Note: Reinforcement Learning - An Introduction

Yanqing Wu

Update: September 20, 2022

Contents

1 Chap 1: Introduction

1.1 Reinforcement Learning

RL is learning what to do, so as to maximize a numerical reward signal.

Agent-oriented learning - learning by interacting with an environment to **achieve a goal**. Learning by trial and error, with only **delayed evaluative feedback (reward)**. The kind of ML like natural learning. Learning can tell for itself when it's right/wrong.

RL means taking optimal action with **long term** results or **cumulative rewards** in mind.

Two most important distinguishing features of RL: trial-and-error search and delayed reward.

RL methods generally incorporate 3 aspects: sensation, action and goal.

Definition 1.1 ***Supervised Learning*** *is learning from a training set of labeled examples provided by a knowledgable external supervisor.*

- Example: a description of a situation
- Label: a specification of the correct action the system should take in a situation, which is often to identify a category to which the situation belongs
- Object of Supervised Learning: for the system to extrapolate (generalize) its responses s.t. it acts correctly in unseen situations
- Con: inadequate for learning from *interaction*, because it is (often) impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act

Definition 1.2 ***Unsupervised Learning*** *is about finding structure hidden in collections of unlabeled data.*

RL is trying to maximize a reward signal, while Unsupervised Learning is trying to find hidden structure.

Key feature 1: the exploration-exploitation dilemma is one of RL's challenges. Exploitation: to obtain reward from experienced. Exploration: to find better actions.

Key feature 2: RL explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment.

1.2 Examples

1.3 Elements of RL

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

Definition 1.3 ***Policy****, defines the learning agent's way of behaving at a given time.*

A policy is a mapping from perceived states of the environment, to actions to be taken, when in those states. A policy may be a simple function or a lookup table, or complex computation (e.g. a search process). Policies may be stochastic, specifying probabilities for each action.

Definition 1.4 ***Reward signal****, defines the goal of a RL problem.*

On each timestep, the environment sends a single number (i.e. *reward*) to the RL agent. The agent's only objective is to maximize the total reward it receives over the long run.

Definition 1.5 ***Value function****, specifies what is good in the long run.*

The *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

!! *Rewards* determine the immediate desirability of states, *values* determines the long-term desirability of states.

State Value function = Expected Return = Discounted sum of all rewards

Definition 1.6 ***Model**, mimics the behavior of the environment.*

The model defines the reward function and transition probabilities. Models are used for *planning*. Model-based methods: use models and planning to solve RL problems. Model-free methods: explicitly trial-and-error learner (almost the opposite of planning).

Modern RL spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

1.4 Limitations and Scope

1.5 An Extended Example: Tic-Tac-Toe

Minimax algorithm wik (2021) (from game theory).

Classical optimization methods are for sequential decision problems (e.g. dynamic programming). We need to estimate an approximate opponent model so to use dynamic programming; in such case, this is similar to RL methods.

During playing, we need to adjust the values of the states as to make more accurate estimates of the probabilities of winning. The current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state's value a fraction of the way toward the value of the later state.

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

S_t, the state before a move. S_{t+1}, the state after the move. $V(S_t)$, the update to the estimated value of S_t. α, step-size parameter, a small positive fraction, influences the rate of learning.

This update rule is an example of a temporal-difference (TD) learning method. TD means the changes are based on a difference between estimates at two consecutive times.

If α is reduced properly over time, then this method converges (for any fixed opponent, it converges to the true probabilities of winning for each state); if α is not reduced to zero over time, then the agent "also plays well against opponents that slowly change their way of playing" (?). My understanding is that, there must have some slow changes, whether the parameter or the opponent model.

Evolutionary methods vs. Value Function methods. Evolutionary methods ignores what happens during the games, and only the final outcome of each game is used (e.g. all of its behavior in a winning game is given credit). In contrast, value function methods evaluate individual states.

In this example, learning started with no prior knowledge except the rules of the game. In reality, prior information can be incorporated into RL in many ways for efficient learning. RL can be used when we have access to true state, or where some states are hidden, or when different states appear to be the same to the learner.

The player need a model of the game to see the result states in response to actions. While model-free systems cannot know how their environments will change in response to actions. Model-free methods are building blocks for model-based methods. Sometimes, it is difficult to construct an accurate environment model, thus model-based methods cannot be used in such cases.

1.6 Summary

RL is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct **interaction** with its environment, without requiring exemplary supervision or complete models of the environment.

RL uses the formal framework of Markov decision processes (MDP) to defined the interaction between a learning agent and its environment in terms of states, actions, and rewards.

The concepts of value and value function are key to most of the RL methods in this book. The authors believe that value functions are critical for efficient search in policy space. The use of value functions distinguishes RL methods from evolutionary methods.

1.7 Early History of RL

Three threads: 1. learning by trial and error, originated in the psychology of animal learning. Time from 1850s to early 1980s' revival of RL; 2. optimal control problem and its solution using value functions and dynamic programming. This thread did not involve learning; 3. temporal-difference methods. These 3 threads came together in late 1980s. The TD and optimal control threads were fully brought together in 1989 with Chris Watkin's development of Q-learning.

The essential idea of trial-and-error learning: 'the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends.'

TD learning methods are driven by the difference between temporally successive estimates of the same quantity.

The authors developed a method for using TD learning + trial-and-error learning, known as the *actor-critic architecture*.

