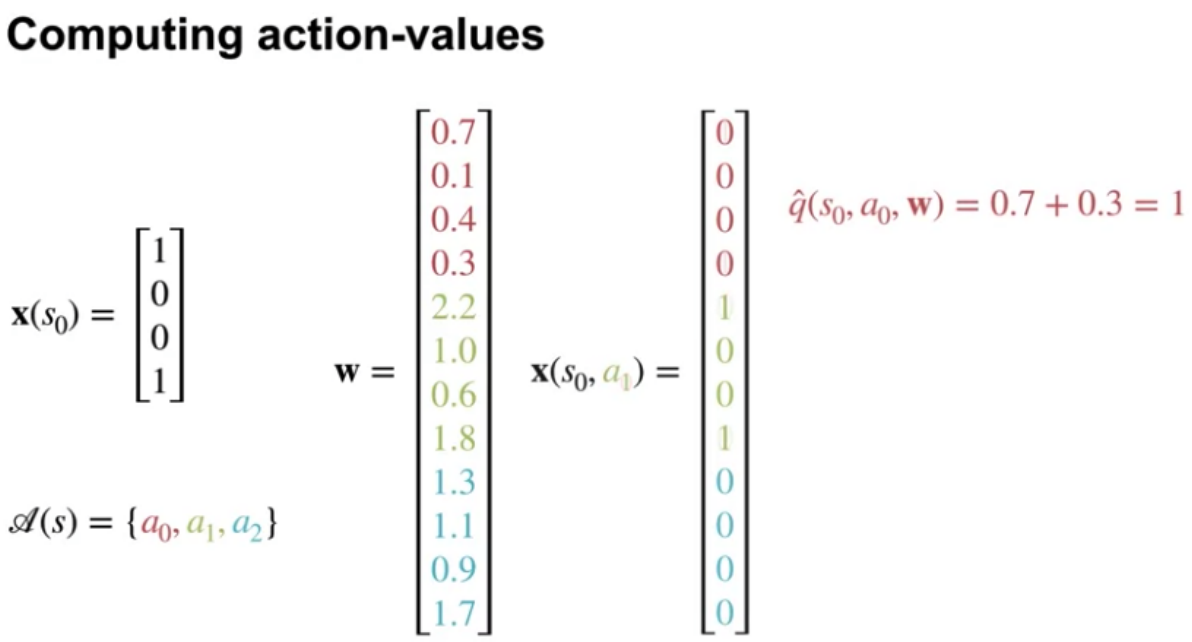

Figure 30: Action-value computation with feature stacking

Episodic semi-gradient control

Whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$. The general gradient-descent update for action-value prediction is:

$$\mathbf{w}_{t+1} = w_t + \alpha \left[U_t - \tilde{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \tilde{q}(S_t, A_t, \mathbf{w}_t) \tag{51}$$

For example, the update for the one-step Sarsa method is:

$$\mathbf{w}_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \tilde{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \tilde{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \tilde{q}(S_t, A_t, \mathbf{w}_t) \tag{52}$$

This method is called the *episodic semi-gradient one-step Sarsa*.

If the action set is discrete and not too large, we can use the techniques already developed in the previous chapters to form control methods. That is, for each possible action a available in the next state S_{t+1}, we can compute $\tilde{q}(S_{t+1}, a, \mathbf{w}_t)$ and then find the greede action $A^*_{t+1} = \text{argmax}_a \tilde{q}(S_{t+1}, a, mathbf w_t)$. Figure 31 show pseudo-code for this algorithm.

