Reinforcement Learning NOTES

Kristian Bonnici

October 7, 2021

Contents

0.1 Summary of Notation

$\doteq$ is used for **"is defined as"**

S = set of **nonterminal states** * $s, s' \rightarrow$ some states

S^+ = set of **all states**, including the terminal state

A = set of **actions**

R = set of **rewards**

We'll assume that each of these sets will have a *finite number of elements.*

$T \rightarrow$ transition function

$V, V_t \rightarrow$ array **estimates** of $v_\pi(s)$ or $v_*(s)$

$v_*(s) \rightarrow$ value of state s under the optimal policy

- $= \max_\pi v_\pi(s)$

$Q, Q_t \rightarrow$ array **estimates** of $q_\pi(s, a)$ or $q_*(s, a)$

Return G: is the total of rewards

- **G for Episodic Tasks:** The return at time step t is the sum of rewards until termination.

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

- **G for Continuing Tasks:** sum of discounted future rewards (made to be always finite).

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + R_{t+k} + ... \\ &\doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \end{aligned}$$

  - **(recursively):**

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

- Where:
  - $\gamma =$ is a parameter, $0 \leq \gamma \leq 1$, called the **discount rate**.
    - If $\gamma = 1$, undiscounted.

Value functions:

- The current time-step's state/action values can be written **recursively** in terms of **future state/action values** with **Bellman equations**.

- **State-Value functions:** expected return G from a given state s, following policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

  - **(recursively):**

$$v_\pi(s) = \mathbb{E}_\pi[\underbrace{R_{t+1}}_{\text{immediate reward}} + \underbrace{\gamma G_{t+1}}_{\text{discounted return at time } t+1} | S_t = s]$$

  - **(State-Value Bellman equation):**

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']] \\ &= \underbrace{\sum_a \pi(a|s)}_{\text{sum over possible action choices}} \underbrace{\sum_{s'} \sum_r p(s', r | s, a)}_{\text{sum over possible rewards and next states}} [r + \gamma v_\pi(s')] \end{aligned}$$

    - The result is a weighted sum of terms consisting of immediate reward plus expected future returns from the next state s'.

  - **(State-Value Bellman Optimality equation):**

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

  - The magic of value functions is that we can use them as a stand-in for the average of an infinite number of possible futures.

- **Action-Value functions:** expected return G if the agent selects action a in state s and then follows policy π thereafter.

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

  - **(Action-Value Bellman equation):**

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \sum_{s'}\sum_r p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s')\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a']] \\ &= \sum_{s'}\sum_r p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')] \end{aligned}$$

  - **(Action-Value Bellman Optimality equation):**

$$q_*(s,a) = \sum_{s'}\sum_r p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

Policy π: mapping from state to action (decision-making rule)

- **Deterministic policy notation:** $\pi(s) = a$
- **Stochastic policy notation:** $\pi(a|s)$
- **Optimal Policy** π_*

$$\text{for } q_*(s,a) \to \pi_*(s) = \arg\max_a q_*(s,a)$$

$$\begin{aligned} \text{for } v_*(s) \to \ \pi_*(s) &= \arg\max_a E_{s'}[r(s,a,s') + \gamma v_*(s')] \\ &= \arg\max_a \sum_{s'} T(s,a,s')(r(s,a,s') + \gamma v_*(s')) \end{aligned}$$

  - With Bellman equation (compare to State/Action-Value Bellman Optimality equation):

$$\pi_*(s) = \arg\max_a \sum_{s'}\sum_r p(s',r|s,a)[r + \gamma v_*(s')]$$

$$\pi_*(s) = \arg\max_a q_*(s,a)$$

- **Value iteration:**, for estimating $\pi \approx \pi_* \to$ `converges to` $v_*(s)$. Only one iteration of iterative policy evaluation is performed between each step of policy improvement.

  - Starting from $V_0^*(s) = 0$ for all ($\forall$) $s \rightarrow$ iterate until convergence (usually change being smaller than some threshold we choose):
    * $V_{i+1}^*(s) = \max_a \sum_{s'} T(s,a,s')(r(s,a,s') + \gamma V_i^*(s'))$
    * $\quad\quad\quad = \max_a ImmediateReward + Discount * FutureRewards$

- **Policy iteration (iterative policy evaluation):**, for estimating $\pi \approx \pi_*$.

1 Introduction to Reinforcement Learning

Reinforcement learning an area of machine learning concerned with how intelligent **agents** ought to take **actions** in an **environment** in order to maximize the notion of cumulative **reward**.

The **environment** is typically stated in the form of a Markov decision process (MDP), because many reinforcement learning algorithms for this context use dynamic programming techniques.

Exploration & Exploitation trade-off: Dilemma: choosing when to explore & when to exploit?

- **Exploration:** improve knowledge for long-term benefit.
- **Exploitation:** exploit knowledge for short-term benefit.

Four main subelements of a reinforcement learning system: a **policy**, a **reward signal**, a **value function**, and, optionally, a **model** of the environment.

- **Policy** defines the learning agent's way of behaving at a given time.
- **Reward signal** indicates what is good in an immediate sense.
- **Value function** specifies what is good in the long run. Roughly speaking, the value of a state is the *total amount of reward* an agent can expect to accumulate over the future, starting from that state.
- **Model** mimics the behaviour of the environment, or more generally, that allows inferences to be made about how the environment will behave.
  - **Used for planning**, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced.

Model-based vs Model-free RL:

- If model → **model-based** methods
- If no model → simpler **model-free** methods
  - explicitly trial-and-error learners—viewed as almost the opposite of planning.
- → **Modern reinforcement learning** spans the spectrum **from** low-level, trial-and-error learning **to** high-level, deliberative planning.