Part I: Tabular Solution Methods

Algorithms in this part are described in simplest forms: the state space and action space are small enough for the approximate value functions to be represented as arrays/tables. In this case, the methods can (often) find exact solutions (i.e. optimal value function and optimal policy). Algorithms in next part can only find approximate solutions but can be applied to much larger problems.

Bandit problems, RL problem with only a single state. Chap 2 describes general problem formulation (finite MDP) and its main ideas (including Bellman equations and value functions).

Three fundamental classes of methods for solving MDP problems: dynamic programming, Monte Carlo methods and TD learning.

- DP: well developed, but require a complete and accurate model of the environment
- MC: no model needed, but not good for incremental computation
- TD: no model needed and fully incremental, but are complex

2 Chap. 2 Multi-armed Bandits

- Evaluative: feedback depends solely on the taken action, indicating results
- Instructive: feedback is independent of the taken action, indicating correct actions
- Nonassociative setting: no learning involved in more than one situation; no need to associate different actions with different situations/states
- Associative setting: the best action depends on the situation

Figure 1: Three one-armed/level bandit machines

2.1 A k-armed Bandit Problem

k-armed bandit problem: You choose one action from k available actions (e.g. choose one of the slot machines). For each choice, you receive a numerical reward. The reward is given based on a probability distribution that specific to the choice (machine). Your objective is to maximize the expected total reward over j action selections (time steps).

Note that the bandit problem is formally equivalent to a one-state Markov decision process.

For now, let us define that

- *Value*: the expected/mean reward of a taken action
- A_t: action selected on time step t
- R_t: corresponding reward of A_t
- $q_*(a)$: $q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$. The (true) value of an arbitrary action a (i.e. the expected reward given a is selected)
- $Q_t(a)$: the *estimated* value of action a at time step t.

If we know each action values, then solving the *k*-armed bandit problem would be always select the action with highest $value$. If we don't know the action values, then we can estimate them. Our goal is to have $Q_t(a)$ to be as close to $q_*(a)$ as possible.

Definition 2.1 ***Greedy actions****, action(s) with highest* ***estimated*** *value.*

exploiting: select greedy action(s). *exploring*: select non-greedy action(s). Exploration may improve the estimate value of non-greedy actions.

2.2 Action-value Methods

Definition 2.2 ***Action-value Methods****, or Q function, methods for estimating the values of actions, and using the estimates to select actions.*

'Q' means the **quality** of an action.

Let see a simple action-value method: *sample-average method*, in which action-value is the mean reward for an action.

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_{i=a}}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_{i=a}}}$$

- $\mathbb{1}$: return 1 if predicate is true, otherwise 0
- in the case when denominator is 0 (which makes the fraction mathematically undefined), $Q_t(a)$ is set as default value (e.g. 0)

- in the case when denominator $\to \infty$, by the Law of Large Numbers, $Q_t(a)$ converges to $q_*(a)$

Definition 2.3 ***Greedy action selection***, *If there are more than one greedy action, select one of them arbitrarily.*

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

Definition 2.4 ***ε-greedy action selection*** *with probability ε, select an action randomly; with probability $1-\varepsilon$, select a greedy action*

2.3 The 10-armed Testbed

For Exercise 2.2, note that we should first convert the sequence of actions and rewards to $Q_t(a)$, then we can decide whether the selection is greedy or exploration based on the definition of ε-greedy action selection.

2.4 Incremental Implementation

Recall that estimated action value Q_n, using sample-average method, is calculated as:

$$Q_n \doteq \frac{R_1 + \cdots + R_{n-1}}{n-1}$$

We don't want to waste memory and computational resource for each new reward. Intuitively, to save memory, we can simplify the equation by first reconstruct the previous estimated action value Q_{n-1} and then evaluate Q_n with the new reward R_n. Therefore,

$$Q_{n+1} = \frac{Q_n \cdot (n-1) + R_n}{n} = \frac{Q_n \cdot n - Q_n + R_n}{n} = Q_n + \frac{1}{n}[R_n - Q_n]$$

The form $Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$ is an important form occurs in this book. The general form is

$$NewEstimate \leftarrow OldEstiamt + StepSize \underbrace{[Target - OldEstimate]}_{error}$$

A simple bandit algorithm

Initialize, for a = 1 to k :

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \arg\max_x Q(a) & \text{with prob. } 1-\varepsilon \text{ (breaking ties randomly)} \\ \text{a random action} & \text{with prob. } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

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

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

2.5 Tacking a Nonstationary Problem

Rewrite the update equation from last section and replace the stepsize with α, we have that

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

The first term tells us that contribution of Q_1 decreases exponentially with time, and the second term tells us the older rewards contribute exponentially less to the sum; i.e., the most recent rewards contribute most to our current estimate.

In previous section, the reward probability distribution is stationary (i.e. doesn't change over time). For nonstatoinary cases, let say we give more weight to recent rewards than to 'long-past' rewards; for doing so, one popular way is to use a constant α (step size).

Not all choices of $\alpha_n(a)$ guarantee convergence. In *stochastic approximation* *theory*, two conditions required to assure convergence "with probability 1"[1]:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty$$

$$\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

The 1st condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The 2nd condition is required to guarantee that the steps eventually become small enough to assure convergence.

Therefore, in sample-average method, $\alpha_n(a) = \frac{1}{n}$ converges, but a constant step-size parameter $\alpha_n(a) = \alpha$ does not converge.

