Reinforcement Learning NOTES

Kristian Bonnici

October 27, 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_***

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

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

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

- **Value iteration:**, for estimating $\pi \approx \pi_* \rightarrow$ `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):
  - $V_{i+1}^*(s) = \max_a \sum_{s'} T(s, a, s')(r(s, a, s') + \gamma V_i^*(s'))$
  - $= \max_a ImmediateReward + Discount * FutureRewards$

- **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:** → 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. → 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 → **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:

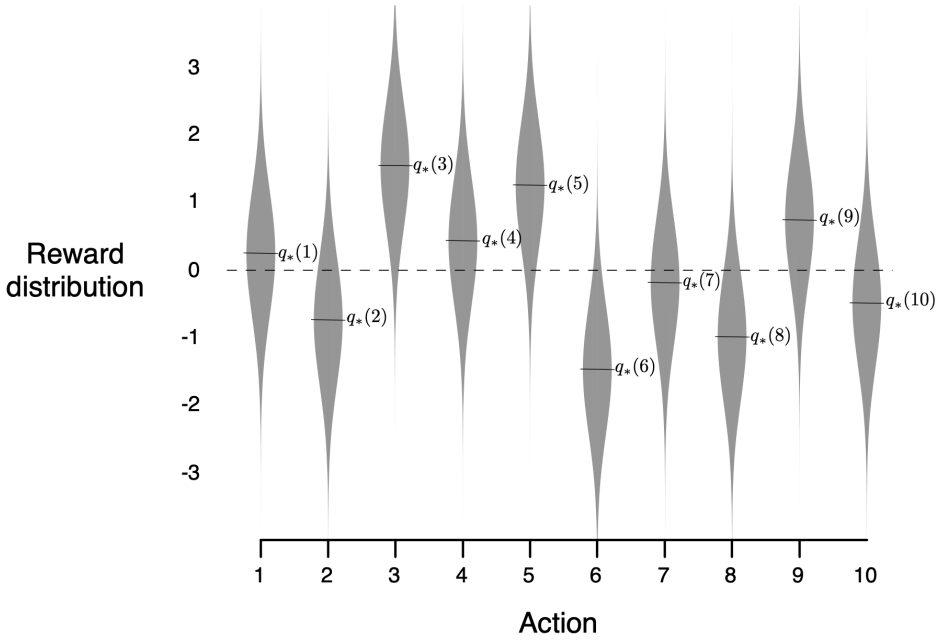


Figure 1: 10-armed testbed

- **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 $\epsilon = 0.1$):

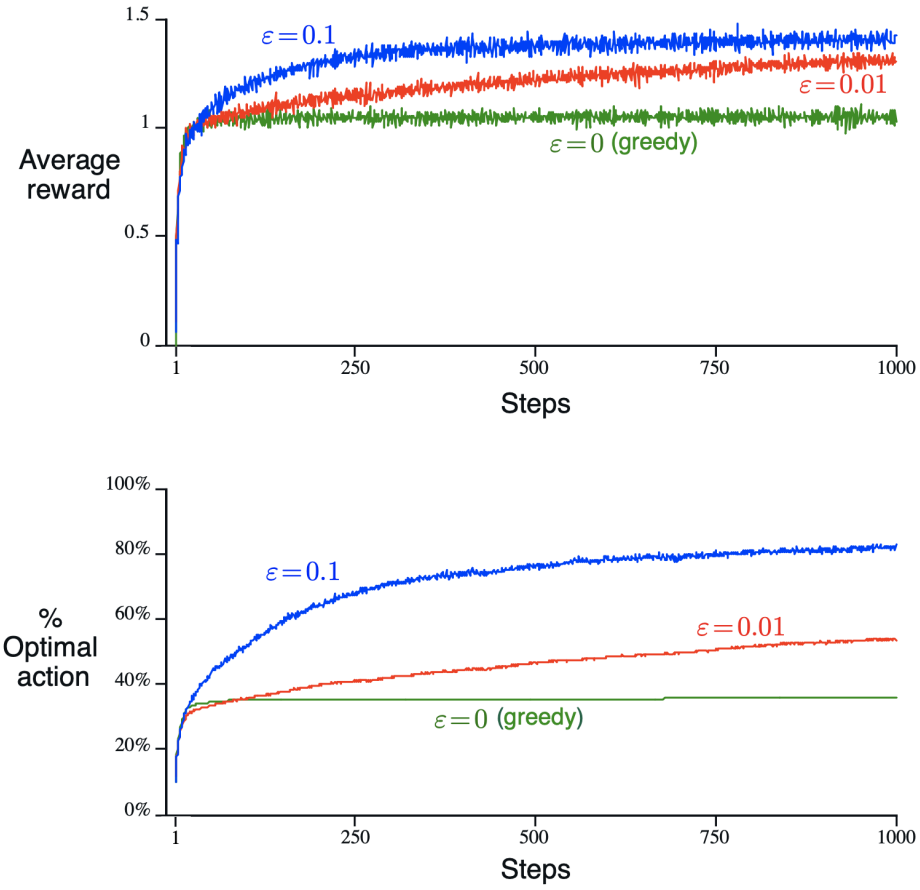


Figure 2: Average performance of ϵ-greedy action-value methods on the 10-armed testbed

- **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

$\rightarrow$ **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**.

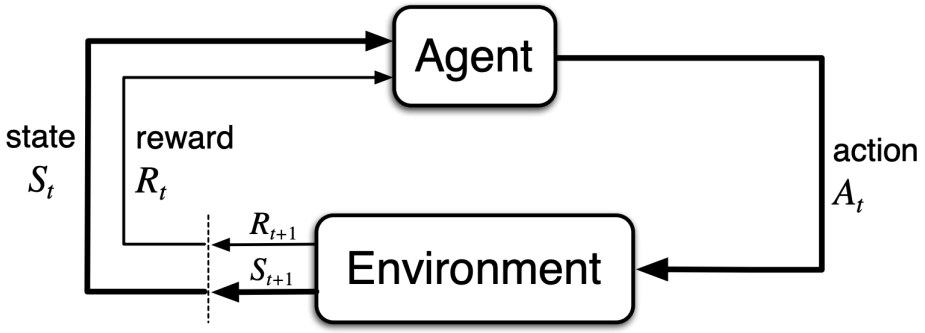


Figure 3: MDP agent-environment interaction

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:

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

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:

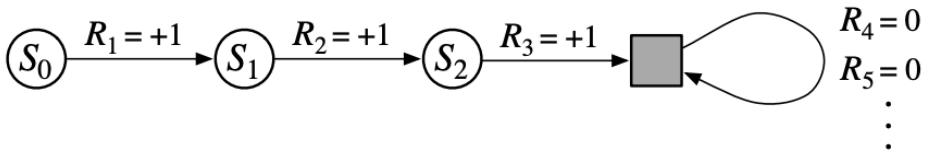


Figure 4: State transition diagram

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**.

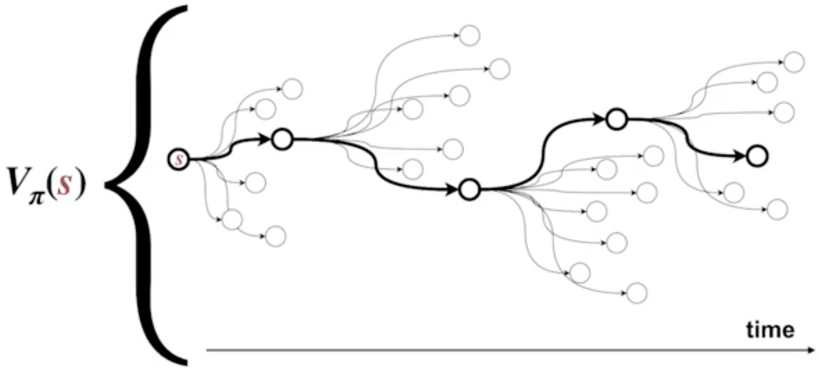


Figure 5: Illustration of value function predicting rewards into the future

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**.

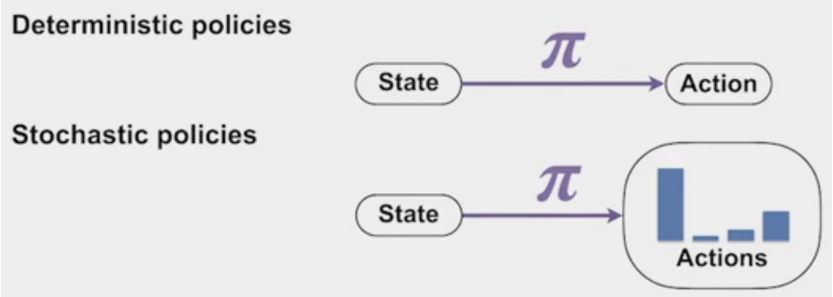


Figure 6: Policies tell an agent how to behave in their environment

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.

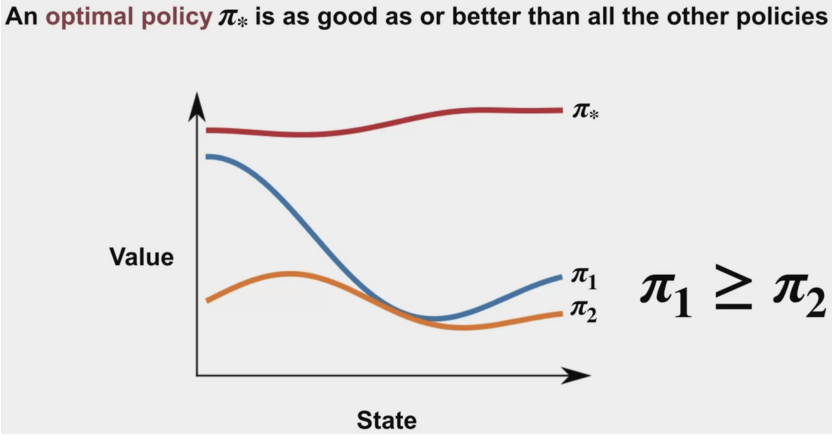


Figure 7: Illustration of optimal policy

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:

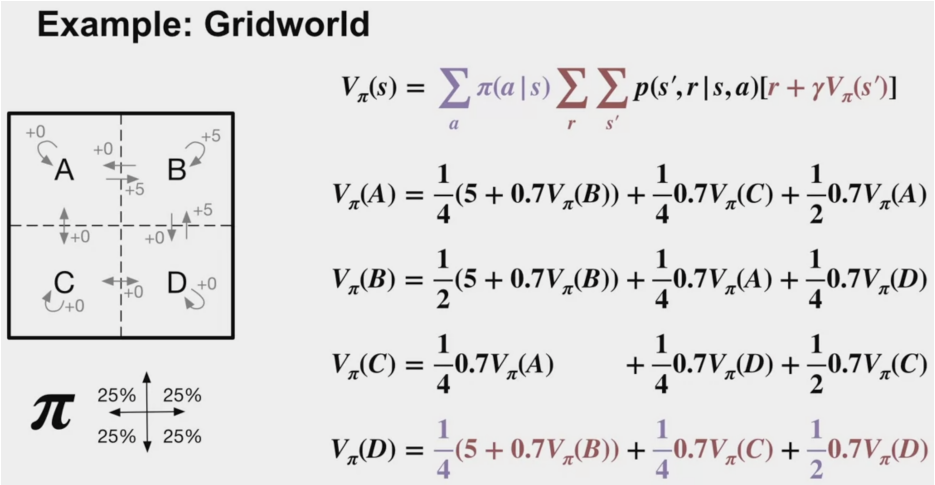


Figure 8: Value functions for all gridworld states

- **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.