Evolutionary methods: (not focused on this course)

- Instead of estimating value functions, these methods apply multiple static policies each interacting over an extended period of time with a separate instance of the environment. The policies that obtain the most reward, and random variations of them, are carried over to the next generation of policies, and the process repeats.
- Although evolution and learning share many features and naturally work together, we do not consider evolutionary methods by themselves to be especially well suited to reinforcement learning problems and, accordingly, we do not cover them in this book.

Part I

Tabular Solution methods

In this part we describe almost all the core ideas of reinforcement learning algorithms in their **simplest forms:** $\rightarrow$ State & action spaces are small enough for the **approximate value functions** to be represented as arrays, or tables.

- Often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy

This **contrasts with** the *approximate methods* described in the next part, which only find approximate solutions, but which in return can be applied effectively to much larger problems.

Chapters in this section:

- 2. **Multi-armed Bandits:** special case of the reinforcement learning problem in which there is only a single state.
- 3. **Finite Markov Decision Processes:** the general problem formulation that we treat throughout the rest of the notes.
    - Its main ideas including **Bellman equations** and **value functions**.
- The next three chapters (4., 5. & 6.) describe **three fundamental classes of methods for solving finite Markov decision problems:**

  4. **Dynamic Programming**
     - \+ Well developed mathematically
     - **-** Require a complete and accurate model of the environment
  5. **Monte Carlo Methods**
     - \+ Don't require a model
     - \+ Conceptually simple
     - **-** Not well suited for step-by-step incremental computation

6. **Temporal-Difference Learning**
   - + Don't require a model
   - + Fully incremental
   - **-** More complex to analyze

The methods also differ in several ways with respect to their efficiency and speed of convergence.

- The remaining two chapters (7. & 8.) **describe how these three classes of methods can be combined** to obtain the best features of each of them.

7. *n*-step Bootstrapping: Strengths of Monte Carlo methods can be **combined** with the strengths of temporal-difference methods via multi-step bootstrapping methods.
8. **Planning and Learning with Tabular Methods:** temporal-difference learning methods can be **combined with** model learning and planning methods (such as dynamic programming) for a **complete and unified solution to the tabular reinforcement learning problem**.

---

2 Multi-armed Bandits

The most **important feature distinguishing reinforcement learning** from other types of learning is that it uses training information that *evaluates* the actions taken rather than instructs by giving correct actions. $\rightarrow$ need for active exploration, for an explicit search for good behaviour.

In this chapter we **study** the evaluative aspect of reinforcement learning **in** a simplified setting, one that does not involve learning to act in more than one situation.

Studying this case enables us to **see** most clearly **how** evaluative feedback **differs from**, and yet **can be combined with**, instructive feedback.

2.1 A k-armed Bandit Problem

Problem statement:

1. Being faced repeatedly with a choice among k different options/actions
2. Receive numerical reward chosen from a stationary probability distribution that depends on the action you selected
   - Each of the k actions has an expected (mean) reward $\rightarrow$ **value** of that action:
     - $q_*(a) = E[R_t|A_t = a]$
   - We denote the **estimated value** as:
     - $Q_t(a)$

- We assume that you don't know the **action values** with certainty, although you may have estimates. Otherwise it would be trivial to solve the k-armed bandit problem: you would always select the action with highest value

Objective: maximize the expected total reward over some time period.

Exploiting vs Exploring

- **Exploiting** your current knowledge:
  - Selecting action whose estimated value is greatest (*greedy* action).
    * **Goal:** maximize the expected reward on the one step
- **Exploring** to improve your estimate of the nongreedy action's value:
  - Not selecting *greedy* action
  - **Goal:** to produce the greater total reward in the long run
- There are many sophisticated methods for balancing **exploration** and **exploitation** for particular mathematical formulations of the k-armed bandit and related problems.
  - most of these methods make strong assumptions about stationarity and prior knowledge that are **either** violated or impossible to verify in applications.

2.2 Action-value Methods

What: methods for (1) estimating the **values** of actions and for using the estimates to (2) make **action** selection decisions.

(1) estimating the **values**

- **ONE natural WAY** for **estimating the true value of an action** is by averaging the rewards actually received **(*sample-average* method)**:

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

Where:

$1_{predicate}$: denotes the random variable that is

1 if predicate is true and

0 if it is not.

- If the denominator:
  - $= 0$, then we instead define $Q_t(a)$ as some default value (e.g. zero).
  - $\to \infty$, then by the **law of large numbers,** $Q_t(a)$ **converges to** $q_*(a)$.
- **NOTE:** Q_n can be **computed in** a computationally efficient manner, in particular, with constant memory and constant per-time-step computation as:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$
$$NewEstimate = OldEstimate + StepSize[Target - OldEstimate]$$

(2) make **action** selection decisions

- The **simplest action selection rule**, is always selecting one of the actions with the highest estimated value **(*greedy* action selection method)**:

$$A_t = \arg\max_a Q_t(a)$$

- A **simplest alternative action selection rule**, is to behave greedily most of the time, but every once in a while, say with small probability ϵ, instead select action randomly **(ϵ-greedy methods)**.