Note that sequences of α that meet the conditions often converge very slowly and often need considerable tunning. These sequences of α often used in theoretical work, and are seldom used in real-world application.

2.6 Optimistic Initial Values

Higher (optimistic) initial values ($Q_i(a)$) encourages more *early* exploration. Because whichever actions are initially selected, the reward is less than the starting estimates; the agent then switches to other actions. This encourages more explorations early on.

However, this may only work well on stationary problems; as in nonstationary problems, some action values may change after certain timesteps. Another limitation is that we may not know what the optimistic initial value should be. Though a combination of exploration technique may be adequate in practice.

[1]i.e. *almost sure*. No distinction between 'sure' and 'almost sure' in finite sample space, but becomes important in infinite sample space. see https://en.wikipedia.org/wiki/Almost_surely

2.7 Upper-Confidence-Bound Action Selection

In ε-greedy action selection, it is not ideal to just randomly select actions; instead, we want to select among the non-greedy actions according to their potential for actually being optimal. Such taking into account both (1) how close their estimates are to being maximal, and (2) the uncertainties in those estimates.

Definition 2.5 *Upper-Confidence-Bound (UCB) Action Selection*

$$A_t \doteq \arg\max_a \left[\underbrace{Q_t(a)}_{\text{exploitation term}} + \underbrace{c\sqrt{\frac{ln\ t}{N_t(a)}}}_{\text{exploration term}} \right]$$

$N_t(a)$, denotes the number of times that action a has been selected prior to time t. If $N_t(a) = 0$, then a is considered to be a maximizing action. c, $c > 0$ (confidence level) controls the degree of exploration.

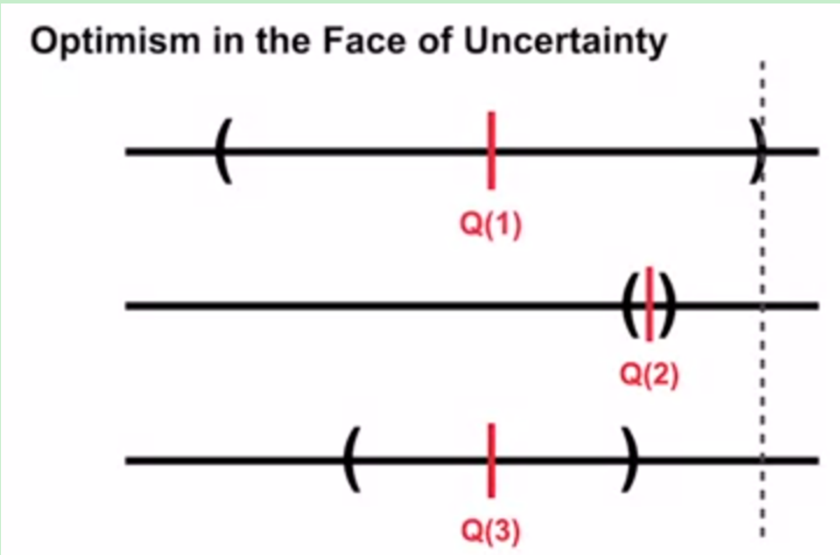


Figure 2: UCB Example with 3 actions. The interval represents the confidence level of the action. Larger confidence, smaller interval. Given the 3 actions, we select the one with highest upper bound. We update the interval based on received rewards.

The idea of UCB action selection is that the square-root term is a measure of the uncertainty/variance in the estimate of a's value. Each time a is selected, the uncertainty is (presumably) reduced as $N_t(a)$ increments. Each time a non-a is selected, t increments but $N_t(a)$ doesn't, thus the uncertainty estimate increases. All actions will eventually be selected, but actions with lower value estimates or that have been frequently selected, will be selected with decreasing frequency over time.

UCB is useful in bandit problems, but is (usually) not practical in general RL settings. It is difficult to deal with (1) nonstationary problems, and (2) large state spaces.

2.8 Gradient Bandit Algorithms

We have considered methods that (1) estimate actions values; (2) select actions based on estimations. We now consider a new approach for action selection: learning a numerical *preference* for each action *a*, which we denote $H_t(a) \in \mathbb{R}$. The larger the preference, the more often that action is taken, but the preference has no interpretation regarding rewards. Only the **relative** preference of one action over another is important. The action probabilities is determined by *soft-max distribution*.

$$P(A_t = a) \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$$

$\pi_t(a)$ is the probability of taking action a at time t. The initial probability are the same for all actions.

There is a natural learning algorithm in this setting based on the idea of stochastic gradient descent. This is based on updating the preference values as such, after taking action A_t and obtaining reward R_t

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \overline{R_t})(1 - \pi_t(A_t))$$

$$H_{t+1}(a) \doteq H_t(a) - \alpha(R_t - \overline{R_t})\pi_t(a) \quad \text{for all } a \neq sA_t$$

$\overline{R_t} \in \mathbb{R}$ is the average of the rewards up to but not including time t (with $\overline{R_1} = R_1$)

If the reward is higher than the baseline $\overline{R_t}$, then the probability of taking A_t in the future is increased; and if the reward is below baseline, then the probability is decreased. The non-selected actions move in the opposite direction.

The above update scheme is equivalent to **stochastic gradient ascent** with batch size 1.

2.9 Associative Search (Contextual Bandits)

For now, we have only considered nonassociative tasks, that is, tasks in which there is no need to associate different actions with different situations. However, in the general RL task, there is more than one situation/state, and the goal is to learn a policy, i.e., a mapping from states to optimal actions given the state.