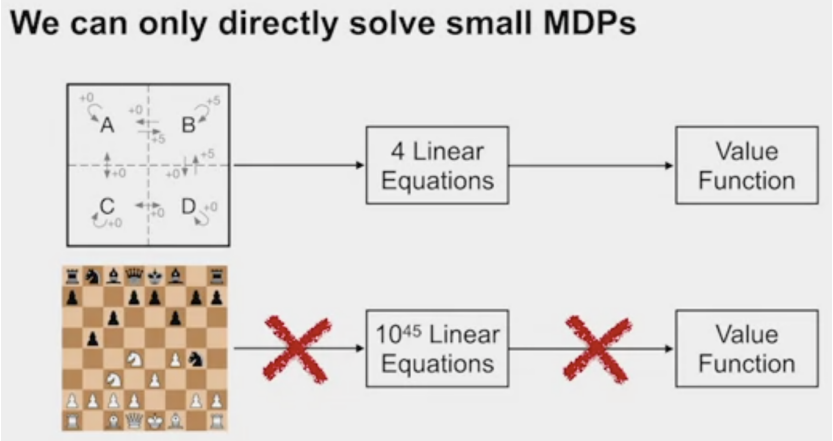


Figure 9: Illustration how directly solving MDPs gets quickly problematic

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** $p : S \times S \times A \to [0,1]$

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

---

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 [| | |] → stores the current approximate value function
  - V' [| | |] → 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

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 (Control)

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

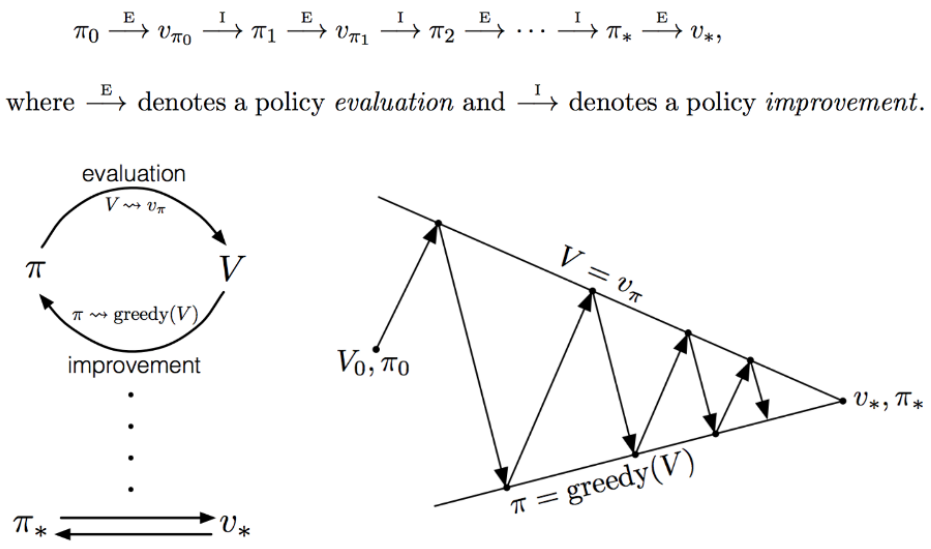


Figure 10: Illustration of policy iteration

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.

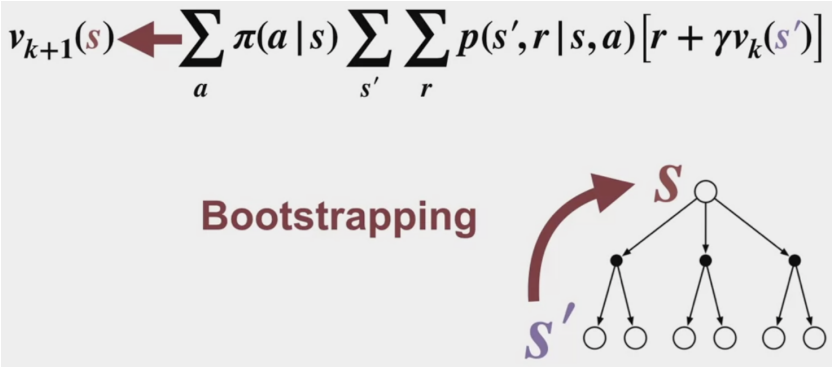


Figure 11: Illustration of bootstrapping in Dynamic Programming

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 → **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) → Monte Carlo methods learn directly from interaction.
- + 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 Policy Evaluation (Prediction)

What: Monte Carlo methods for **Policy Evaluation** (estimating value function v_π for a specific policy π).

Implications of MC learning:

- **learns directly from experience** → no need to keep a large model of the environment.
- can **estimate the value of an individual state independently of the values of any other states**.
- **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 **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.

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

Input: a policy π to be evaluated
Initialize:

- $V(s) \in \mathbb{R}$, for all $s \in \mathcal{S}$ arbitrarily
- $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π

$G \leftarrow 0$

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

$G \leftarrow \gamma G + R_{t+1}$
Unless S_t **appears in** $S_0, S_1, ..., S_{t-1}$:
Append G to $Returns(S_t)$
$V(S_t) \leftarrow average(Returns(S_t))$

- We can repeat the whole process over many episodes (loop forever) and eventually learn a good estimate for the value function.
- **We can avoid keeping all the sampled returns in a list:** We can incrementally update the sample average estimated using the formula:

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

5.2 Monte Carlo Estimation of Action Values

What: Using **Monte Carlo methods** to learn **Action-Value functions** q_π (instead of state-values).

- Recall that we **learned state-values by** averaging sample returns from that state.

- Similarny we **learn action-values by** averaging sample returns from that state, action pair.

Why: Action values are useful for learning a policy. They allow us to compare different actions in the same state. Then, we can switch to a better action if one is available.

5.3 Monte Carlo Control

What: Using **Monte Carlo methods** to **approximate optimal policies**. Namely Monte Carlo methods to implement a Generalized Policy Iteration (GPI) algorithm (policy evaluation and policy improvement processes interact).

How:

- For the policy improvement step, we can make the policy greedy with respect to the agent's current action value estimates.
- For the policy evaluation step, we will use a Monte Carlo method to estimate the action values.
- For Monte Carlo policy iteration it is natural to **alternate between** evaluation and improvement on an episode-by-episode basis.
  - **After each episode**,
    1. the **observed returns** are used for policy evaluation, and
    2. then the policy is improved at all the states visited in the episode.

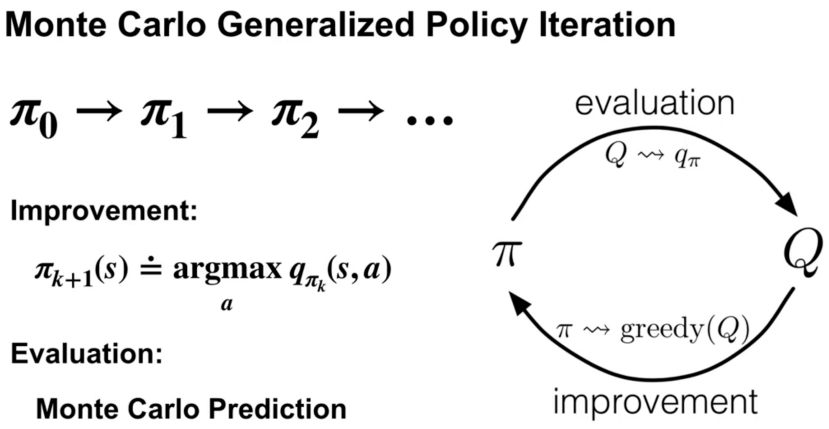


Figure 12: Process of Monte Carlo General Policy Iteration.

A complete simple algorithm along these lines, which we call Monte Carlo ES, for Monte Carlo with Exploring Starts, is given in pseudocode below.

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

Initialize:

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

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(\mathcal{S}_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π

$G \leftarrow 0$

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

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

Unless S_t, A_t **appears in** $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **:**

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

- **Exploring start:** 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 selected as the start. This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

5.4 Monte Carlo Control without Exploring Starts

What: Alternative ways (to Exploring Starts) for performing exploration. Here we use ϵ-Soft policies.

Why: It can be difficult to randomly sample an initial State action pair.

- e.g. For example, how would you randomly sample the initial State action pair for a self-driving car? How could we ensure the agent can start in all possible States? We would need to put the car in many different configurations in the middle of a busy freeway. This would be dangerous and impractical.

ϵ-Soft policies: take each action with probability at least Epsilon over the number of actions ($\epsilon/|\mathcal{A}|$).

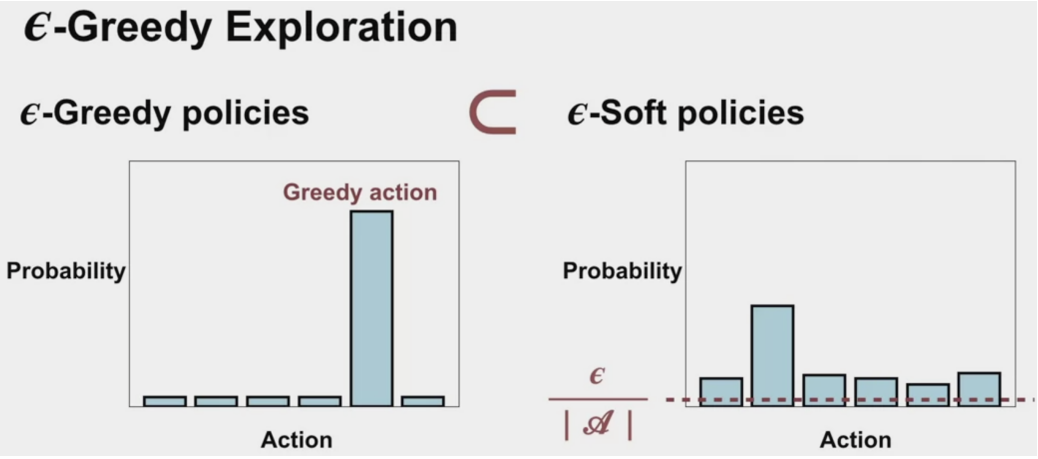


Figure 13: ϵ-greedy policies are a subset of ϵ-soft policies

However, if our policy always gives at least Epsilon probability to each action, it's impossible to converge to a deterministic optimal policy

$$\pi_* > Optimal\ \epsilon\text{–}soft$$

We will later discuss how we can learn the optimal policy using a different method called **Q-learning**.

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

Algorithm parameter: small ϵ ¿ 0
Initialize:

- $\pi(s) \leftarrow$ an arbitrary ϵ-soft policy
- $Q(s,a) \in \mathbb{R}$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily
- $Returns(s,a) \leftarrow$ an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode from S_0, A_0, following π:

$G \leftarrow 0$

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

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

Unless S_t, A_t **appears in** $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$

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

5.5 Off-policy Prediction via Importance Sampling

What: Importance sampling allowing us to do Off-Policy learning. More precisely, we use the important sampling ratios to correct the returns generated by a behavior policy and we modified the on-policy Monte Carlo prediction algorithm for off-policy learning.

Why: Off-Policy learning allows us to learn optimal policy from suboptimal behaviour. This way we can learn an optimal policy but still maintain exploration.

On-Policy vs. Off-Policy:

- **On-Policy:** improve and evaluate the policy being used to select actions.
- **Off-Policy:** improve and evaluate a different policy (behavio policy $b(s|a)$) from the one used to select actions (target policy $\pi(s|a)$).
  - behavio policy $b(s|a)$ is generally an exploratory policy
  - One key rule of off policy learning is that the behavior policy must cover the target policy

$$\text{If } \pi(a|s) > 0 \text{ then also } b(a|s) > 0$$

- Learning on-policy or off-policy may perform differently in control depending on the task.