2.3 Bandit Algorithm Examples

Example: 10-bandit problems

- Given:

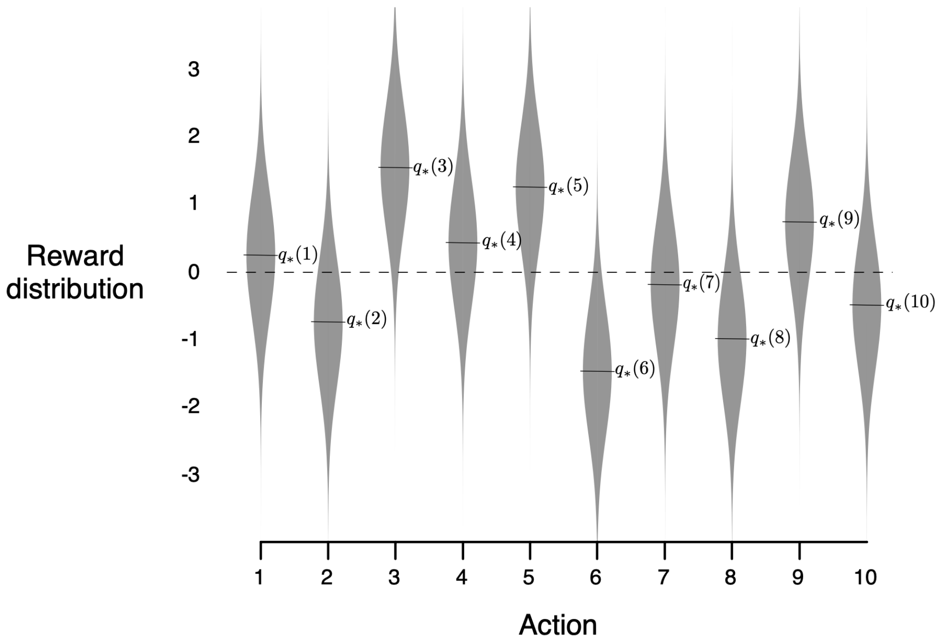

Figure 1: 10-armed testbed

- **Figure 1:** An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$ unit variance normal distribution, as suggested by these gray distributions.

- Then compares a greedy method with two ϵ-greedy methods ($\epsilon = 0.01$ and $\epsilon = 0.1$):

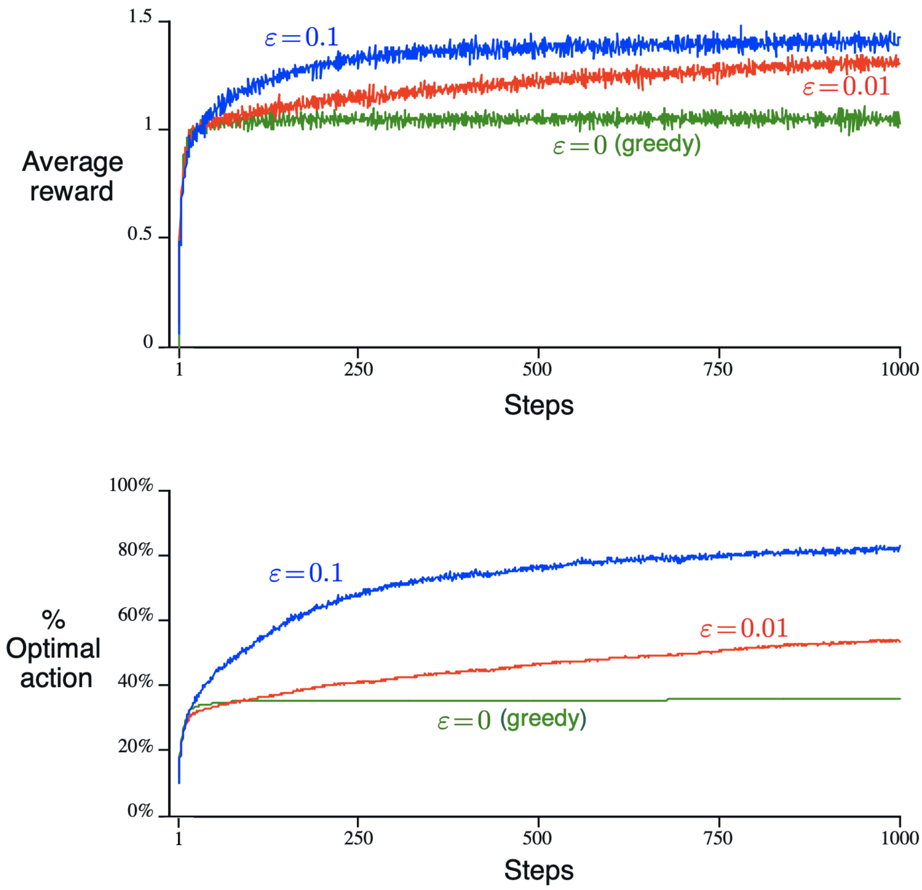

Figure 2: Average performance of ϵ-greedy action-value methods on the 10-armed testbed

- **Figure 2:** Average performance of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

- NOTE:

  - The $\epsilon = 0.01$ method improved more slowly, but eventually would perform better than the $\epsilon = 0.1$ method on both performance measures shown in the figure.
  - It is **also possible** to reduce ϵ over time to try to get the best of both high and low ϵ values.

– With **noisier rewards** it takes more exploration to find the optimal action.

Exploration is beneficial even in the deterministic worlds **if:**

- **nonstationary** task, that is, the **true values of the actions changed over time** → **agent's decision-making policy changes**.
- **nonstationary** is the case most commonly encountered in reinforcement learning.

Example: A simple bandit algorithm

Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ϵ-greedy action selection is shown in the box below.

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

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \texttt{with probability } 1-\epsilon \;\; \texttt{(breaking ties randomly)} \\ \texttt{random action} & \texttt{with probability } \epsilon \end{cases}$$

$$R \leftarrow bandit(A)$$

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

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

- Where:
  – **function** $bandit(a)$ is assumed to **take** an action and **return** a corresponding reward

2.4 Tracking a Nonstationary Problem

What: true values of the actions changed over time → agent's decision-making policy changes.