Suppose when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values). For instance, if screen=red (actual reward is hidden from you), select arm 1; if screen=green, select arm 2... This is an example of an associative search task, it involves both trial-and-error learning to search for the best actions, and association of these actions with the situations in which they are best. This task is now called *contextual bandits* in the literature.

In this setting, we still don't have actions affecting the next situation as well as the reward. This will add another layer of complexity and lead us to full RL problems.

- (level 1) k-armed bandit problem: select one of k actions that have fixed probabilities. For each selected action, a reward is given. The goal is to maximize the total reward after j selections.

- (level 2) associative search task (contextual bandits): Same as k-armed bandit problem, the action affects only the immediate reward. Unlike k-armed bandits problem, what situations/state you are facing are told and it involves learning a policy.
- (level 3) full RL problem: On top of associative search task, actions affect next situation and reward.

2.10 Summary

This chapter presented several ways to balance exploration and exploitation:

- ε-greedy methods choose randomly among actions a small fraction of the time to encourage exploration
- UCB methods choose deterministically but favouring actions with uncertain estimates
- Gradient methods don't estimate action values, but preferences, and choose actions probabilistically according to the preferences
- Optimistic initialization sets optimistic initial Q values in order to encourage a very active exploration in the initial phase, leading to a fast convergence with no bias

There is no method among the above that is best. Common ways to assess performance of these kind of algorithms are through graphs:

- learning curve: shows the performance of an algorithm vs. iterations for a certain parameter setting
- parameter studies: summarize each algorithm by the average reward over the first 1000 steps, and show such a value for various parameter settings and algorithms.

One should also consider the sensitivity to parameter setting, that is an indication of **robustness**.

Despite the simplicity of methods presented in this chapter, they can be considered state of the art. More sophisticated methods usually impose additional complexity and assumptions.

Another approach to balancing exploration and exploitation in k-armed bandit problems is to compute a special kind of action value called *Gittins index*, which is an instance of Bayesian methods. This assume a known initial distribution over the (stationary) action values and then update the distribution exactly after each step. In general, the update computations are complex, but for *conjugate priors* they are easy. One way to select the best action at each time step is based on posterior probability. This method, sometimes called posterior sampling or Thompson sampling.

2.11 Learning Objectives (UA RL MOOC)

Lesson 1: The K-Armed Bandit Problem

1. Define reward

A number received for each taken action.

2. Understand the temporal nature of the bandit problem

Temporal means "related to time" in RL. From my understanding, the temporal nature of the bandit problem is that the decision of choosing action depends on which time steps in. For instance, on time step one, when the gambler does not have knowledge of the "value" of each machine, he needs to play a machine randomly. On time step two, he may play a different machine than on step one. After some rounds, the gambler has some value estimations of each machine; he then can play greedily with the highest value action. In general, the action selection strategy hinges on time steps, i.e., the action selection is temporal dependent.

3. Define k-armed bandit

Given k actions/choice, each with a numerical reward, each reward are given based on fixed probabilities. You need to do this selection in multiple rounds and your aim is to maximize the total reward.

4. Define action-values

values = action values = action value function

Action-value is the mean/expected reward of an action.

Lesson 2: What to Learn? Estimating Action Values

1. Define action-value estimation methods

$$Q(a) = \frac{\sum R}{\text{action count}}$$

2. Define exploration and exploitation

Exploration select random actions and exploitation select greedy action.

3. Select actions greedily using an action-value function

$$A = argmax(q_values)$$

4. Define online learning

Online learning is when the agent interacts with the environment. Offline learning is when the agent learns from fixed datasets

5. Understand a simple online sample-average action-value estimation method

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

6. Define the general online update equation

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

7. Understand why we might use a constant stepsize in the case of non-stationarity

Use a constant stepsize makes the agent able to adapt to changing environment. If we use 1/N(a) as stepsize in non-stationary cases, when the true values change suddenly in long steps, the 1/N(a) may be very small and can hardly learn the new values.

Lesson 3: Exploration vs. Exploitation Tradeoff

8. Define epsilon-greedy

Epsilon-greedy is used to balance exploitation and exploration. With small probability epsilon, we choose randomly among all actions (exploration), with 1-epsilon probability, we choose the highest value action (exploitation).

9. Compare the short-term benefits of exploitation and the long-term benefits of exploration

The short-term benefits of exploitation is we maximize the reward immediately, the long-term benefits of exploration is we have a better knowledge of the estimated values (i.e., Q(a)->q*(a)), thus giving us higher total rewards in the long run.

10. Understand optimistic initial values

We declare initial values way higher than their true values, thus the name optimistic. In such way, we are encouraged to try out all actions early on. When one action is selected, its value is certainly to drop (since we assign high initial value), allowing other non-touched actions to be selected.

11. Describe the benefits of optimistic initial values for early exploration

see 10

12. Explain the criticisms of optimistic initial values

Optimistic initial values may only well on stationary cases. In nonstationary cases, the advantage of optimistic dwindles as the step size increases. Another disadvantage is that we do not know how to select the "correct" optimistic initial values.

13. Describe the upper confidence bound action selection method

This is also a exploration-exploitation balancing technique. This encourages the agent to choose the action with highest UCB. The more an action is selected, the narrower the confidence interval of an action is, the less the explore-term of the action is.