Importance Sampling: A way of estimating the expected value of a distribution using samples from a different distribution.

- Derivation of Importance Sampling – Consider:
  - Random variable x is being sampled from a probability distribution b

$$\text{Sample: } x \sim b$$

  - We want to estimate the expected value of x, but with respect to the target distribution π.

$$\text{Estimate: } \mathbb{E}_\pi[X]$$

– Because x is drawn from b, we cannot simply use the sample average to compute the expectation under π.
– We can use importance sampling to correct the expectation:

$$\begin{aligned}\mathbb{E}_\pi[X] &= \sum_{x\in X} x\frac{\pi(x)}{b(x)}b(x) \\ &= \sum_{x\in X} x \underbrace{\rho(x)}_{\text{importance sampling ratio}} b(x) \\ &= \mathbb{E}_b[X\rho(x)]\end{aligned}$$

– We can then use importance sampling to **estimate the expectation from data:** We just need to compute a weighted sample average with the importance sampling ratio as the weightings:

$$\begin{aligned}\mathbb{E}_b[X\rho(x)] &= \sum_{x\in X} x\rho(x)b(x) \\ &\approx \frac{1}{n}\sum_{i=1}^{n} x_i\rho(x_i)\end{aligned}$$

– We can now estimate the expected value of x under distribution π by using the sample average, the samples drawn from distribution b.

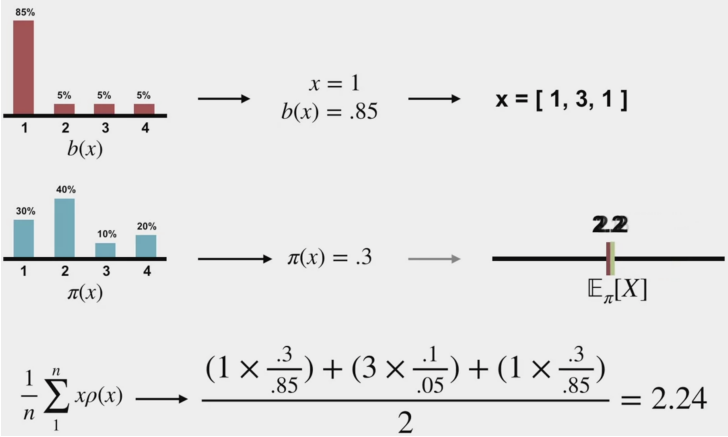


Figure 14: Example importance sampling (samples 1, 3 & 1)

* With samples just from b, we managed to get a pretty good estimate of the expectation under π.

Off-policy Prediction via Importance Sampling:

- The expectation of the return under π, sampling from b:

$$V_\pi(s) = \mathbb{E}_b[\rho G_t | S_t = s]$$

Where:

$$\text{Rho: } \rho = \frac{\mathbb{P}(\text{trajectory under } \pi)}{\mathbb{P}(\text{trajectory under } b)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

Where:

$$\mathbb{P}(\text{trajectory under } b) = \prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

$$\mathbb{P}(\text{trajectory under } \pi) = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

Incremental Implementation:

Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis, using average returns.

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

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$**:**

- $Q(s,a) \in \mathbb{R}$ (arbitrarily)
- $C(s,a) \leftarrow 0$

Loop forever (for each episode):

b any policy with coverage of π

Generate an episode following b

$G \leftarrow 0$

$W \leftarrow 1$

Loop for each step of episode, $t = T-1, T-2, ..., 0$**, while** $W \neq 0$**:**

$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$W \leftarrow W \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$

- W : accumulated product of important sampling ratios on each time step of the episode.
- We compute Rho ρ from t to $T-1$ incrementally ($W = \rho_{t:T-1}$).

$$W_{t+1} \leftarrow W_t \rho_t$$

  - We can compute this recursively without having to store all past values of Rho.

5.6 Off-policy Monte Carlo Control

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

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$**:**

- $Q(s,a) \in \mathbb{R}$ (arbitrarily)
- $C(s,a) \leftarrow 0$
- $\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):

b any soft policy

Generate an episode using b

$G \leftarrow 0$

$W \leftarrow 1$

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

$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

6 Temporal-Difference (TD) Learning

One of the most fundamental concepts in reinforcement learning!

What: class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods are a combination of Monte Carlo (MC) ideas and dynamic programming (DP) ideas.

How:

- Like MC, TD methods can learn directly from raw experience without a model of the environment's dynamics.
- Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

For the control problem (finding an optimal policy), DP, TD, and MC methods all use some variation of **generalized policy iteration (GPI)**. The differences in the methods are primarily differences in their approaches to the prediction problem.

6.1 TD Prediction

TD vs. MC

- **Both** use experience to solve the prediction problem.
- **MC methods** wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known).
- **TD methods** wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$.

We can use these recursive formula to incrementally update our estimated value:

First recall:

- G_t **(recursively):**

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

- $v_\pi(s)$ **(recursively):**

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

Incremental Monte Carlo update rule (Monte Carlo estimate without saving lists of returns):

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

- We can replace G_t with the reward + the estimate of the return in the next state (so we don't have to wait until the end of the episode, but we still have to wait to the next step):

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{t target}} - V(S_t)]}_{\text{TD-error } \delta_t}$$

- TD updates the value of one state towards its own estimate of the value in the next state. As the estimated value for the next state improves, so does our target.
  - In fact, we've done something like this before in dynamic programming. In DP, we update $V(S_t)$ toward the value of all possible next states. The primary difference is in DP, we use an expectation over all possible next states. We needed a model of the environment to compute this expectation, in TD we only need the next state. We can get that directly from the environment without a model.

  We can get the next state directly from the environment without a model, **1-Step TD**:

  - Think of time $t+1$ as the current time step and time t as the previous time step.
  - Simply store the state from the previous time step in order to make our TD updates: $s_t \leftarrow s_{t+1}$

– Only then can we update the value of the previous state.

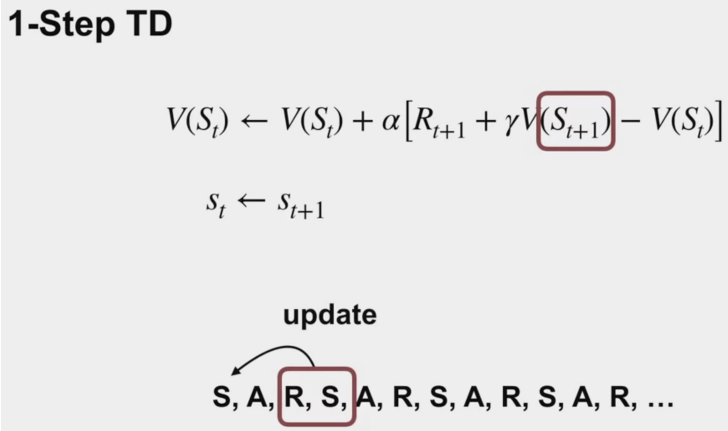


Figure 15: Illustration of 1-Step TD, TD(0)

TD(0) a.k.a 1-Step TD

So the simplest TD method (TD(0)) makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{t target}} - V(S_t)]}_{\text{TD-error } \delta_t}$$

immediately on transition to S_{t+1} and receiving R_{t+1}.

It is a special case of the $TD(\lambda)$ and n-step TD methods.

The below pseudocode specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

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

Loop for each episode:

Initialize S

Loop for each step of episode:

$A \leftarrow$ action given by π for S

Take action A observe R, S'

$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

until S **is terminal**

6.2 Rich Sutton: The Importance of TD Learning

Learning to predict (Prediction Learning) is the single most scalable kind of learning.

- It is the unsupervised supervised learning.
  - We have a target (just by waiting)
  - Yet no human labeling is needed!
- Prediction learning is **scalable** model-free learning.

TD learning is a method for learning to predict from another, later, learned prediction.

- i.e. *learnig a guess from guess*
- The TD error is the difference between the two predictions, the *temporal difference*

- Otherwise TD learning is the same as supervised-learning, backpropagating the error

TD learning benefits and insights.

- Relevant only on *multi-step* prediction problems (everything other than the classical supervised learning setup)
  - with information about it possibly revealed on each step
- It is learning specialized for general, multi-step prediction, which may be key to perception, meaning, and modeling of the world
- It takes advantage of the **state property**
  - which makes it fast, data efficient
  - which also makes it asymptotically biased
- It is computationally congenial
- **Widely used in RL** to predict future rewards (value functions)
- **Key to** Q-learning, Sarsa, TD(λ), Deep Q networks, TD-Gammon actor-critic methods, Samuel's checker player
  - but not AlphaGo, helicopter autopilots, pure policy-based methods
- Appears to be how brain reward systems work
- Can be used to predict any signal, not just rewards

6.3 Advantages of TD Prediction Methods

- Unlike dynamic programming, TD methods do not require a model of the environment.
- Unlike Monte Carlo, TD methods can learn online (update the values on every step) without waiting to know the final outcome. Bootstrapping allows us to update the estimates based on other estimates.
- Converge faster than MC methods.

6.4 Optimality of TD(0)

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α, as long as α is chosen to be sufficiently small.

The constant-α MC method also converges deterministically under the same conditions, but to a different answer.

- Under batch training, constant-α MC converges to values, V(s), that are sample averages of the actual returns experienced after visiting each state s.

These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean-squared error measure shown in Figure 16 below.

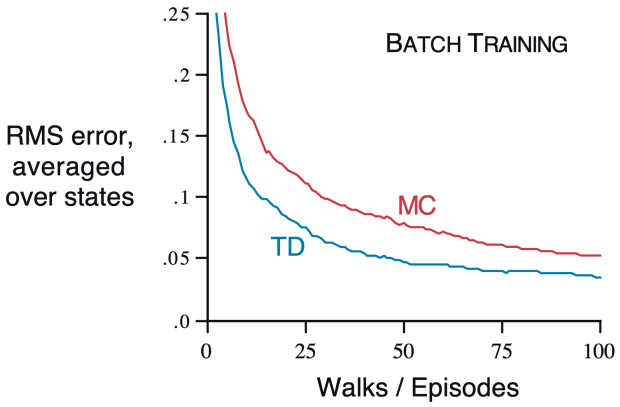


Figure 16: Performance of TD(0) and constant-α MC under batch training on the random walk task

The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns.

On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

6.5 Sarsa: On-policy TD Control

What: algorithm that perfrms on-policy TD control by learning an action-value function q_π.

The **SARSA** acronym describes the data used in the updates:

- **S**tate, **A**ction, **R**eward, next **S**tate, next **A**ction.
- S_t , A_t , R_{t+1} , S_{t+1} , A_{t+1}

Sarsa algorithm:

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

The general form of the Sarsa control algorithm is given in the box below.

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

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

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)

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

$S \leftarrow S'; A \leftarrow A'$

until S **is terminal**

In Sarsa, the agent needs to know its next state action pair before updating its value estimates. That means it has to commit to its next action before the update.

Sarsa converges with probability 1 to an optimal policy and action-value function **as long as** all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with "ϵ-greedy policies by setting " $= 1/t$).

6.6 Q-learning: Off-policy TD Control

What: algorithm that perfrms off-policy TD control by learning an action-value function q_π.

Q-learning algorithm:

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

Sarsa vs. **Q-learning**:

- **Sarsa** is a sample-based version of policy iteration which uses Bellman equation for action values, that each depend on a fixed policy.
- **Q-learning** is a sample-based version of value iteration which iteratively applies the Bellman's Optimality Equation.

Revisiting Bellman equations

Sarsa: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big)$

$$q_\pi(s, a) = \sum_{s', r} p(s', r \mid s, a)\Big(r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a')\Big)$$

Q-learning: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\big)$

$$q_*(s, a) = \sum_{s', r} p(s', r \mid s, a)\big(r + \gamma \max_{a'} q_\pi(s', a')\big)$$

- The optimality equations enable **Q-learning** to directly learn Q-star instead of switching between policy improvement and policy evaluation steps.