We can easily extend the update in (52) to expected Sarsa as well:

$$\mathbf{w}_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \sum_a^{'} \pi(a'|S_{t+1}) \tilde{q}(S_{t+1}, a', \mathbf{w}_t) - \tilde{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \tilde{q}(S_t, A_t, \mathbf{w}_t) \tag{53}$$

And, since Q-learning is a special case of expected Sarsa (see Chapter 6), the update for Q-learning is:

$$\mathbf{w}_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \max_a{}^{'} \tilde{q}(S_{t+1}, a', \mathbf{w}_t) - \tilde{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \tilde{q}(S_t, A_t, \mathbf{w}_t) \tag{54}$$

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)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Figure 31: Semi-gradient sarsa

Exploration vs exploitation with function approximation

As we discussed before, two ways of achieving exploration are choosing optimistic initial values, and relying on ϵ-greedy policies. The former is a more systematic way of exploration, whereas the latter relies on randomness alone.

In the tabular case, setting optimistic initial values happens via initializing the state-action values. With function approximation, we instead do so via initializing the weight vector(s). However, optimistic initial values are problematic in a function approximation setting vs a tabular setting, because in the former, updates generalize (to some degree) across multiple states. This means that state-action values of states that have not yet been visited may already become more pessimistic due to generalization.

To facilitate systematic exploration with this approach, changes to the value function need to be more localized. For example, function approximation with tile coding can produce such localized updates. In contrast, ϵ-greede is generally applicable and easy to use with function approximation. The drawback is that it relies on randomness for exploration, and hence is not a direct way of exploration.

Average reward

Like the discounted setting, the *average reward* setting applies to continuing problems, for which the interaction between agent and environment goes on and on forever without termination or start states. Unlike that setting, however, there is no discounting. For reasons discussed later, the discounted setting is problematic with function approximation, and thus the average-reward setting is needed to replace it.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply the *average reward*, while following that policy, we is denoted as $r(\pi)$:

$$
\begin{aligned}
r(\pi) &= \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \\
&= \lim_{t\to\infty} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r
\end{aligned} \tag{55}
$$

where the expectations are conditioned on the initial state S_0, and on the subsequent actions $A_0, A_1, ..., A_{t-1}$ being taken according to π. The second and third equations hold if the MDP is *ergodic*. In an ergodic MDP, the starting state and any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities.

For most practical purposes, it may be adequate to simply order policies according to their average reward per time step, in other words, according to their $r(\pi)$. We consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Note that the steady state distribution μ_π is the special distribution under which, if you select actions according to π, you remain in the same distribution. That is, for which:

$$
\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s'|s, a) = \mu_p i(s') \tag{56}
$$

In the average-reward setting, returns are defined in terms of differences between rewards and the average reward:

$$
G_t = R_{t+1} - r(\pi) + R_{t+1} - r(\pi) + ... \tag{57}
$$

This is known as the *differential return*, and the corresponding value functions are known as *differential* value functions. Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all the γs and replace all rewards by the difference between reward and the true average reward. Similarly, there is also a differential form for the two TD errors (one for state-value and the other for state-action value).

With these alternate definitions, most of our algorithms and many theoretical results carry through to the average-reward setting without change. For example, and average reward version of semi-gradient Sarsa could be defined just as in (52), execpt with the diffferential version of the TD error: er

$$
\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[R_{t+1} - \overline{R}_t + \tilde{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \tilde{q}(S_t, A_t, \mathbf{w}_t)] \nabla \tilde{q}(S_t, A_t, \mathbf{w}_t) \tag{58}
$$

Figure 32 gives pseudo-code to a complete algorithm.

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)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Figure 32: Differential semi-gradient sarsa

Chapter 13 - Policy gradient methods

Rather than learning action-values in order to select actions, we can also learn a *parameterized policy* directly. This way, we can select actions without consulting a value function. Taking such an approach has a number of potential advantages. First, the (learned) approximate policy can approach a deterministic policy naturally and autonomously, something that ϵ-greedy action selection does not do, as it keeps choosing non-optimal policies to explore. Second, and in contrast to the this, policy parameterization allows action selection with arbitrary probabilities. That means that if a stochastic policy is optimal, policy gradient methods are more likely to learn them naturally. Figure 34 further below illustrates these two advantages. Third, the parameterized policy may be a simpler function to approximate than the action-value action function. Finally, it is often a useful way to inject prior knowledge about the desired form of the policy into the RL system.