14. Define optimism in the face of uncertainty

We always select the action with highest (most optimistic) UCB. When the action is selected, it either be good (the UCB largely stays) or bad (the UCB shrinks and enable other actions to be selected).

15. Different constant step size and decaying step size

A larger step size moves us more quickly toward the true value, but can make our estimated values oscillate around the expected value. A step size that reduces over time can converge to close to the expected value, without oscillating. On the other hand, such a decaying stepsize is not able to adapt to changes in the environment. Nonstationarity—and the related concept of partial observability—is a common feature of reinforcement learning problems and when learning online

3 Finite Markov Decision Processes

When you're presented with a problem in industry, the first and most important step is to translate that problem into a Markov Decision Process (MDP). The quality of your solution depends heavily on how well you do this translation.

Definition 3.1 *finite Markov Decision Processes, involves evaluative feedback (as in bandits) and associative settings (choosing different actions in different situations)*

MDPs are a classical formalization of *sequential* decision making, where actions influence both immediate rewards and subsequent states (thus influence future/delayed rewards as well). MDPs need to consider trade-off between immediate reward and delayed reward.

In bandit problems, we estimated the value $q_*(a)$ of each action a.

In MDPs, we estimate $q_*(s, a)$ (value of each action a in each state s), or $v_*(s)$ (value of each state given optimal action selections).

MDPs are an ideal form of the RL problem where precise theoretical statements can be made. Key elements of RL's mathematical structure: returns, value functions, and Bellman equations.

3.1 The Agent-Environment Interface

- agent: learner and decision maker
- environment: everything outside the agent, and which agent interacts with

At each time step t, the agent receives some representation of the environment's $state$, S_t, and on the state selects an $action$, A_t. One time step later, the agent receives reward R_{t+1} as a result of the action, and find itself in a new state S_{t+1}. The trajectory/sequence is:

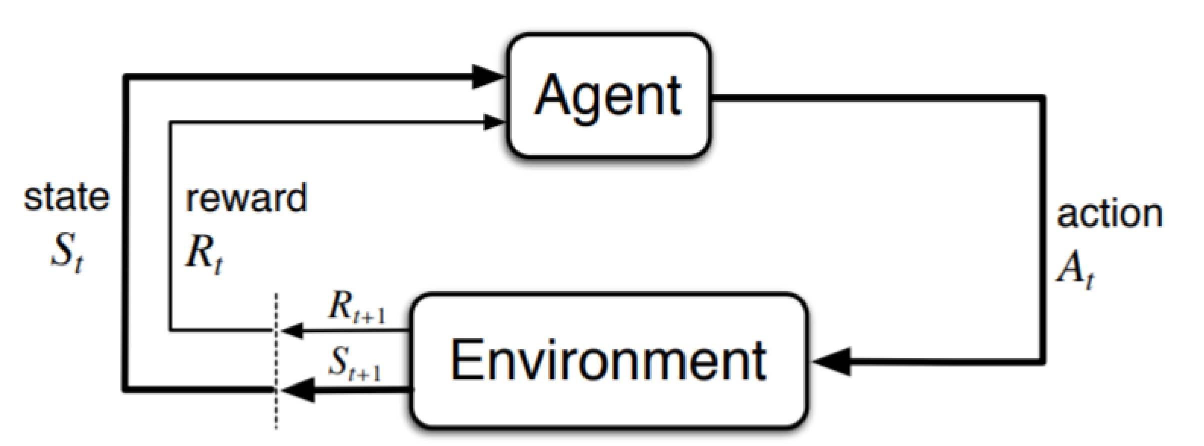


Figure 3: The agent-environment interaction in a MDP

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots$$

In a *finite* MDP, the state sets, action sets and reward sets are finite.

Following defines the *dynamics* of the MDP.

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

p specifies a probability distribution for each choice of s and a. The probability of each possible value for S_t and R_t depends on the immediately preceding state and action, S_{t-1} and A_{t-1} (note not including earlier states and actions).

Definition 3.2 *Markov property*[2]*, the future only depends on the current state, not the history.*

Definition 3.3 *state-transition probabilities,*

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

We can express state transition probabilities in an Matrix as shown in Figure 4. Each row in the matrix represents the probability from moving from our original state to any successor state. Sum of each row is equal to 1.