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

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

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'

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

$S \leftarrow S'$

until S is terminal

How is Q-learning off-policy?

- Since Q-learning learns about the best action it could possibly take rather than the actions it actually takes, it is learning off-policy.
- **Why no important sampling ratios (with 1-step)?** Because the agent is estimating action values with unknown policy. It does not need important 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 also converges to the optimal value function as long as the aging continues to explore and samples all areas of the state action space.

6.7 Expected Sarsa

The algorithm is nearly identical to Sarsa, except the TD error uses the expected estimate of the next action value instead of a sample of the next action value.

- Sarsa:

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

- Expected Sarsa:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)) \\ &\leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)) \end{aligned}$$

Motivation: Sarsa estimates this expectation by sampling the next date from the environment and the next action from its policy. But the agent already knows this policy, so why should it have to sample its next action? Instead, it should just compute the expectation directly.

Expected Sarsa vs. Sarsa (Pross & Cons)

- There's a huge upside to calculating the expectation explicitly: Expected Sarsa has a more stable update target than Sarsa.
- The lower variance comes with a downside though. Computing the average over next actions becomes more expensive as the number of actions increases. When there are many actions, computing the average might take a long time, especially since the average has to be computed every time step.
- Expected Sarsa is more robust than Sarsa to large step sizes.

Expected Sarsa can do off-policy learning without using importance sampling.

Expected Sarsa with greedy target policy $\equiv$ (*equivalent to*) Q-Learning

$\rightarrow$ Q-Learning is a special case of Expected Sarsa.

- Expected Sarsa can also do off-policy learning without using importance sampling.

7 n-step Bootstrapping

NOTE

Idea of n-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously. **Here** we instead consider the n-step bootstrapping idea on its own.

What: methods that unifies MC & TD(0) methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task.

Motivation: Neither MC methods nor one-step TD methods are always the best. The best methods are often intermediate between the two extremes.

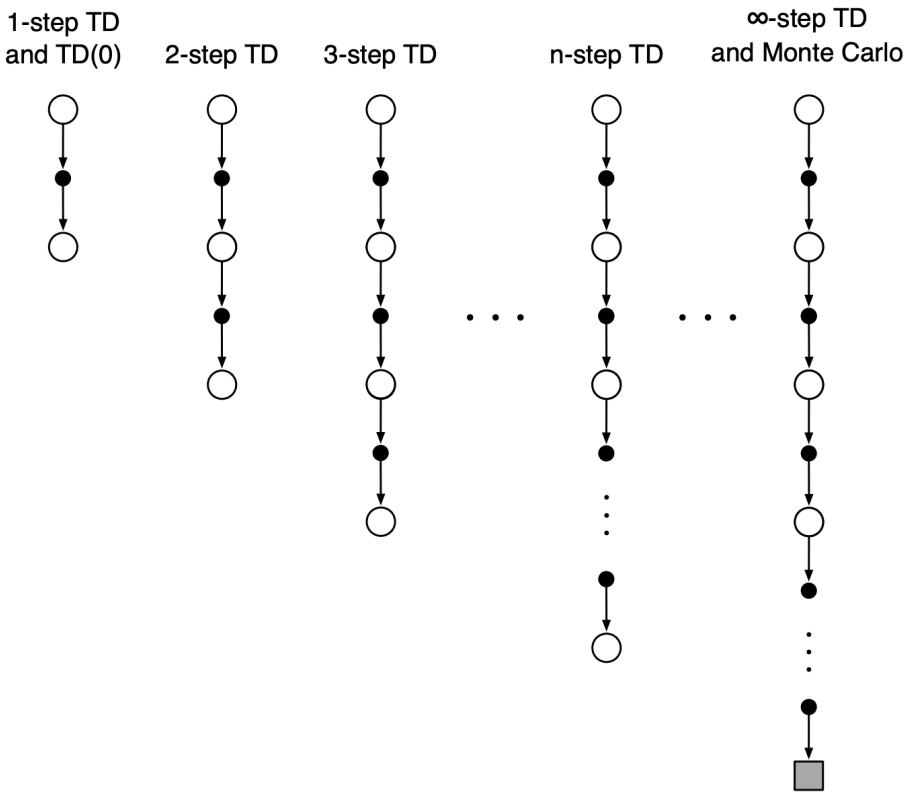


Figure 17: These n-step methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

How they work:

- n-step TD will perform an update based on the next n rewards, and the estimated value of the corresponding state (n steps ahead).

Returns of the method on the spectrum:

- **Monte Carlo** updates the estimate of $v_\pi(S_t)$ is updated in the direction of the complete return:

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

Where:

T: is the last time step of the episode

  - Let us call this quantity the *target* of the update.

- All n**-step returns** can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$.
  - Return in 1-step TD method:

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

  - Return in 2-step TD method:

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

  - Return in n-step TD method:

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

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

n-step TD Algorithm:

- No real algorithm can use the n-step return until after it has seen R_{t+n} and computed V_{t+n-1}. The first time these are available is $t+n$. The natural state-value learning algorithm for using n-step returns is thus

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

- while the values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$, for all $s \neq S_t$.

8 Planning and Learning with Tabular Methods (Model based RL)

We've seen:

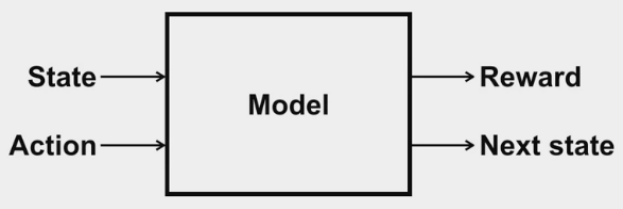
- sample-based methods like TD which learn only from sampled experience.
- dynamic programming methods which plan by using complete information about how things work, without having to make decisions.

It would be even better if we could obtain an intermediate method that can leverage the best of both extremes.

So far, we've used off-policy learning to facilitate exploration in a problem that has one goal. But actually you can use off-policy learning to learn how to get to many different goals.

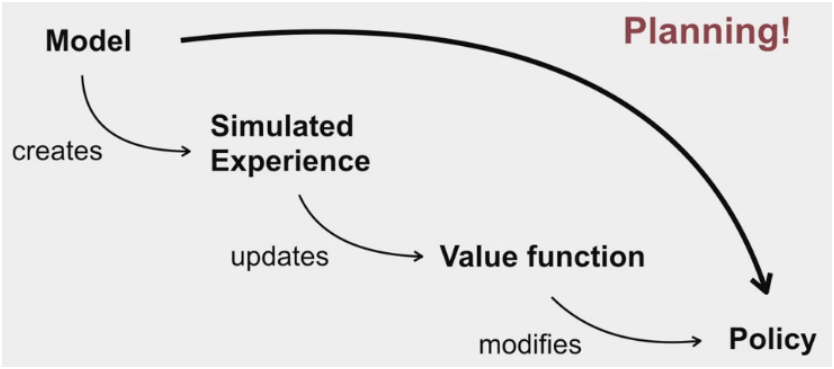
8.1 Models and Planning

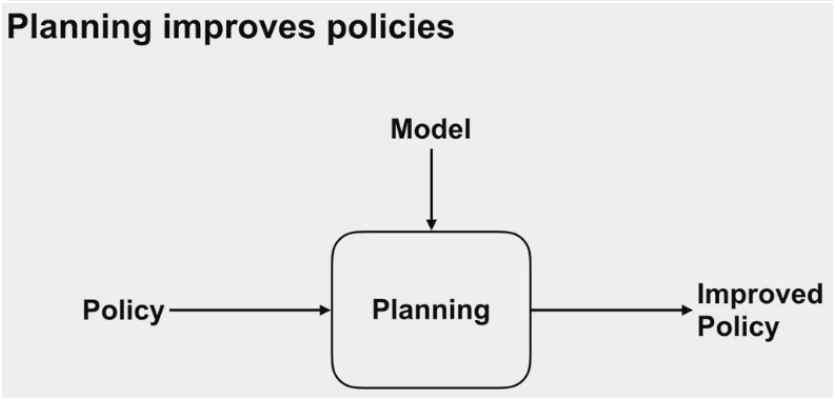
Model: is used to store knowledge about the dynamics. It should produce a possible next state and reward. This allows us to see an outcome of an action without having to actually take it.



- **Planning** refers to the process of using a model to improve a policy (with simulated experience). One way is to:
  1. Simulate experience (sample experience from the model)
     - Simulating experience improves the sample efficiency. The addition of simulated experience means the agent needs fewer interactions with the world to come up with the same policy.

2. Perform value function updates as if those experiences happened
3. Behaving greedily with respect to these improved values results in improved policy.





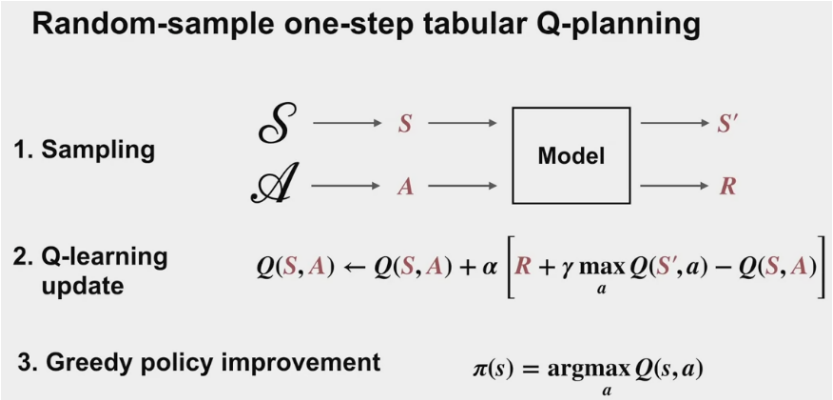
Types of Models:

- **Sample Model:** produces an actual outcome drawn from some underlying probabilities. They generate samples without explicitly storing the probability of each outcome.
  - **Pros** & **Cons**:
    - \+ require less memory / computationally inexpensive (because random outcomes can be produced according to a set of rules)
    - − can only approximate expected outcome
- **Distribution Model:** completely specifies the likelihood or probability of every outcome.
  - **Pros** & **Cons**:

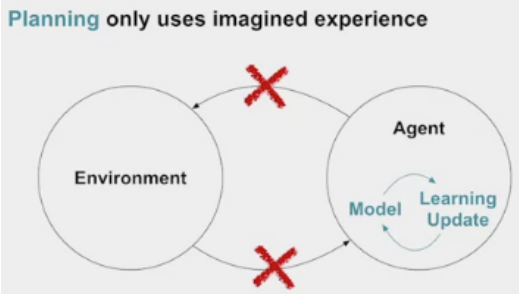
* can compute the exact expected outcome (by summing over all outcomes weighted by their probabilities). These can be used to assess risk.
* can be difficult to specify and can become very large

Random-sample one-step tabular Q-planning:

- Process:



- Assumptions:
  - we have a sample model of the transition dynamics.
  - we have a strategy for sampling relevant state action pairs

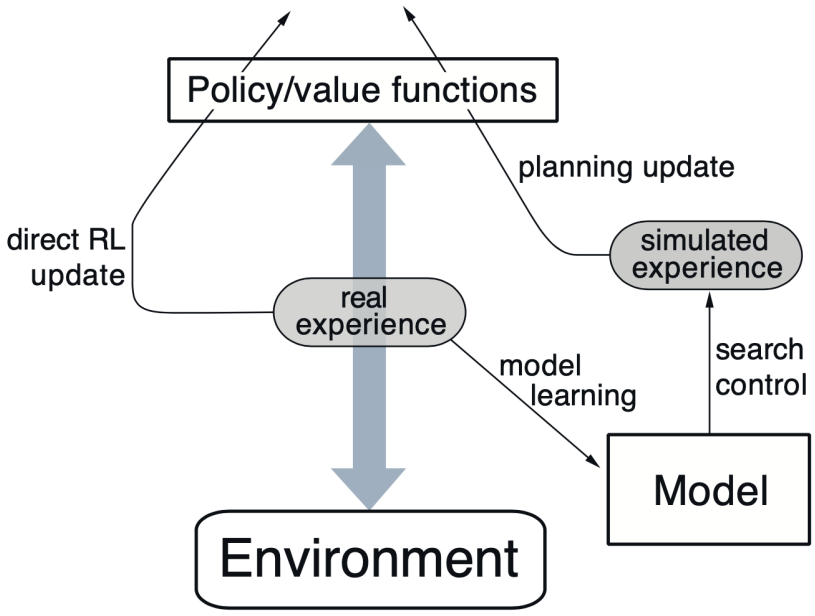


- Connection with Q-learning:
  - Recall that **Q-learning** uses experienced from the environment, they performs an update to improve a policy.
  - In **Q-planning**, we use experience from the model and perform a similar update to improve a policy.