Adjustments: it makes sense to give more weight to recent rewards than to long-past rewards.

- A **POPULAR WAY** it to use a **constant step-size parameter** α (2.3 modified to be):

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

Where:

$\alpha \in (0, 1]$: is constant

→ **resulting in** Q_{n+1} **being a weighted average of past rewards and the initial estimate** Q_1 (sometimes called an *exponential recency-weighted average*):

$$Q_{n+1} = (1 - \alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1 - \alpha)^{n-1} R_i \qquad (2.6)$$

2.5 Optimistic Initial Values

What: a common strategy of balancing exploration-exploitation.

How: encouraging early exploration with optimistic initial values for all possible actions.

Limitations:

- Drive only early exploration
- Not well-suitable for non-stationary problems
- We may not always know how to set the optimistic initial values, because in practice we may not know the maximal reward.

3 Finite Markov Decision Processes

Markov decision process (MDP) provides a mathematical framework for modeling sequential decision-making in situations where outcomes are partly random and partly under the control of a decision maker.

A **MDP is a 4-tuple** (S, A, p, R), where:

- S is a set of states called the state space,
- A is a set of actions called the action space (alternatively, $A(s)$ is the set of actions available from state s),
- $p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a

The state and action spaces may be **finite** or **infinite**:

- e.g. the set of real numbers is infinite.
- Some processes with countably infinite state and action spaces can be reduced to ones with finite state and action spaces.

A **policy function** π is a (potentially probabilistic) mapping from state space to action space.

Optimization objective $\rightarrow$ find a good "policy" for the decision maker.

- **EXTRA:** Once a MDP is **combined with** a policy in this way, this fixes the action for each state and the resulting combination **behaves like a Markov chain** (since the action chosen in state s is completely determined by $\pi(s)$ and $\Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ reduces to $\Pr(s_{t+1} = s' \mid s_t = s)$, a **Markov transition matrix**).

3.1 The Agent–Environment Interface

Agent: learner and decision maker.

Environment: thing agent interacts with, comprising everything outside the agent.

The environment also gives rise to **rewards**, special numerical values that the agent seeks to maximize over time through its choice of **actions**.

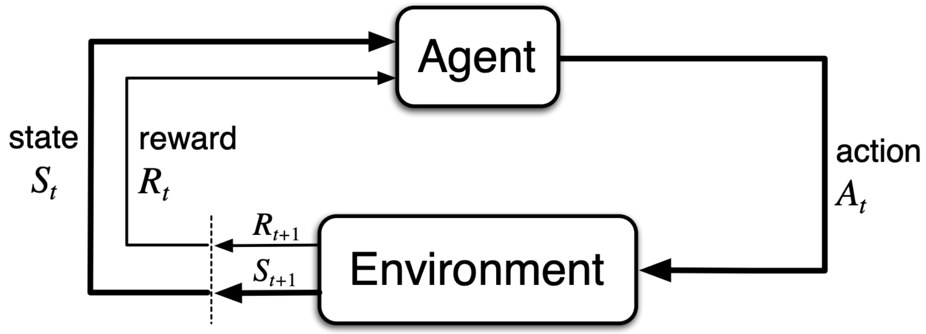

Figure 3: MDP agent-environment interaction

The MDP and agent together give rise to a *sequence* or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

In general, **actions** can be any decisions we want to learn how to make, and the **states** can be anything we can know that might be useful in making them.

In a **finite MDP**, the sets of states, actions, and rewards (S, A, and R) all 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.

There is a **probability of those values occurring at time t**, given particular values of the preceding state and action:

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

Where:

- s' = particular values of the random variable S ($s' \in S$)
- r = particular values of the random variable R ($r \in R$)

The **function** p defines the dynamics of the MDP. p specifies a probability distribution for each choice of s and a.

Total probability is thus:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, for\ all\ s \in S, a \in A(s)$$

Markov property: The state must include information about all aspects of the past agent–environment interaction that make a difference for the future.

- (only present matters)
- (things/rules/transition model are stationary)
- We will assume the Markov property throughout this book.

Calculations from the four-argument dynamics function:

- **state-transition probabilities** $p : S \times S \times A \to [0, 1]$
  - $p(s'|s, a) = Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a)$
- **expected rewards for state–action pairs** $r : S \times A \to \mathbb{R}$
  - $r(s, a) = E[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} \sum_{s' \in S} p(s', r|s, a)$
- **expected rewards for state–action–next-state triples** $r : S \times A \times S \to \mathbb{R}$
  - $r(s, a, s') = E[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)}$

3.2 Goals and Rewards

Reward Hypothesis:

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

It is critical that the rewards we set up truly indicate what we want accomplished.

- **For example**, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or gaining control of the center of the board. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal.

Reward signal is your way of communicating to the robot what you want it to achieve, not how you want it achieved.

3.3 Returns and Episodes

In general, we seek to maximize the expected **return**, where the return, denoted G_t.

Episodic Tasks

G_t is in the simplest case the **return** of the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \qquad (3.7)$$

Tasks with *episodes* of this kind are called **episodic tasks**. In episodic tasks we **sometimes need to distinguish** the set of all nonterminal states, denoted S, from the set of all states plus the terminal state, denoted S^+. The time of termination, T, is a random variable that normally varies from episode to episode.

Continuing Tasks

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on **continually without limit**. We call these **continuing tasks**.