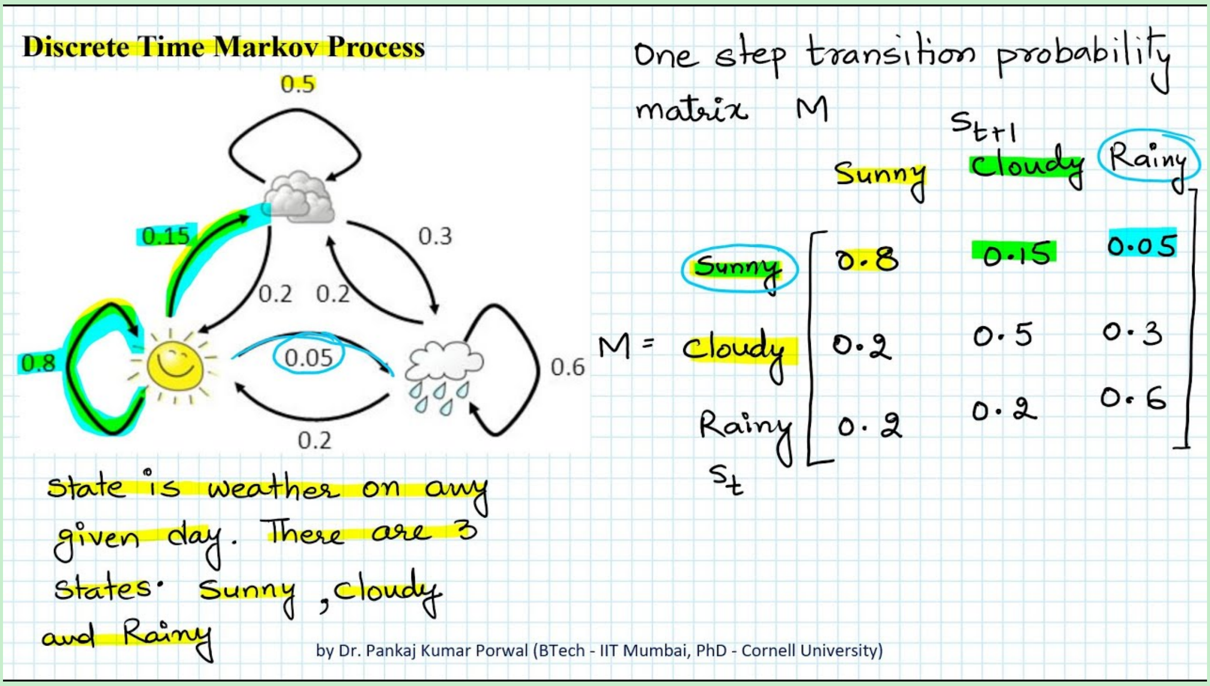
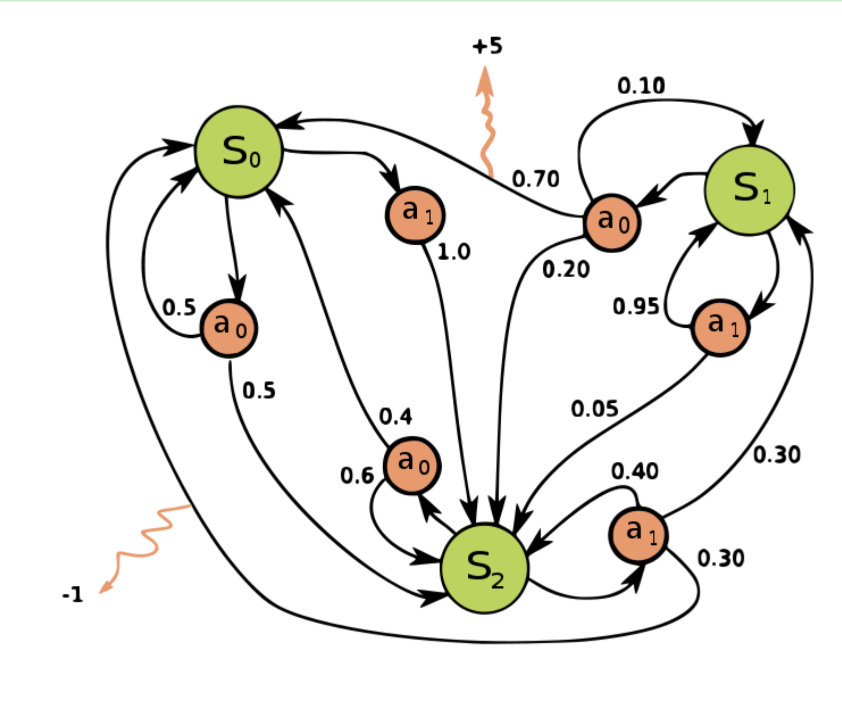


Figure 4: An example of state transition probability.



Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).

Figure 5: A complex example of state transition probability.

Definition 3.4 *expected rewards for state-action pairs*

[2]Markov property refers to the memoryless property of a stochastic process. "only present matters"

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

Definition 3.5 *expected rewards for state-action-nextState,*

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

Question: Why does the definition of the reward function $r(s,a,s')$ involve the term $p(s'|s,a)$? https://ai.stackexchange.com/q/20244/55308

Expectation of reward after taking action a in state s and ending up in state s' would be:

$$r(s'a,s') \doteq \sum_{r\in\mathcal{R}} r \cdot p(r|s,a,s')$$

Since we do not define $p(r|s,a,s')$, we have following equation with product rule:

$$p(s',r|s,a) = p(s'|s,a) \cdot p(r|s',s,a)$$

After moving $p(s'|s,a)$ to the LHS, we see that:

$$r(s,a,s') \doteq \sum_{r\in\mathcal{R}} r\frac{p(s',r|s,a)}{p(s'|s,a)}$$

agent-environment The agent-environment boundary represents the lit of the agent's absolute control, not of its knowledge. Anything that cannot be changed by the agent.

Any problem of learning goal-directed behavior can be reduced to 3 signals passing back and forth between an agent and its environment:

- action: the choice made by the agent
- the basis on which the choice is made
- the agent's goal

Example 3.1 *Bioreactor,*

- *actions (vector): target temperatures, target stirring rates*

- *states (vector): sensory readings, ingredients input, target chemical*
- *rewards (number): production rate of useful chemical (target)*

Example 3.2 *Robotic Arm (pick-and-place task)*

- *actions: voltages applied to each joint motor*
- *states: readings of joint angles and velocities*
- *rewards: +1 for successfully object pick-and-place; small negative reward at each time step (punish jerkiness motion)*

3.2 Goals and Rewards

In RL, the purpose/goal of the agent is to maximize *cumulative reward in the long run*. It is important that the set-up rewards indicate what we truly want accomplished. In particular, rewards are not the place to tell/impart agent prior knowledge about *how* to achieve our goal. For example, in a chess game, the agent should only be reward for winning, instead of achieving subgoals like taking opponent's pieces. Better places for imparting this kind of prior knowledge in (1) initial policy; or (2) initial value function.