8.2 Dyna (Architecture): Integrated Planning, Acting, and Learning

What: one way to **combine** direct RL updates and planning updates.

Dyna Architecture:



- **Search control:** controlling how the model generates simulated experience, what states the agent will plan from (most be something the agent has seen before, otherwise the model wouldn't know what happens next).

Example: Agent learning path without vs. with (n) planning steps:

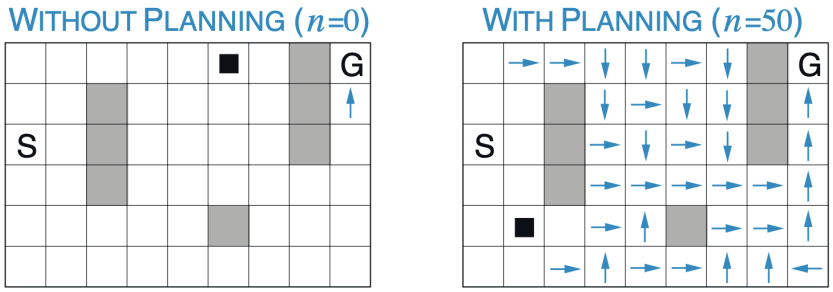


Figure 18: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode.

Pseudocode algorithm for Dyna-Q:

Tabular Dyna-Q

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

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \epsilon$-greedy(S,Q)

(c) Take action A observe R, S'

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

(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)

(f) **Loop repeat** n **times:**

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S,A)$

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

Dyna-Q is more sample efficient than Q-learning

8.3 When the Model Is Wrong / Inaccurate

Models are inaccurate when transitions they store are different from transitions that happen in the environment.

How models can be inaccurate:

- **Incomplete model:** At the beginning of learning, the agent hasn't tried most of the actions in almost all of the states. → Transitions associated with trying those actions in those states are simply missing from the model.
- **Changing environment:** Taking an action in a state could result in a different next state and reward than what the agent observed before the change.

Planning with an inaccurate model improves the policy or value function with respect to the model and not the environment.

Dyna-Q can plan with an incomplete model by only sampling state action pairs that had been previously visited.

Dyna-Q^+ algorithm: uses a reward bonus in its planning updates to encourage exploration. By exploring, Dyna-Q^+ keeps its model up-to-date and accurate, resulting in better performance.

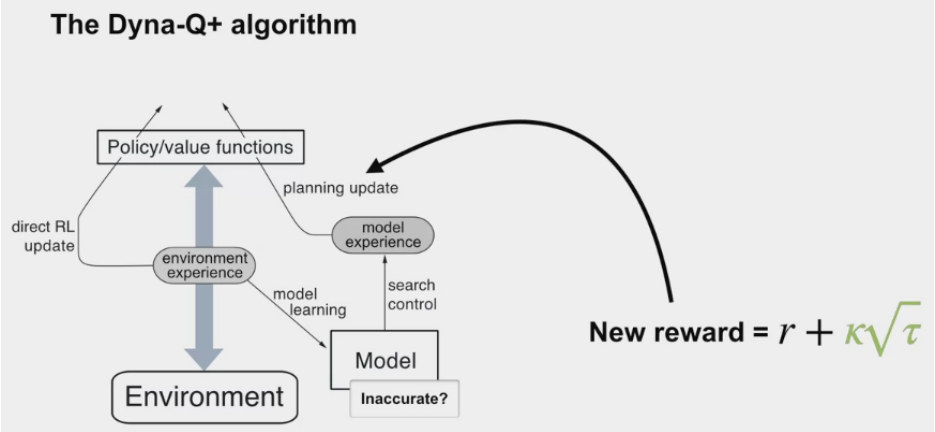
$$\text{New reward} = r + \kappa + \sqrt{\tau}$$

Where:

r: actual reward

τ: time step since transition was last tried

κ: small constant that controls the influence of the bonus on the planning update


The Dyna-Q+ algorithm
Policy/value functions
planning update
direct RL update
model experience
environment experience
model learning
search control
Model
Environment
Inaccurate?
New reward = $r + \kappa\sqrt{\tau}$

Part II

Approximate Solution methods

In this part we describe how to **apply RL for arbitrarily large state spaces**. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to *generalize* from previous encounters with different states that are in some sense similar to the current one. We can use *function approximation* (instance of *supervised learning*) for this task.

In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our **goal instead** is to find a good approximate solution using limited computational resources.

This involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part

Chapters in this section:

- Initially we restrict attention to **on-policy** training:
  9. On-policy **prediction** case (policy is given and only its value function is approximated)
  10. On-policy **control** case (approximation to the optimal policy is found)
11. **Off-policy** learning with function approximation
12. Introducing and analyzing the algorithmic mechanism of **eligibility traces**, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases.
13. Exploring a different approach to control, **policy-gradient methods**, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

9 On-policy Prediction with Approximation

Function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data.

9.1 Value-function Approximation

Function approximation 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 $s \to u$ (s should be more like u) of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Supervised methods in value approximation:

- must be able to learn online
- must handle nonstationary target functions (target functions that change over time)

Generalization in the context of policy evaluation means that updates to the value estimate of one state influence the value of other states.

- Benefits:
  - can speed learning by making better use of the experience we have → we may not have to visit every state as much to get this values correct if we can learn its value from similar states

Discrimination in the context of policy evaluation means the ability to make the values for two states different to distinguish between the values for these two states.

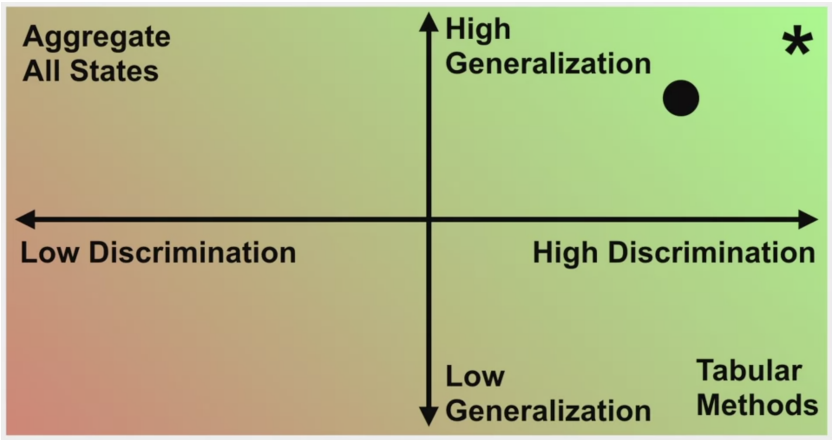


Figure 19: Categorizing methods based on generalization and discrimination

- * = What we would really like is a learning method that achieves good generalization and a good discrimination. Such a method would generalize the learn values to similar states, allowing it to learn faster, but it could also discriminate between states. Meaning, with more data, the value function approximation can accurately represent the values.
- • = In practice, we are more likely to get a point here, where we trade off some level of discrimination for generalization.

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

Policy evaluation under *function approximation* **requires us to specify an objective** → Mean Squared Value Error ($\overline{VE}$):

$$\overline{VE}(w) \doteq \sum_{s \in \mathcal{S}} \mu(s)(v_\pi(s) - \hat{v}(s,w))^2$$

Intuition:

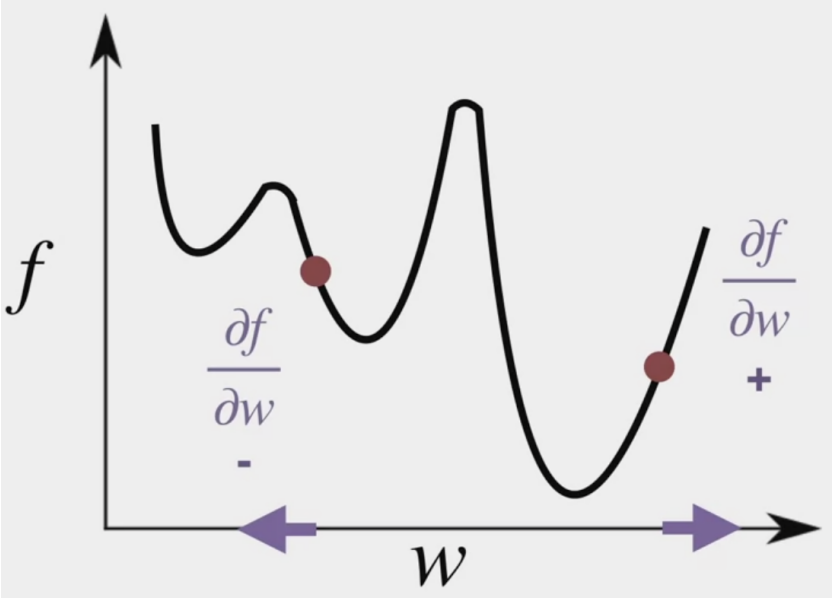
- By assumption we have far more **states** than **weights**, so making one state's estimate more accurate invariably means making others' less accurate.

- We are obligated then to say which states we care most about → We must specify a **state distribution** $\mu(s) > 0, \sum_s mu(s) = 1$, representing how much we care about the error in each state s. The states that the policy spends more time in have a higher weight in the objective. We care less about errors in the states the policy visits less frequently.

9.3 Stochastic-gradient and Semi-gradient Methods (for $\overline{VE}$ minimization)

9.3.1 Stochastic-gradient Methods

Recall Derivative: tells us how to locally change W to increase or decrease f.



- **Sign** $(+/-)$**:** indicates the direction to change W to increase f.
- **Magnitude:** the slope of the function f at the point W.

Gradient: Derivative on Multiple Dimensions. So it tells us how f changes as the vector W changes.

- If f is privatized by more than one variable, then W is a vector. The gradient is a vector of partial derivatives, indicating how a local change in each component of W affects the function f.