We will use the notation $\theta \in \mathbb{R}^d$ for the policy's parameter vector. Hence, we write $\pi(a, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\}$ for the probability that action a is taken at time t in state s with parameter θ.

We consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\theta)$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{59}$$

where $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ_t. All methods that follow this general schema are called *policy gradient methods*. Methods that learn approximations to *both* policy and value functions are often called *actor-critic methods*, where *actor* is a reference to the learned policy, and *critic* to the learned value function.

The policy can be parameterized in any way, as long as $\pi(a|s, \theta_t)$ is differentiable with respect to its parameters. Further, the parameterization has to satsisfy the regular conditions of a probability distribution, i.e. $\pi(a|s, \theta_t) \in [0, 1]$ for all s, a, and θ, and $\sum_a \pi(a|s, \theta) = 1$.[10] This means that not all parameterizations of

[10] In practice, to ensure exploration, we generally require that $\pi(a|s, \theta_t) \in (0, 1)$

π are equally qualified, because they do not satisfy these conditions. For example, a linear approximation is not guaranteed to satisfy these conditions.

If we write the general parameterization of numerical preferences for state-action pairs as $h(s, a, \theta) \in \mathbb{R}$, we can apply non-linear transformation to $h(.)$ to satisfy these conditions, e.g. the soft-max distribution:

$$\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \tag{60}$$

The action preferences themselves can then be parameterized arbitrarily, e.g. by a linear function, or an ANN. Figure 33 shows how the action preferences are transformed by the softmax function. Even for negative preferences, the softmax will yield a small but non-zero probability.

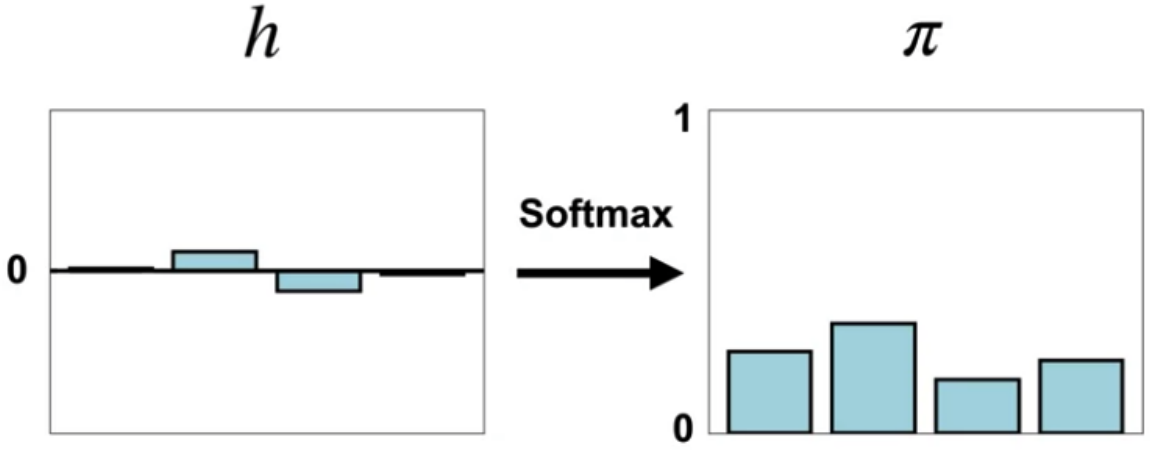

Figure 33: Softmax transformation

It is important not to confuse action preferences with action values. Action preferences indicate how much the agent prefers each action, but they are not summaries of future reward. Only the relative differences between preferences are important.