In this book we usually use a definition of return that is slightly more complex conceptually but much simpler mathematically. The additional concept that we need is that of **discounting**. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad (3.8)$$

Where:

- $\gamma =$ is a parameter, $0 \leq \gamma \leq 1$, called the **discount rate**.
  - If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence $\{R_k\}$ is bounded.
  - If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1}. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced.
  - As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.
    - If $\gamma = 1$, undiscounted.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$
\begin{aligned}
G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\
&= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\
&= R_{t+1} + \gamma G_{t+1}
\end{aligned}
$$

- Note that this works for all time steps $t < T$, even if termination occurs at $t + 1$, if we define $G_T = 0$
- Note that although the return is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant, if $\gamma < 1$.

3.4 Unified Notation for Episodic and Continuing Tasks

In the preceding section we described two kinds of reinforcement learning tasks:

- **episodic tasks:** agent–environment interaction naturally breaks down into a sequence of separate episodes.
- **continuing tasks:** agent–environment interaction don't breaks down into a sequence of separate episodes.

We have defined the return as a sum over a finite number of terms in one case (3.7) and as a sum over an infinite number of terms in the other (3.8). These two can be unified by considering **episode termination** to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero. For example, consider the state transition diagram:

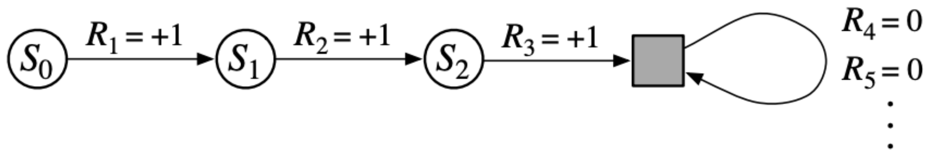

Figure 4: State transition diagram

Starting from S_0, we get the reward sequence $+1, +1, +1, 0, 0, 0, \ldots$. Summing these, we get the same return whether we sum over the first T rewards (here $T = 3$) or over the full infinite sequence. This remains true even if we introduce discounting.

Thus, we can define the **return, in general**, according to (3.8), using the convention of omitting episode numbers when they are not needed, and including the possibility that = 1 if the sum remains defined (e.g., because all episodes terminate). Alternatively, we can write:

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

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

3.5 Policies and Value Functions

Almost all **reinforcement learning algorithms involve estimating value functions** — functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). They allow an agent to query the quality of its current situation instead of waiting to observe the long-term outcome. Thus, **value functions** enable us to judge the quality of different **policies**.

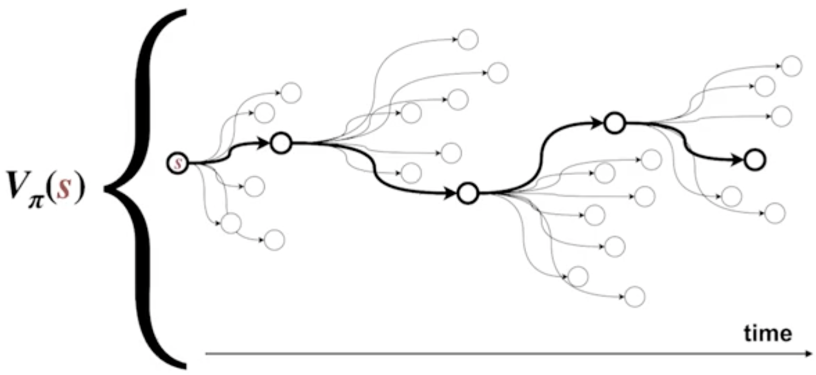

Figure 5: Illustration of value function predicting rewards into the future

Policy π is a **mapping from** states **to** probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

A policy by definition depends only on the current state. It cannot depend on things like time or previous states. This is best thought of as a restriction on the state, not the agent. The state should provide the agent with all the information it needs to make a good decision.

- Accordingly, value functions are defined with respect to particular ways of acting, called **policies**.

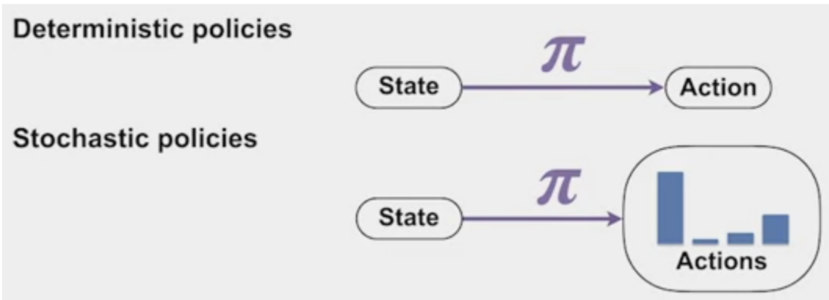

Figure 6: Policies tell an agent how to behave in their environment

Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

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

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], for\ all\ s \in S \qquad (3.12)$$

Where:

- $E_\pi[]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step.

The **action-value function** $q_\pi(s, a)$ for policy π is defined as the **value of taking action** a **in state** s **under a policy** π, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \quad (3.13)$$

A **fundamental property of value functions** used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return (3.9).

3.6 Optimal Policies and Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run.

For finite MDPs, we can precisely define an **optimal policy** π_* as follows: * $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$ * **Always at least one policy** that is better than or equal to all other policies.