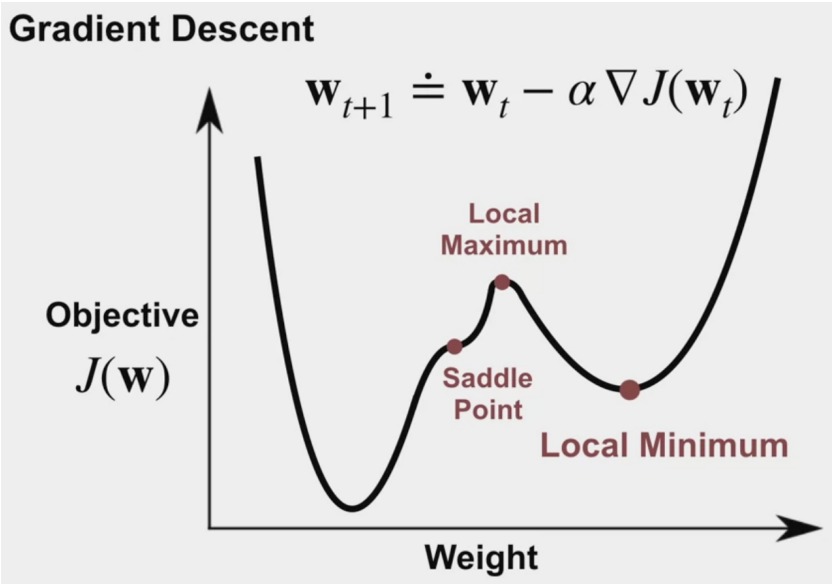
$$w \doteq \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_d \end{bmatrix} \quad \nabla f \doteq \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \dots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$$

  - The sign of each component of the gradient specifies the direction to change the associated component of W, in order to increase f. The magnitude of the component specifies how quickly f changes, as W moves in that direction.
  - The gradient gives the direction of steepest descent. It provides the direction to change W, so that locally f is maximally increased.

Mean Squared Value Error ($\overline{VE}(w)$) is a function of the weights to minimize with **Gradient Descent** **(used to find stationary points of objectives)**.

- Gradient Descent update rule:

$$w_{t+1} \doteq w_t - \underbrace{\alpha}_{\text{Step-Size}} \underbrace{\nabla f(w_t)}_{\text{Gradient}}$$



– Local minima is stable, so gradient descent will tend to converge there. However, for convex functions, gradient descent is guaranteed to converge to the global minimum, which is the best possible setting of the weights for the objective.

Gradient for Policy Evaluation:

- **(Standard) Gradient Descent:**

Gradient of the Mean Squared Value Error Objective

$$\begin{aligned}
&\nabla \sum_{s\in\mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2 \\
&= \sum_{s\in\mathcal{S}} \mu(s)\,\nabla[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2 \\
&= -\sum_{s\in\mathcal{S}} \mu(s) 2[v_\pi(s) - \hat{v}(s,\mathbf{w})]\,\nabla\hat{v}(s,\mathbf{w})
\end{aligned}$$

$$\hat{v}(s,\mathbf{w}) \doteq <\mathbf{w},\mathbf{x}(s)>$$

$$\nabla\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)$$

  – The gradient of the value function approximation indicates how to change the weights to increase the value for that state.
  – Computing the gradient for $\overline{VE}$ requires summing over all states. This is generally not feasible for large state spaces. Also, we likely do not know the distribution μ.

- **Stochastic Gradient Descent: only uses a stochastic estimate of the gradient (to approximate this gradient).**
    * expectation of each stochastic gradient = gradient of the objective
  – Though we do not explicitly have μ, we can sample states from it simply by following the policy. We can use the $(S, v_\pi(S))$ pairs to make an update to decrease the error on states.
  – **By making small updates in the direction that improves error on each $(S, v_\pi(S))$ pair**, we might sometimes increase the error on the full objective. But the overall trend will be to make progress for the full objective.
  – One remaining practical issue: we do not have access to v_π.

- **Gradient Monte Carlo for value estimation:**
  - We can get rid of v_π from our update rule.
  - Using samples of the return for each visited state, S_t, as we did for Monte Carlo methods.

Gradient Monte Carlo

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[\ G_t\ - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(s, \mathbf{w}_t)$$

Recall that

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right]$$

$$\mathbb{E}_\pi\left[2[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})\right]$$

$$= \mathbb{E}_\pi\left[2[\ G_t\ - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})\right]$$

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

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

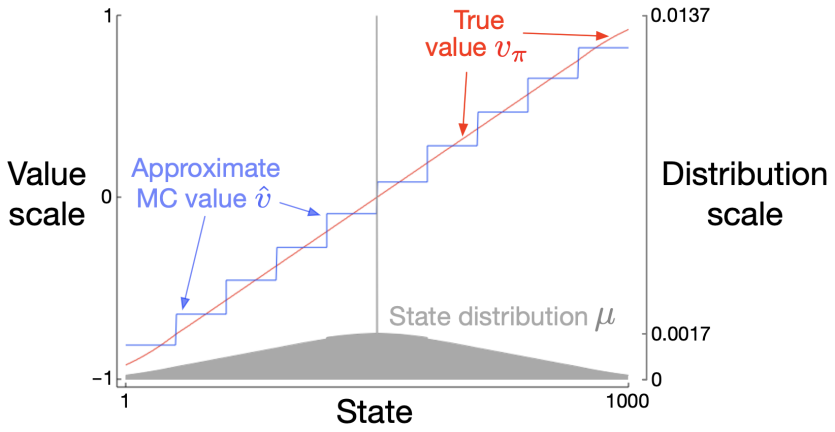
Generate an episode using π

Loop for each step of episode, $t = 0, 1, ..., T-1$**:**

$$w \leftarrow w + \alpha(G_t - \hat{v}(S_t, w))\nabla \hat{v}(S_t, w)$$

Example: Gradient Monte Carlo with State Aggregation

- Random walk starting from 500, terminating in either 0 (w. return -1) or 1 (w. return +1).
- With state aggregation, aggregate the 1000 states to 10 weights.



9.3.2 Semi-gradient TD

What: TD learning with function approximation (approximation to stochastic gradient descent).

- *Semi-gradient method*, as TD is not performing gradient descent updates on the squared error. It's not a gradient descent algorithm.

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

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

Initialize S

Loop for each step of episode:

Choose $A \sim \pi(\cdot|S)$

Take action A, observe R, S'

$w \leftarrow w + \alpha((R + \gamma\hat{v}(S', w)) - \hat{v}(S, w))\nabla\hat{v}(S, w)$

$S \leftarrow S'$

Until S is terminal

We can denote approximation of $v_\pi(S_t)$ with U_t. This yields the following general SGD update for state-value prediction:

$$w_{t+1} \doteq w_t + \alpha(U_t - \hat{v}(S_t, w_t))\nabla\hat{v}(S_t, w_t)$$

Semi-gradient TD vs Gradient Monte Carlo:

- **Semi-gradient TD:** biased TD updates → cannot guarantee convergence to a local minimum. But we often prefer TD learning over Monte Carlo anyway because it can converge more quickly, due to taking advantage of bootstrapping.
- **Gradient Monte Carlo:** converge to a local minimum

9.4 Linear Methods

What: Linear value function approximation is a strict generalization of tabular value function approximation.

Benefits:

- Linear methods are simpler to understand and analyze mathematically.
- Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation.
- Much of the theory of TD learning is for the linear setting.

Linear methods approximate the state-value function by the inner product between w and $x(s)$:

$$\hat{v}(s, w) \doteq w^T x(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

Where:

$x(s)$: *feature vector* representing state s

It is natural to use Stochastic Gradient Descent (SGD) updates with linear function approximation. The **gradient** of the approximate value function with respect to w in this case is

$$\nabla \hat{v}(s, w) = x(s)$$

Thus, in the linear case the general SGD update reduces to a particularly simple form:

$$w_{t+1} \doteq w_t + \alpha(U_t - \hat{v}(S_t, w_t))x(S_t)$$

Linear (Semi-gradient) TD

- **Converges** to a well understood approximation called the TD fixed point W_{TD}.
  - It is a solution to this linear system:

    $$W_{TD} = A^{-1}b$$

  - Converged when (Expected TD Update = 0)
- TD update with linear function approximation: Expected TD Update. It characterizes the expected change in the weight from one time step to the next.

  $$\mathbb{E}[\Delta W_{TD}] = w_t + \alpha(b - AW_{TD})$$

  - **Where:**
    - **matrix** A is defined in terms of an expectation over the features

      $$A \doteq \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T]$$

    - **vector** b vector b is defined in terms of the reward and the features

      $$b \doteq \mathbb{E}[R_{t+1}x_t]$$

- **Key takeaway** is that even though TD does not converge to the minimum of the mean squared value error, It does converge to the minimum of a principled objective, based on Bellman equations.

9.5 Feature Construction for Linear Methods

Here we discuss different ways to construct state, and state-action features.

Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems.

A **limitation of the linear** form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j.

9.5.1 Polynomials

While the basic polynomial features we discuss here do not work as well as other types of features in RL, they serve as a good introduction because they are simple and familiar.

9.5.2 Fourier Basis

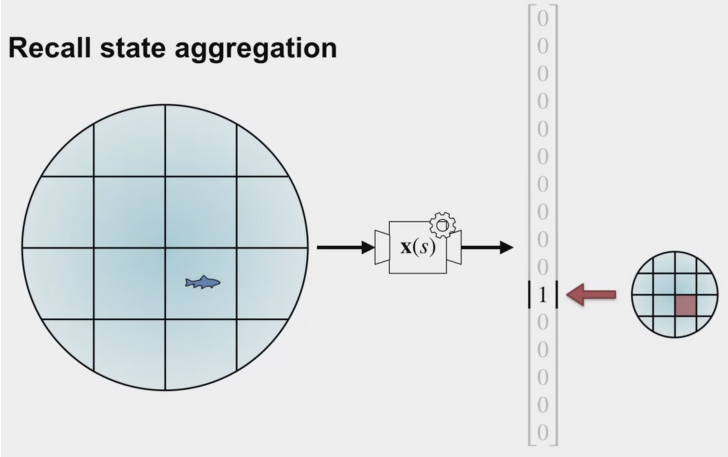
Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies.

In RL, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of RL problems.

9.5.3 Coarse Coding

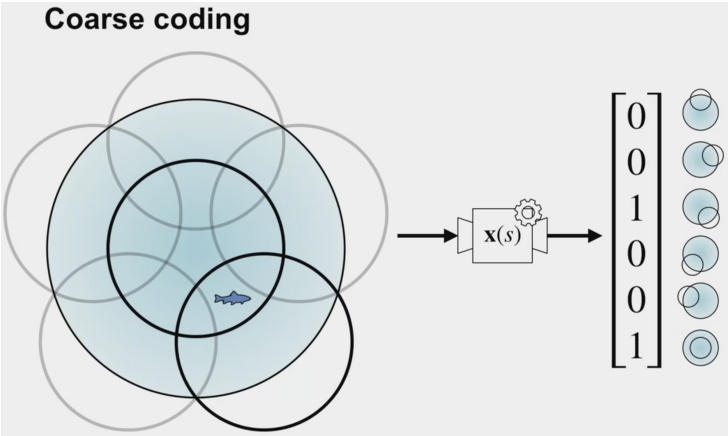
Recall, **state aggregation:** is the process of aggregating states for a smaller number of weights. Similar states are aggregated together.

- **Example:** represent the location of this fish that's swimming in a pond, a two-dimensional continuous state space:



Course coding is a generalization of state aggregation that allows shapes to overlap. By allowing the shapes to overlap, we obtain a more flexible class of **feature representations** called course coding.

- **Example:** represent the location of this fish that's swimming in a pond, a two-dimensional continuous state space:



Generalization Properties of Coarse Coding:

- If the union of the receptive fields for the active features is large, the feature representation generalizes more.
- Using different shapes in coarse coding can change the direction of generalization as well.

- The overlap between shapes dictates the level of discrimination.
- Every state within the same shape will have the exact same feature vector. As a result, they must all have the same approximate value.
- size, number, and shape of the features all affect the **discriminative ability** of the representation.

9.5.4 Tile Coding

What: a computationally efficient way to perform Course Coding.