The reward signal is a way of imparting to the agent *what* you want achieved, not *how* you want it achieved.

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

3.3 Returns and Episodes

A RL agent's goal is to maximize the cumulative reward it receives in the long run. In general, we seek to maximize the *expected return* G_t.

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

where T is a final time step.

The final time step marks the end of a subsequences.

Definition 3.6 *Episode, a subsequence of the agent-environment interaction. Each episode ends in* ***terminal state****, followed by a* ***reset*** *to a standard starting state or to a* ***sample*** *from a standard distribution of starting states. The next episode begins independently of how the previous one ended.*

The set of all nonterminal states, $\mathcal{S}$. All states and the terminal state, $\mathcal{S}^+$.

Definition 3.7 *Episodic tasks, tasks with episodes that with terminal state.*

Definition 3.8 *continuing tasks, tasks without terminal state.*

This task uses discounted rewards to deal with $T = \infty$.

Definition 3.9 *discounted rewards G_t. discount rate, $0 \leq \gamma \leq 1$*

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

The discount rate determines the present value of future rewards. As long as the reward sequence R_k is bounded, the infinite sum G_t has a finite value. If $\gamma \to 0$, the agent is myopic; if $\gamma \to 1$, the agent is farsighted.

If the reward is constant R, then we have the *geometric series*

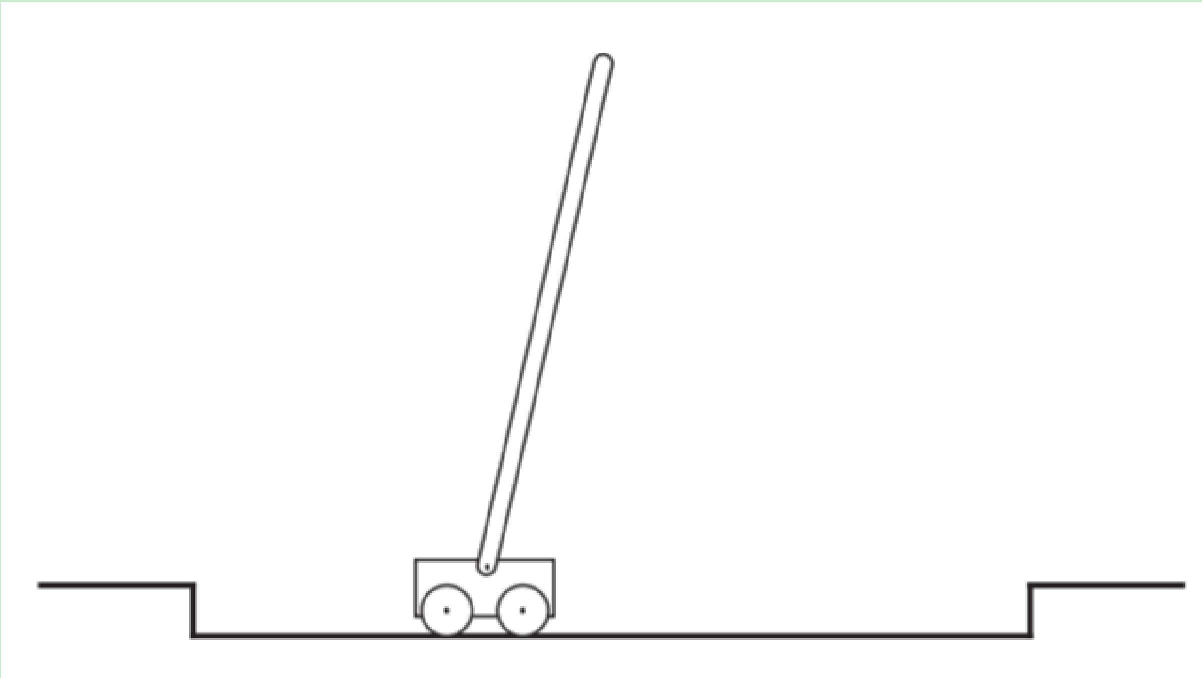
$$G_t = \sum_{k=0}^{\infty} R\gamma^k = R\frac{1}{1-\gamma}$$

Note that the returns at successive time steps are related to each other (which is important for RL):

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = R_{t+1} + \gamma G_{t+1}$$

Example 3.3 *pole-balancing*

- *terminal state: if the pole falls past a given angle from vertical or if the cart runs off the track*
- *starting state: pole reset to vertical*
- *reward (episodic): +1 for each time step before falling*
- *reward (continuing): -1 on each falling, 0 for all other times*



3.4 Unified Notation for Episodic and Continuing Tasks

We use $S_{t,i}$ to represents state at time t of episode i (same for $A_{t,i}$, $R_{t,i}$, etc). In practice, we dropped the i as we normally don't have to distinguish between different episodes. So, now S_t refers to $S_{t,i}$.

We use an $absorbing state$ to replace the terminate state. The absorbing state transits to itself and have reward = 0. This way, we can have the same formula of G_t for episodic and continuing tasks.