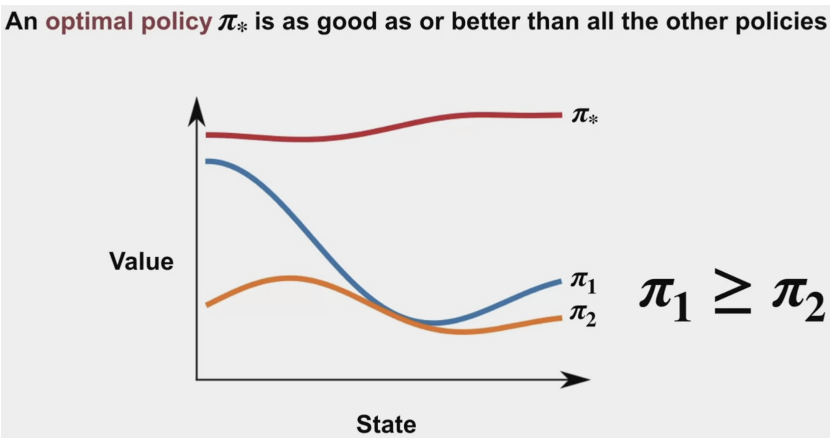

Figure 7: Illustration of optimal policy

How to find an optimal policy:

- Once we had the optimal state value function, it's relatively easy to work out the optimal policy.
- If we have the optimal action value function, working out the optimal policy is even easier.

All optimal policies share the same **optimal state-value function** v_* defined as:

$$v_*(s) = \max_\pi v_\pi(s) \; for \; all \; s \in S \qquad (3.15)$$

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

$$q_*(s,a) = \max_\pi q_\pi(s,a) \; for \; all \; s \in S \; \& \; a \in A(s) \qquad (3.16)$$

For the state–action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* **in terms of** v_* as follows **(Bellman equation)**:

$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \qquad (3.17)$$

3.7 Optimality and Approximation

For the kinds of tasks in which we are interested, optimal policies can be generated only with extreme computational cost. In particular, the amount of computation it can perform in a single time step.

3.8 Why Bellman equations

Bellman equations define a **relationship between** the value of a state or state-action pair and its possible successor states.

Consider:

- **Environment:**
  - 4 states (A, B, C, D) in a square gridworld
  - 25% probability of moving (up, down, left or right)
    - bumping into a border will result into staying in the current state
- **Policy:** Uniform random policy
- **Rewards:** The reward is 0 everywhere except for any time the agent lands in state B, the reward is +5.
- **Discoun factor γ:** 0.7

For the given setup, we can write value functions for each of the states (A, B, C, D) in form of Bellman equations:

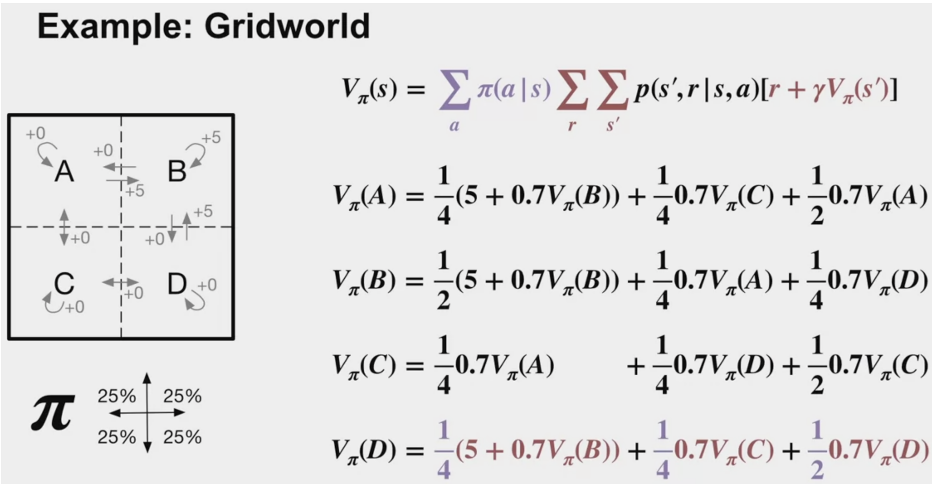

Figure 8: Value functions for all gridworld states

- **Unique solutions to the Bellman equations:**
  - $V_\pi(A) = 4.2$
  - $V_\pi(B) = 6.1$
  - $V_\pi(C) = 2.2$
  - $V_\pi(D) = 4.2$

The **important thing to note** is that the Bellman equation reduced an unmanageable infinite sum over possible futures, to a simple linear algebra problem.

In this case we used the **Bellman equation** to directly write down a system of equations for the state values, and then some the system to find the values. This approach may be possible for MDPs of moderate size. However, in more complex problems, this won't always be practical.

- e.g. Consider the game of chess for example. We probably won't be able to even list all the possible states, there are around 10 to the 45 of them.

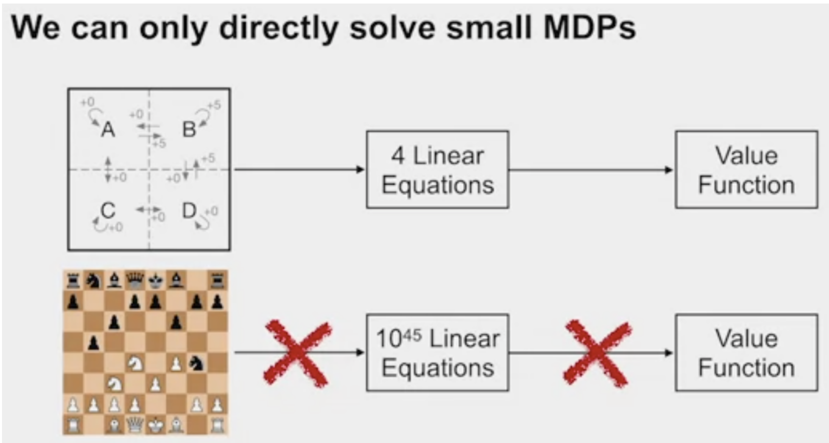

Figure 9: Illustration how directly solving MDPs gets quickly problematic

3.9 Summary

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run.

The **first step in applying reinforcement learning** will always be to formulate the problem as an MDP.

Markov property: The state must include information about all aspects of the past agent–environment interaction that make a difference for the future.