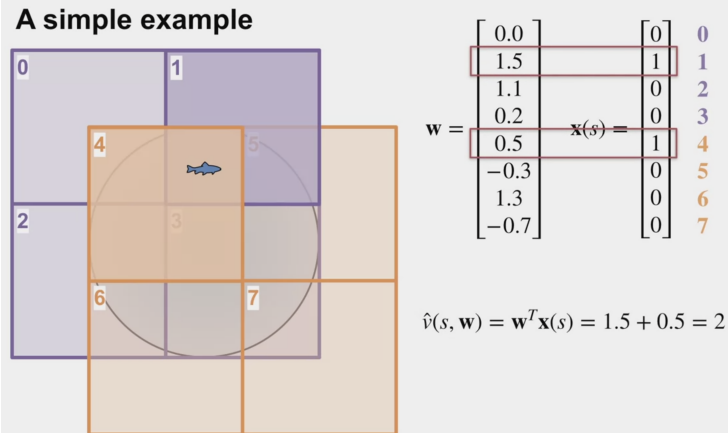
How: performs an exhaustive partition of the state space using overlapping grids → many small intersections → better discrimination.

- Put several tilings on top of each other, with each tiling is offset by a small amount.
- For:
  - **one tiling**, generalization only occurs within the square
  - **multiple tilings**, updates in a state generalize to other intersecting states

layering squares over this squished environment is like laying rectangles over the unsquished environment. By using rectangles, we can control the broadness of the generalization across each dimension of the state-space.

Good in low dimensional environments. However, as the number of dimensions grows, the number of required tiles grows exponentially. As a result, it can be necessary to tile input dimension separately.

- **Example:** represent the location of this fish that's swimming in a pond, a two-dimensional continuous state space:



9.5.5 Radial Basis Functions (RBFs)

What: natural generalization of coarse coding to continuous-valued features.

How: Rather than each **feature** being either 0 or 1, it can be anything in the interval [0, 1], reflecting various *degrees* to which the feature is present.

A **typical RBF feature**, x_i, has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s, and the feature's prototypical or center state, c_i, and relative to the feature's width, σ_i:

$$x_i(s) \doteq \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right)$$

The **norm** or **distance metric** of course can be chosen in whatever way seems most appropriate to the states and task at hand. The figure below shows a one-dimensional example with a Euclidean distance metric.

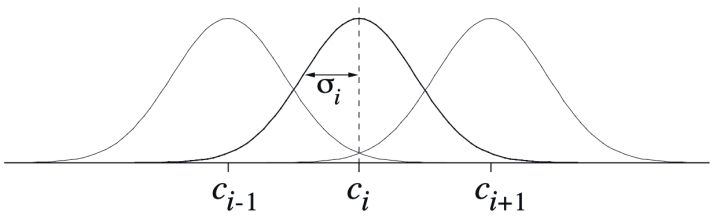


Figure 20: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance.

9.6 Selecting Step-Size Parameters Manually

Most SGD methods require the designer to select an appropriate step-size parameter α. Ideally selecting the step-size parameter α would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually.

The classical choice:

$$\alpha_t = 1/t$$

- produces sample averages in tabular MC methods
- not appropriate for TD methods, for nonstationary problems, or for any method using function approximation.

In tabular case

- $\alpha = 1$ will result in a complete elimination of the sample error after one target (we usually want to learn slower than this).
- a step size of $\alpha = \frac{1}{10}$ would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use $\alpha = \frac{1}{100}$.
- In general, if $\alpha = \frac{1}{\tau}$, then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about τ experiences with the state.

With general function approximation there is not such a clear notion of number of experiences with a state, as each state may be similar to and dissimilar from all the others to various degrees. However, there is a similar rule that gives similar behavior in the case of linear function approximation. Suppose you wanted to learn in about τ experiences with substantially the same

feature vector. A good rule of thumb for setting the step-size parameter of linear SGD methods is then

$$\alpha \doteq (\tau \mathbb{E}[x^T x])^{-1}$$

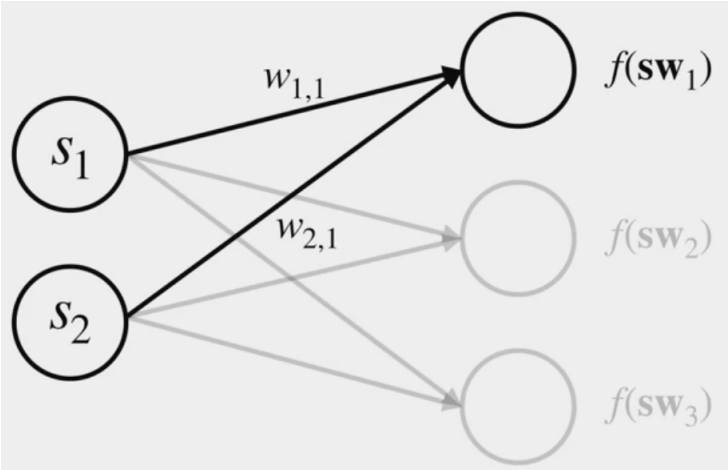
where x is a random feature vector chosen from the same distribution as input vectors will be in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally $x^T x$ is a constant.

9.7 Nonlinear Function Approximation: Artificial Neural Networks

What: a flexible and powerful class of nonlinear function approximators.

Why: Neural networks provide a strategy for learning a useful set of features.

Feed forward pass of a neural network mathematically (e.g.):



- At **each node** we have two vectors:
  - Inputs: $s = [s_1, s_2]$
  - Weights for each input: $w_1 = [w_{1,1}, w_{2,1}]^T$
- At **each node** we dot product the weights w_1 with the input s and pass the result through the activation function $f(s \times w_t)$.

- **Each node** has a different set of weights → produce a different output, (which we call a *feature*).

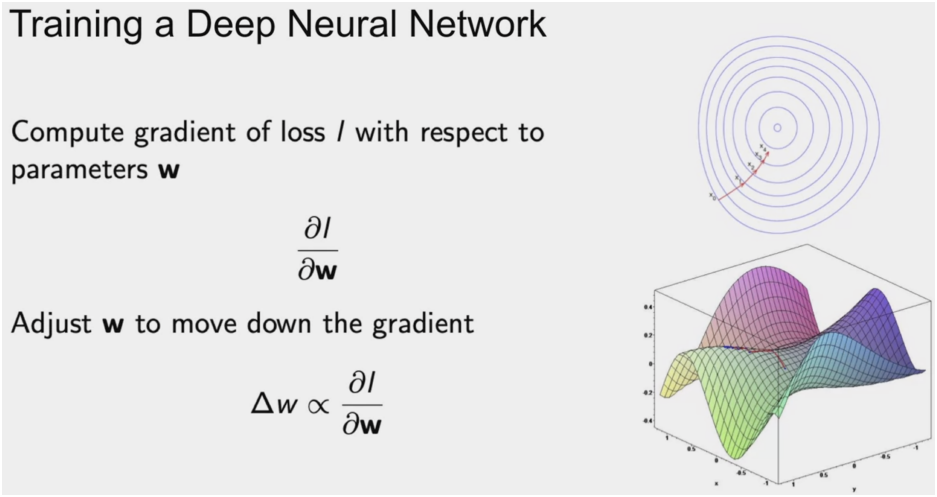
So neural networks can be viewed as a way to construct features, and that neural networks are non-linear functions of state.

Deep Neural Networks:

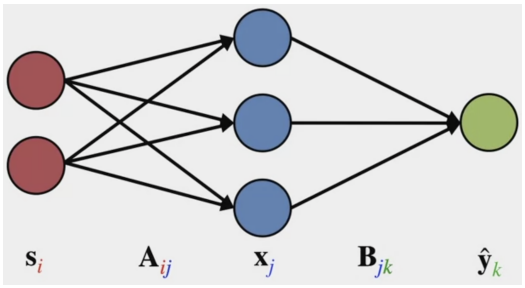
- **Why:** depth can facilitate learning features **through** composition and abstraction.
- **Challenge:** In practice, picking the right architecture can be difficult and can significantly impact performance.
- **Universal Approximation Property:** in theory, a neural network need not be deep. A neural network with a single hidden layer can approximate any continuous function given that is sufficiently wide.
- **Depth** = # of hiddel layers in the network
  - allows composition of features → can produce more specialized features by combining modular components.
  - can also be helpful for obtaining abstractions. Deep neural networks compose many layers of lower-level abstractions with each successive layer contributing to increasingly abstract representations.
    - For example, a network can be designed with a **bottleneck layer**. The idea is simple. Each successive layer contains less nodes than the layer before. The representation is the layer with the fewest nodes and contains the key details needed for prediction.

Backpropagation: (Gradient Descent for Training Neural Networks)

- **In a nutshell:**



- **Notation:**



  - **weights (matrix)** A linearly weight s to produce the features x
  - **weights (matrix)** B linearly weight x to produce the output $\hat{y}$
  - **(output of hidden layer)** $x \doteq f_A(\psi)$
    - $\psi \doteq sA$
  - **(final output)** $\hat{y} \doteq f_B(\theta)$
    - $\theta \doteq xB$

- The errors δB from the output of the network are propagated backwards to this earlier layer to help determine the role A had in producing that error (δA).

- **Full derivation (deriving the gradient):**

  - **Gradient for B:**
    1. Take the partial derivative **of the loss function** $L(\hat{y}_k, y_k)$ **with respect to** the first set of **weights** B: $\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}}$

* We use the chain rule given the derivative of L with respect to $\hat{y}$ **times** the derivative of $\hat{y}$ with respect to B:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial B_{jk}}$$

2. Use the chain rule again for this: $\frac{\partial \hat{y}_k}{\partial B_{jk}}$ derivative:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot \frac{\partial f_B(\theta_k)}{\partial \theta_k} \cdot \frac{\partial \theta_k}{\partial B_{jk}}$$

* Because θ is a linear function of B, and because x does not depend on B:

$$\frac{\partial \theta_k}{\partial B_{jk}} = x_j$$

* So general equation for B:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot \frac{\partial f_B(\theta_k)}{\partial \theta_k} \cdot x_j$$

* To ease notation while computing the gradient with respect to A, let's define a new term δ^B:

$$\delta_k^B \doteq \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot \frac{\partial f_B(\theta_k)}{\partial \theta_k}$$

- **Gradient for A:** The main difference is that we have one extra chain rule step because the weights A also affect x.

1. Take the partial derivative **of** the **loss function** $L(\hat{y}_k, y_k)$ **with respect to** the second set of **weights** A, combined with the previous chain rule step δ_k^B: $\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}}$
   - We use the chain rule:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B \cdot \frac{\partial \theta_k}{\partial A_{ij}}$$

2. Use the chain rule again for this: $\frac{\partial \theta_k}{\partial A_{ij}}$ derivative:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B \cdot B_{jk} \cdot \frac{\partial x_j}{\partial A_{ij}}$$

3. Using the chain rule again for $\frac{\partial x_j}{\partial A_{ij}}$, we get the derivative of $f_A(\psi)$ with respect to ψ **times** the derivative of ψ with respect to A.

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B \cdot B_{jk} \cdot \frac{\partial f_A(\psi_j)}{\partial \psi_j} \cdot \frac{\partial \psi_j}{\partial A_{ij}}$$

   - Because $\psi = sA$, we get:

$$\frac{\partial \psi_j}{\partial A_{ij}} = s_i$$

   - So general equation for A:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B \cdot B_{jk} \cdot \frac{\partial f_A(\psi_j)}{\partial \psi_j} \cdot s_i$$

   - We can clean up this derivative by again, defining a term δ^A.

$$\delta_j^A = (B_{jk}\delta_k^B)\frac{\partial f_A(\psi_j)}{\partial \psi_j}$$

- **Both gradients A and B can now be rewritten in a similar form:**

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_j^A s_i$$

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \delta_k^B x_j$$

   - They have a term δ that contains an error signal times their input.