An ϵ-greedy policy derived from action-values can behave very differently from a softmax policy over action preferences. In ϵ-greedy, the action corresponding to the highest valued action is selected with high probability. The probability selecting all the other actions is quite small. Actions with nearly the same but lower action values are selected with much lower probability. On the other hand, actions with very poor action values are still selected frequently due to the epsilon exploration step. So even if the agent learns an action has terrible consequences, it will continue to select that action much more frequently than it would under the softmax policy. Figure 34 illustrates these differences graphically.

With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ϵ-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action value.

The policy gradient theorem

Recall that the ultimate goal of RL is to learn a policy that obtains as much reward as possible in the long run. When we parameterize our policy directly, we can also use this goal directly as the learning objective. In the past chapters, we have always distinguished between the episodic and the continuing case.

For the episodic case, we have written the objective as follows:

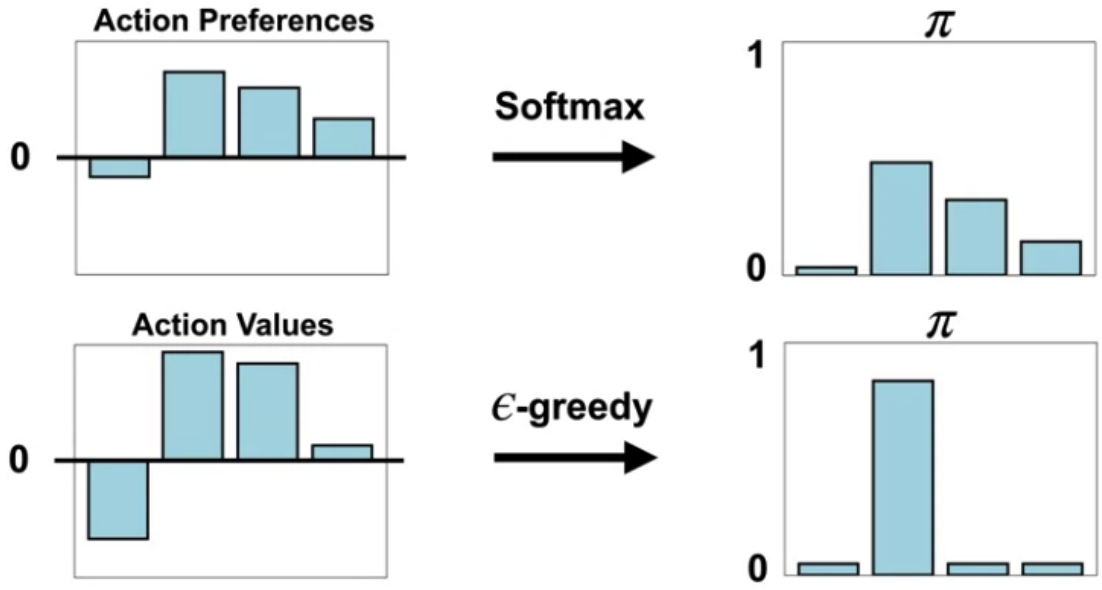

Figure 34: Softmax vs ϵ-greedy

$$G_t = \sum_{t=0}^{T} R_t \tag{61}$$

whereas for the continuous case, we have added discounting to prevent the infinite sum from exploding:

$$G_t = \sum_{t=0}^{T} \gamma^t R_t \tag{62}$$

In chapter 10, we introduced yet another way to write returns in the continuous case, using the average reward notation:

$$G_t = \sum_{t=0}^{T} R_t - r(\pi) \tag{63}$$

Let's now write the average reward objective in a form that we can optimize:

$$r(\pi) = \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r \tag{64}$$

where the last sum is the expected reward when starting in state s and choosing action a, $\mathbb{E}[R_t|S_t = s, A_t = a]$. The last two sums is the expected reward over all possible actions when starting in state s, $\mathbb{E}_\pi[R_t|S_t = s]$. Finally, all three sums give us the average reward, by also considering the fraction of time we spent in state s under policy π which is captured by $\mu(s)$, giving $\mathbb{E}_\pi[\mathbb{R}_\approx]$.