- (only present matters)

- (things/rules/transition model are stationary)
- We will assume the Markov property throughout this book.

Policy π, $\pi(a|s)$: action to take for any given state

- Any policy: $\pi(s) \rightarrow a$
- Optimal policy: $\pi^*(s) \rightarrow a$ * maximizes long-term expected reward

State-Value functions $v_\pi(s)$: expected cumulative rewards when starting in s and following policy π thereafter.

- or MDPs, we can define v_π formally by

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

- **State-Value functions $v_\pi(s)$:** can be decomposed into **immediate** and **future** components using **Bellman equation**. Bellman equation forms the basis of a number of ways to compute, approximate, and learn v_π.

$$V(s) = E[G_t|s_t = s]$$
$$V(s) = E[r_{t+1} + \gamma V(s_{t+1}|s_t = s)]$$

Where:

$\gamma =:$ is a parameter, $0 \leq \gamma \leq 1$, called the **discount rate**.

When $0 \rightarrow$ consider only immediate rewards

When approaches $1 \rightarrow$ forward looking

Action-Value functions $q_\pi(s, a), Q_\pi$: expected cumulative reward of taking action a when starting in s and following policy π thereafter.

$$q_\pi(a) = E[R_t|A_t = a]$$

- **Action-Value functions** $v_\pi(s)$**:** can also be decomposed into **immediate** and **future** components using **Bellman equation**.

$$Q_\pi(s,a) = E_\pi[r_t + \gamma Q_\pi(S_{t+1}, a_{t+1})|s_t = s, a_t = a]$$

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s')r(s,a,s') + \gamma \sum_{s'} T(s,a,s')Q_\pi(s', \pi(s'))$$

Return G **& Rewards** R**:** return is the total of rewards

- $R(s)$ = reward of **entering state** s
- $R(s,a)$ = reward of **entering state** s & **taking action** a
- $R(s,a,s')$ = reward of **being in state** s & **taking action** a & **entering state** s'

State S**:** every state agent could be in

- S = set of **nonterminal states**
  - s' = particular values of the random variable S ($s' \in S$)
- S^+ = set of **terminal states**

Actions $A, A(s)$**:** every action agent could take

Function p **(a.k.a Model / Transition function):** defines the dynamics of the environment. p specifies a probability distribution for each choice of s and a.

- **state-transition probabilities** $p : S \times S \times A \rightarrow [0,1]$

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

---

4 Dynamic Programming (DP)

What: refers to a collection of algorithms that can be used to **compute optimal policies** given a perfect model of the environment as a Markov decision process (MDP).

Pros and cons:

- + Well developed mathematically
  - All methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.
- **-** Require a complete and accurate model of the environment (access to the dynamics function p)

DP algorithms **use the Bellman equations to define iterative algorithms** for both policy evaluation and policy control.

4.1 Policy Evaluation vs. Policy Control

What:

- **Policy Evaluation:** the task of determining the value function v_π for a specific policy π
- **Policy Control:** the task of improving an existing policy π (until it's optimal π_*).

Why:

- **Policy Evaluation:** for assessing how good a policy is $\rightarrow$ (to improve it)
- **Policy Control:** for finding the optimal policy (goal of reinforcement learning)

4.2 Iterative Policy Evaluation (Prediction)

What: **In DP** we can do **policy evaluation** to **iteratively** improve value function.

How: by turning **Bellman equation** into an **update rule**:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')]$$

$$\downarrow$$

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_k(s')]$$

- This will **produce** a sequence of better and better approximations to the value function.
- We begin with an **arbitrary initialization** for our approximate value function, let's call this v_0.
  - For any v_0:
    $$\lim_{k \to \infty} v_k = v_\pi$$
- To implement iterative policy evaluation, we **store two arrays**:
  $$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V(s')]$$
  - V [| | | |] $\rightarrow$ stores the current approximate value function
  - V' [| | | |] $\rightarrow$ stores the updated values
    * we can compute the new values from the old one state at a time without the old values being changed in the process.
    * At the end of a full sweep, we can write all the new values into V; then we do the next iteration.
  - It is also possible to implement a version with only one array, in which case, some updates will themselves use new values instead of old. This single array version is still guaranteed to converge, and in fact, will usually converge faster.

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

Input: π (policy to be evaluated)

Algorithm parameters:

- $\theta \leftarrow$ small threshold determining accuracy of estimation

Initialize:

- $V(s)$, for all $s \in \mathcal{S}^+$ arbitrarily
- $V'(s)$, for all $s \in \mathcal{S}^+$ arbitrarily
- $V(terminal) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$**:**

$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$

$V \leftarrow V'$

Until: $\Delta < \theta$ (a small positive number)

Output: $V \approx v_\pi$

- We track the largest update Δ to each state value in a given iteration.
- The outer loop terminates when this maximum change Δ is less than some user-specified constant θ.

4.3 Policy Improvement (Control)

Policy improvement theorem: tells us that greedified policy is a strict improvement, (unless the original policy was already optimal). In other words, it tells us that we can construct a strictly better policy by acting greedily with respect to the value function of a given policy, unless the given policy was already optimal.

Greedy policy:

$$\pi'(s) = \arg\max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Better:

$$q_\pi(s, \pi'(s)) \geq q_\pi(s, \pi(s)) \text{ for all } s \in \mathcal{S} \rightarrow \pi' \geq \pi$$

Strictly better:

$$q_\pi(s, \pi'(s)) > q_\pi(s, \pi(s)) \text{ for at least one } s \in \mathcal{S} \rightarrow \pi' > \pi$$

4.4 Policy Iteration

Policy iteration is the process of alternating between policy evaluation and policy improvement (control), which can be illustrated by the following figure:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*.