Backpropagation Algorithm (w. SGD)

Gor each datapoint (s, y) in dataset D:

Compute delta for B: $\delta_k^B \doteq \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot \frac{\partial f_B(\theta_k)}{\partial \theta_k}$

Compute gradient for B: $\nabla_B^{jk} = \delta_k^B x_j$

$B \leftarrow B - \alpha_B \nabla_B$

Compute delta for A: $\delta_j^A = (B_{jk}\delta_k^B)\frac{\partial f_A(\psi_j)}{\partial \psi_j}$

Compute gradient for A: $\nabla_A^{ij} = \delta_j^A s_i$

$A \leftarrow A - \alpha_A \nabla_A$

Here we compute δ^A using δ^B. Notice, that by computing the gradients of the end of the network first, we avoid recomputing the same terms for A, that were already computed for δ^B. In fact, this is the main idea behind back propagation. It is simply gradient descent with this efficient strategy to compute gradients.

This derivation and algorithm easily extend to deeper networks. The δ for the earlier layer is similarly computed recursively using the δ in the next layer.

Optimization Strategies for NNs

- **Weight Initialization:**
  - **Motivation:** Choice of w starting points can play a big role in the performance of the neural network.
  - **Common Strategies:**
    - **Random Sample**: One simple yet effective initialization strategy, is to randomly sample the initial weights from a normal distribution with small variance.

$$w_{init} \sim \mathcal{N}(0, 1)$$

      - This way, each neuron has a different output from other neurons within its layer. This provides a more diverse set of potential features. By keeping the variants small, we ensure that the output of each neuron is within the same range as its neighbors.
      - One downside to this strategy is that, as we add more inputs to a neuron, the variance of the output grows.
    - **Scaled Random Sample**: gets around the variance issue of 'Random Sample' by scaling the variance of the weights, by one over the square root of the number of inputs.

$$w_{init} \sim \frac{\mathcal{N}(0, 1)}{\sqrt{n_{inputs}}}$$

- **More sophisticated update mechanisms:**
  - **Motivation:** More sophisticated update mechanisms can speed up neural network optimization.

- **Common Strategies:**
  - **Momentum (heavy-ball method)**: similar to the regular stochastic gradient descent update plus an extra term called the momentum M.

$$w_{t+1} \leftarrow w_t - a\nabla_w L(w_t) + \lambda M_t \tag{1}$$
$$M_{t+1} \leftarrow \lambda M_t - \alpha\nabla_w L \tag{2}$$

    - The momentum term summarizes the history of the gradients using a decaying sum of gradients with decay rate Lambda.
      - If recent gradients have all been in similar directions, then we gained momentum in that direction. This means, we make a large step in that direction.
      - If recent updates have conflicting directions, then it kills the momentum. The momentum term will have little impact on the update and we will make a regular gradient descent step.
  - **Vector step-size adaption**: uses a separate step size for each weight in the network.
    - Global scalar step size is problematic because it can result in updates that are too big for some weights and too small for other weights.

10 On-policy Control with Approximation

In this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, w) \approx q_*(s, a)$, where $w \in \mathbb{R}^d$ is a finite-dimensional weight vector. So the feature representation has to represent actions as well.

With **discrete action spaces** we can make feature representation to represent actions as well by have a separate function approximator for each action. This can be accomplished by stacking the features.

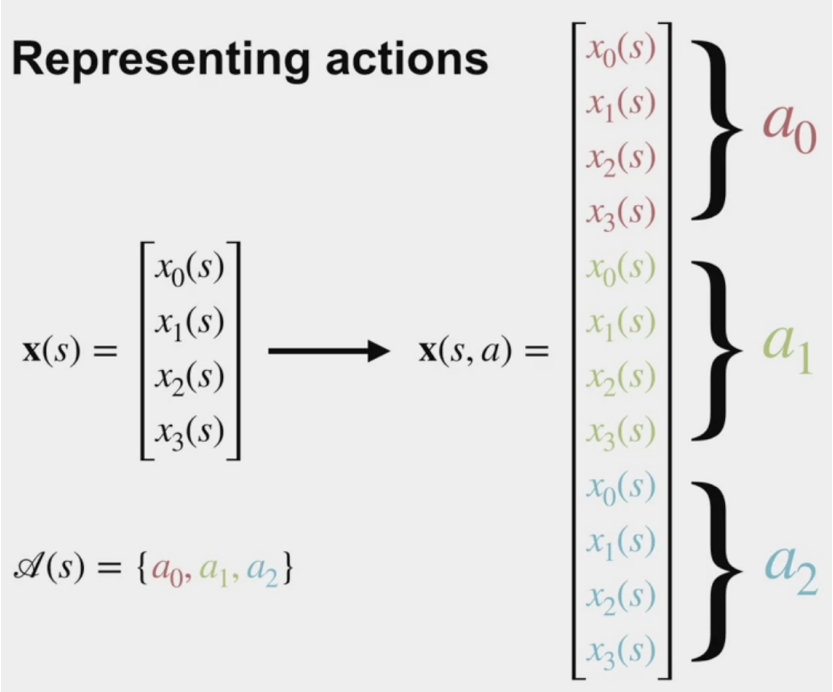


Figure 21: Example stacked feature representation for 4 features (representing states), and 3 actions.

The action values are then the dot products between each segment of the weight vector and the feature vector.

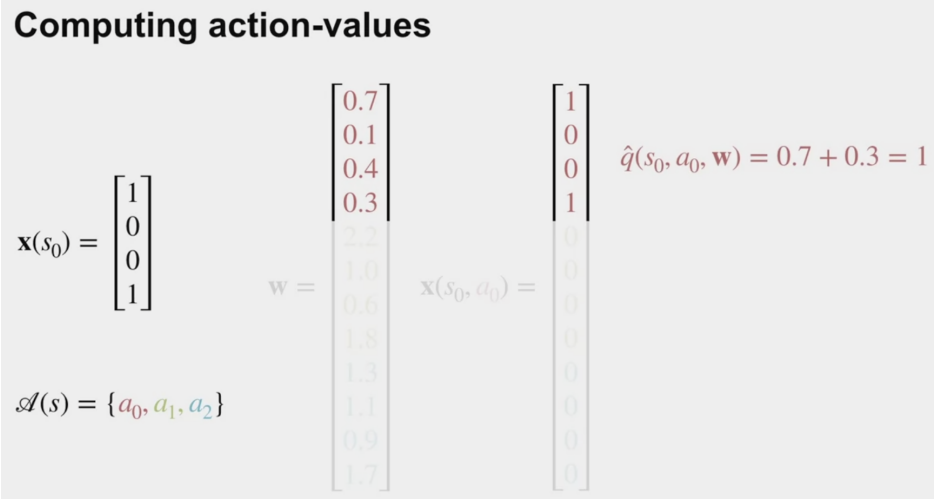
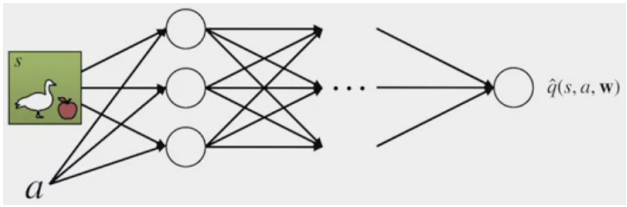


Figure 22: Example of calculating action-value for s_0 and a_0.

For **continuous action spaces** or if you want to **generalize over actions**, the action can be passed as an input like any other state variable.

For NNs we would input both the state and the action to the network. There would only be one output. The approximate action value for that state and action.



10.1 Episodic Sarsa with Function Approximation

Difference to tabular SARSA: We **use** parameterized action value functions $\hat{q}(s, a, w)$ **for** the action-value estimates. The update also changes to use the gradient to update the weights similar to semi-gradient TD.

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: $\alpha > 0$ & small $\epsilon > 0$
Initialize: value-function weights $w \in \mathbb{R}^d$ arbitrarily (e.g., $w = 0$)

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:

$w \leftarrow w + \alpha(R - \hat{q}(S, A, w))\nabla\hat{q}(S, A, w)$

Go to next episode

Choose A' as a function of $\hat{q}(S', \cdot, w)$ (e.g. ϵ-greedy)

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

$S \leftarrow S'; A \leftarrow A'$

10.2 Expected Sarsa with Function Approximation

Sarsa into expected Sarsa:

- Recall Sarsa with function approximation:

$$w \leftarrow w + \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w))\nabla\hat{q}(S_t, A_t, w)$$

- Expected Sarsa with function approximation:

$$w \leftarrow w+\alpha(R_{t+1}+\gamma\sum_{a'}\pi(a'|S_{t+1})\hat{q}(S_{t+1}, a', w)-\hat{q}(S_t, A_t, w))\nabla\hat{q}(S_t, A_t, w)$$

  - We compute the action values from our weight vector for every action in the next state. Then we compute this expectation under the target policy.

10.3 Exploration under Function Approximation

Optimistic initial values → initializing the weights such that the resulting values are optimistic (largest possible return).

- Can be difficult in complicated functions like neural networks.
- Depending on how our features generalize, optimistic initial values may not result in the same kind of systematic exploration we see in the tabular case.

ϵ-greedy is generally applicable and easy to use even in cases with non-linear function approximation.The only thing Epsilon greedy needs are the action value estimates, independent of how they are initialized or approximated.

- However, Epsilon greedy is not a directed exploration method. It relies on randomness to discover better actions near states followed by the current policy. It is therefore not as systematic as exploration methods that rely on optimism.

10.4 Average Reward: A New Problem Setting for Continuing Tasks

What: Average Reward Formulation, a new way of formulating continuing problems.

Why: In continuing tasks, we might be interested in extremely long horizon performance.

How: it cares equally about nearby and distant rewards.

- **Average Reward Objective:**

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

- More generally using the state visitation, μ:

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

– This inner term is the expected reward in a state under policy π. The outer sum takes the expectation over how frequently the policy is in that state. Together, we get the expected reward across states (average reward for a policy).

Deciding which actions from a state are better? What we need are action values for this new setting?

- **Differential Returns:** In the average reward setting, returns are **defined** in terms of differences between rewards R and the average reward $r(\pi)$.

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

- **represents** how much more reward the agent will receive from the current state and action compared to the average reward of the policy.
- The differential return can only be used to compare actions if the same policy is followed on subsequent time steps. To compare policies, their average reward should be used instead.
- Here $q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ captures how much more reward the agent will get by starting in a particular state than it would get on average over all states if it followed a fixed policy.

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)(r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a'))$$

11 *Off-policy Methods with Approximation

The extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning.

In this chapter we explore the convergence problems, take a closer look at the theory of linear function approximation, introduce a notion of learnability, and then discuss new algorithms with stronger convergence guarantees for the off-policy case. In the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning.

11.1 Q-learning with Function Approximation

Q-learning is a special case of expected Sarsa

- Recall, Expected Sarsa with function approximation:

$$w \leftarrow w+\alpha(R_{t+1}+\gamma\sum_{a'}\pi(a'|S_{t+1})\hat{q}(S_{t+1},a',w)-\hat{q}(S_t,A_t,w))\nabla\hat{q}(S_t,A_t,w)$$

- Q-learning with function approximation:

$$w \leftarrow w+\alpha(R_{t+1}+\gamma\max_{a'}\hat{q}(S_{t+1},a',w)-\hat{q}(S_t,A_t,w))\nabla\hat{q}(S_t,A_t,w)$$

  - In Q-learning, the target policy is greedy with respect to the approximate action values. Computing the expectation under greedy policy is the same as computing the maximum action value.