The goal is to learn that parameters θ that maximize this average reward objective. That is:

$$\nabla_\theta r(\pi) = \nabla_\theta \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r \tag{65}$$

As mentioned before, methods that rely on this approach are called policy-gradient methods. It is important to differentiate this *direct* approach of policy learning (i.e. control) with the *indirect* approach of GPI that we have discussed so far. In GPI, we first learn approximate action values. These are then used to infer a good policy. With policy-gradient methods we are learning the policy directly, without the back and forth between estimation and control.

One challenge that arises when estimating the gradient in (65) is that $\mu(s)$ changes with the policy, i.e. with changes in θ. This problem was not encountered in the GPI framework, since gradient descent was learning parameters $\mathbf{w}$ under a *given* policy.

Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*. It provides an analytical expression for the gradient in (65) that does **not** involve the derivative of the state distribution:

$$\nabla_\theta r(\pi) = \sum_s \mu(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a) \tag{66}$$

where $q_\pi(s,a)$ is the familiar approximate value of choosing action a in state s.

Estimating the policy gradient

The policy gradient theorem (66) gives an exact expression for the gradient. All that is needed is some way of sampling whose expectation equals or approximates this expression. Rather than summing over states, we simply make updates from states we observe when following policy π:

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(S_t, a, \mathbf{w}) \nabla \pi(a|S_t,\theta) \tag{67}$$

This yields an unbiased estimate of (66) because the sum over $\mu(s)$ is the same as the expectation of following policy π (i.e. $\mathbb{E}_\pi$). We can further simplify this expression by also getting rid of the sum over actions. To achieve this, it would be nice if the sum over a was weighted by π and so was an expectation under π. This is easily fixed by multiplying and dividing by $\pi(a|S,\theta)$ simultaneously:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha \sum_a \pi(a|S,\theta) \frac{1}{\pi(a|S,\theta)} q_\pi(S_t, a, \mathbf{w}) \nabla \pi(a|S_t,\theta) \\ &= \theta_t + \alpha \mathbb{E}_\pi \left[\frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S,\theta)} q_\pi(S_t, a, \mathbf{w}) \right] \end{aligned} \tag{68}$$

Using a sample A_t to instantiate the stochastic gradient ascent algorithm yields the following update:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha \left[\frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S,\theta)} q_\pi(S_t, A_t, \mathbf{w}) \right] \\ &= \theta_t + \alpha \nabla \ln \pi(A_t|S_t,\theta) q_\pi(S_t, A_t, \mathbf{w}) \end{aligned} \tag{69}$$

Actor-critic methods

Even within policy gradient methods, value-learning methods like TD still have an important role to play. In this setup, the parameterized policy plays the role of an *actor*, while the value function plays the role of a *critic*, evaluating actions selected by the actor.

Consider again the stochastic gradient ascent policy parameter update in (69). Since we don't have access to q_π, so we have to approximate it, e.g. by applying (differential reward) TD:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(A_t|S_t, \theta)[R_{t+1} - \bar{R} + \hat{\nu}(S_{t+1}, \mathbf{w})] \tag{70}$$

with $\hat{\nu}$ being the differential value function. This is the critic part of the algorithm, which can be approximated by any state value learning algorithm, like semi-gradient TD.

There is one more thing we can do to improve the algorithm, which is to subtract a baseline value estimate $\hat{\nu}(S_t, \mathbf{w})$:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(A_t|S_t, \theta)[R_{t+1} - \bar{R} + \hat{\nu}(S_{t+1}, \mathbf{w}) - \hat{\nu}(S_t, \mathbf{w})] \tag{71}$$

Notice that the last expression in the square brackets is equal to the TD error δ:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(A_t|S_t, \theta)\delta \tag{72}$$

Why do we subtract this baseline? Because it tends to reduce the variance of the upate, and hence results in faster learning.