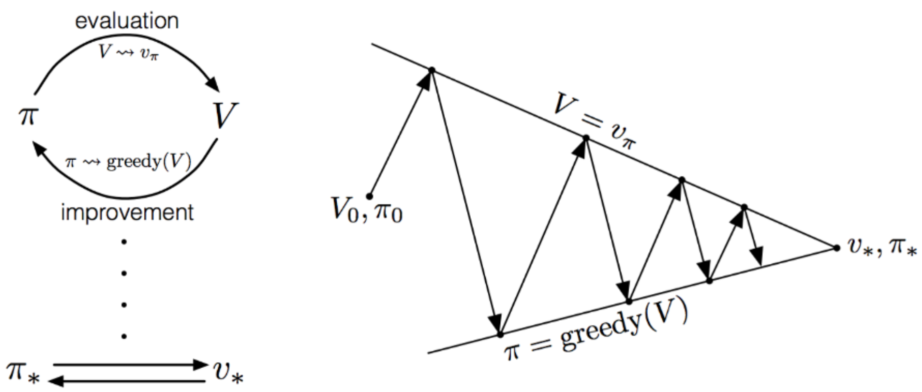

Figure 10: Illustration of policy iteration

Briefly speaking, we initially take a random policy π, then compute a state-value function v_π and use v_π to compute q_π. After that, we select the new greedy policy $\pi'(s)$ from q_π:

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

Each policy is guaranteed to be an improvement on the last unless the last policy was already optimal.

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 $s \in \mathcal{S}$**:**

   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

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

4.5 Generalized Policy Iteration (GPI)

What: We use the term generalized policy iteration (GPI) to refer to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes.

Why: Almost all reinforcement learning methods are well described as GPI.

4.6 Value Iteration

What: a special case of generalized policy iteration.

Why: It allows us to **combine** policy evaluation and policy improvement into a single update.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameters:

- $\theta > 0 \leftarrow$ small threshold determining accuracy of estimation

Initialize:

- $V(s)$, for all $s \in \mathcal{S}^+$ arbitrarily
- $V(terminal) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$**:**

$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Until: $\Delta < \theta$ (a small positive number)

Output: a deterministic policy, $\pi \approx \pi_*$, such that:

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

We do not run policy evaluation to completion. We perform just one sweep over all the states. After that, we greedify again. We can write this as an update rule $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$ which applies directly to the state-value function. The update does not reference any specific policy, hence the name **value iteration**.

- Instead of updating the value according to a fixed falsey, we update using the action that maximizes the current value estimate.

4.7 Asynchronous Dynamic Programming

What: a special case of generalized policy iteration.

Asynchronous dynamic programming methods give us freedom to update states in any order, they do not perform systematic sweeps.

Asynchronous algorithms can propagate value information quickly through **selective updates**. This can sometimes be more efficient than a systematic sweep.

4.8 Efficiency of Dynamic Programming

The **key insight of dynamic programming** is that we do not have to treat the evaluation of each state as a separate problem. We can use the other value estimates we have already worked so hard to compute.

The process of using the value estimates of successor states to improve our current value estimate is known as **bootstrapping**. This can be much more efficient than (e.g. a Monte Carlo) method that estimates each value independently.

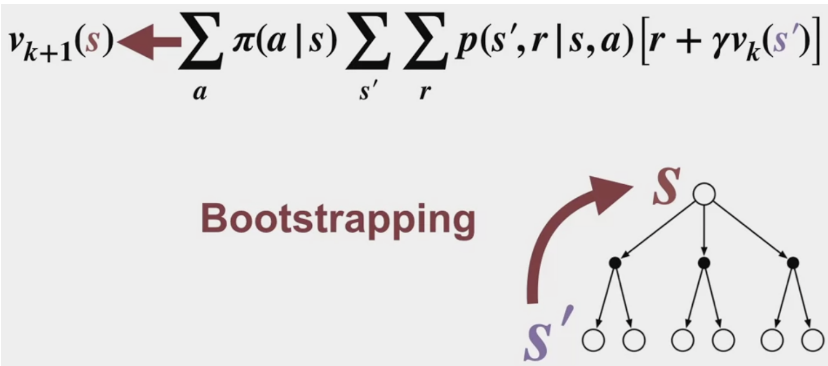

Figure 11: Illustration of bootstrapping in Dynamic Programming

5 Monte Carlo Methods

NOTE

To ensure that well-defined returns are available, here we define Monte Carlo methods only for **episodic tasks**.

- That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected.
- Only on the completion of an episode are value estimates and policies changed.

What: sample-based methods for solving reinforcement learning problem.

- **estimating** value functions $\rightarrow$ **discovering** optimal policies.

How: based on **averaging** sample returns **for each** state–action pair.

Pros and cons:

- + Don't require a model (direct access to the environment dynamics)
- + Conceptually simple
- - Not well suited for step-by-step incremental computation

Base:

- Unlike previously, we **don't assume complete knowledge of the environment**.
  - Unknown transaction dynamics (T) and reward function (r)
- Require only *experience* — **sample** sequences of **states, actions, and rewards** from actual or simulated interaction with an environment.
- There are **multiple states**, *each acting like a different bandit problem* (like an associative-search or contextual bandit) and the different bandit problems are interrelated.

Terms:

- The term **"Monte Carlo"** is often used more broadly for any estimation method whose operation involves a significant random component.
  - **HERE:** Here we use it specifically for methods based on averaging complete returns.

5.1 Monte Carlo Prediction

Monte Carlo methods for learning the state-value function for a given policy.

The **first-visit MC method** (focus of this chapter) estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the **every-visit MC method** averages the returns following all visits to s.