Figure 35 illustrates how the actor and critic interaction based on (72). After we execute an action, we use the TD error to decide how good the action was compared to the average for that state. If the TD error is positive, then it means the selected action resulted in a higher value than expected. Taking that action more often should improve our policy. That is exactly what this update does. It changes the policy parameters to increase the probability of actions that were better than expected according to the critic. Correspondingly, if the critic is disappointed and the TD error is negative, then the probability of the action is decreased. The actor and the critic learn at the same time, constantly interacting. The actor is continually changing the policy to exceed the critics expectation, and the critic is constantly updating its value function to evaluate the actors changing policy.

Figure 36 gives pseudo-code for this control algorithm.

Gaussian policies for continuous actions

Policy-based methods offer practical ways of dealing with large action spaces, even continuous spaces with an infinite number of actions. Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution.

For example, the action set might be the real numbers, with actions chosen from a normal (Gaussian) distribution:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{73}$$

with μ and σ the mean and standard deviation of the normal distribution. To produce a policy parameterization, the policy can be defined as the normal probability density over a real-value scalar action, with mean and standard deviation given by parametric function approximators that depend on the state:

$$p(x) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left(-\frac{(x-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right) \tag{74}$$

We divide the policy's parameter vector into two parts, $\theta = [\theta_\mu, \theta_\sigma]^T$. The mean can be approximated as a linear function, the standard deviation must always be positive and is better approximated as the exponential of a linear function:

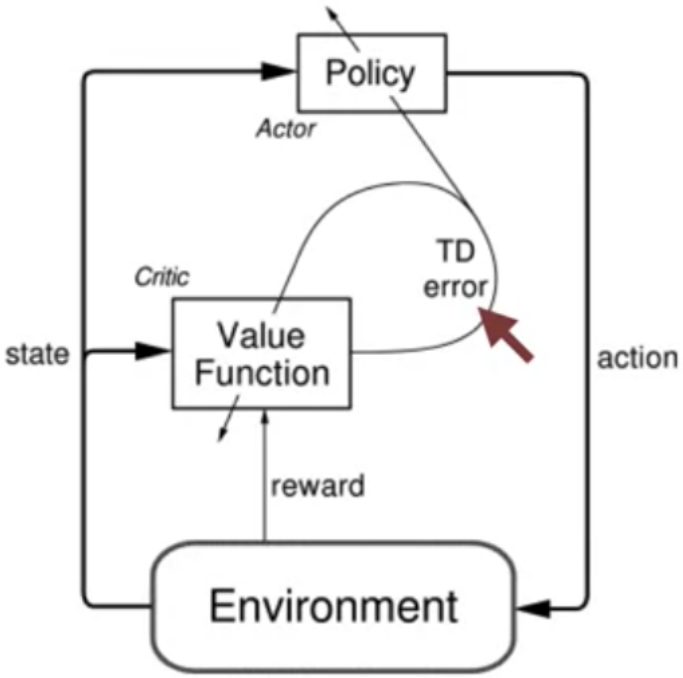

Figure 35: Actor-critic interaction

$$\mu(s,\theta) = \theta_\mu^T \mathbf{x}_\mu(s) \text{ and } \sigma(s,\theta_\sigma) = \exp\left(\theta_\sigma^T \mathbf{x}_\sigma(s)\right) \tag{75}$$

With these definitions, all the algorithms discussed previously can be applied to learn to select real-value actions. Note that σ controls the degree of exploration, and therefore we typically initiliaze it a large values. We expect it to shrink over the course of learning.

Actor-Critic (continuing), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Initialize $\bar{R} \in \mathbb{R}$ to 0

Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g. to 0)

Algorithm parameters: $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$

Initialize $S \in \mathcal{S}$

Loop forever (for each time step):

$A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$

Take action A, observe S', R

$\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$

$S \leftarrow S'$

Figure 36: Actor-critic algorithm