3.5 Policies and Value Functions

Value functions (either V or Q) are **always** conditional on some policy π. Sometimes in literature we leave off the π or $*$ and just refer to V and Q, because it is implicit in the context, but ultimately, every value function is always with respect to some policies.

Definition 3.10 *There are two value functions:*

- *value functions of states, or V, estimate how much expected return it is received for the agent to be in a given state;*
- *value functions of state-action pairs, or Q, estimate how much expected return it is received to perform a given action in a given state.*

Definition 3.11 *policy, $\pi(A_t = a|S_t = s)$, a mapping from states to probabilities of selecting each action. In other words, policy is a distribution over actions for each possible state.*

- deterministic policy: map each state to an action. $\pi(s)$.
- stochastic policy: map each state to a distribution over all possible actions. $\pi(a|s)$

Valid policy can only depend on current state, not other things like time or previous state (if, for example, last action affect rewards heavily, then the last action must include in the states).

Expectation of R_{t+1} in terms of π and the four-argument function p:

$$\mathbb{E}\left[R_{t+1}|S_t\right] = \sum_{a\in\mathcal{A}} \pi\left(a|S_t\right) \cdot r(S_t, a) = \sum_{a\in\mathcal{A}} \pi\left(a|S_t\right) \cdot \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p\left(s', r|S_t, a\right)$$

My word explanation: the expectation of reward given current state S_t= the total sum of the probability of perform action a at state s $\cdot$ the reward r $\cdot$ the probability of getting reward r (when performs action a at state s) (?)

Definition 3.12 *state-value function for policy π, (value function of a state s under a policy π), $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs:*

$$v_\pi(s) \doteq \mathop{\mathbb{E}}_\pi[G_t|S_t = s] = \mathop{\mathbb{E}}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in \mathcal{S}$$

Definition 3.13 *action-value function for policy π, (value of taking action a in state s under a policy π), $q_\pi(s,a)$, is the expected return starting from s, taking the action a, and **then** following policy π:*

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

The main difference between Q and V then, is the Q-value lets you play a hypothetical of potentially taking a different action in the first time step than what the policy might prescribe and then following the policy from the state the agent winds up in.

The value functions enable us to judge the quality of a policy.

Equation of v_π in terms of q_π and π:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$$

My word explanation: the value function is the total sum of probability of choosing action a at state s $\times$ the action-value of taking each action.
Equation of v_π in terms of π and p:

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

Equation of q_π in terms of v_π and the four argument p

$$q_\pi(s,a) = \sum_{s',r} p(s', r|s,a) \underbrace{[r + \gamma v_\pi(s')]}_{G_t?}$$

Definition 3.14 *Monte Carlo methods, estimation methods that involve averaging over many random samples of actual returns.*

For example, an agent follows policy π; and for each state s, the agent maintains an average of the actual returns which have followed that state, then the average converges to the state's value $v_\pi(s)$. (Recall the Law of Large Numbers, or sample-average method in Chapter 2). For another example, if separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values $q_\pi(s, a)$.

Definition 3.15 *Bellman equation for v_π:*

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

It expresses a relationship between the value of a state and the values of its successor states. First, we expand the expected return as a sum over possible action choices made by the agent. Second, we expand over possible rewards and next states condition on state s and action a. We break it down in this order because the action choice depends only on the current state, while the next state and reward depend only on the current state and action.

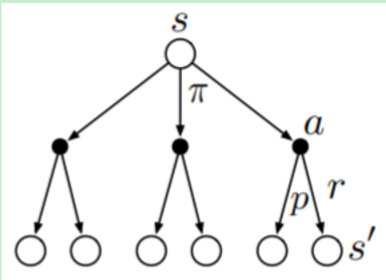


Figure 6: Backup diagram for v_π

Note that in backup diagram, the nodes do not necessarily represent distinct states (e.g. a state might be its own successor).

The value function v_π is the unique solution to its Bellman equation (??). My understanding is that since v_π is the only solution, meaning that v_π must converge to a point?

Definition 3.16 *Bellman equation for q_π:*

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s')q_\pi(s',a')]$$

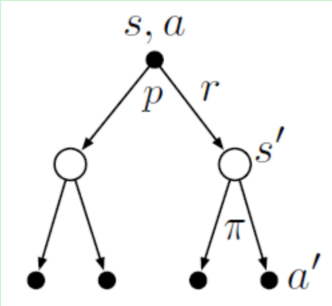
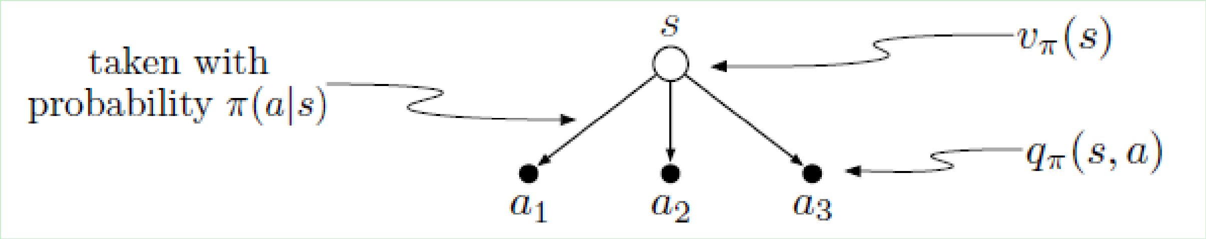
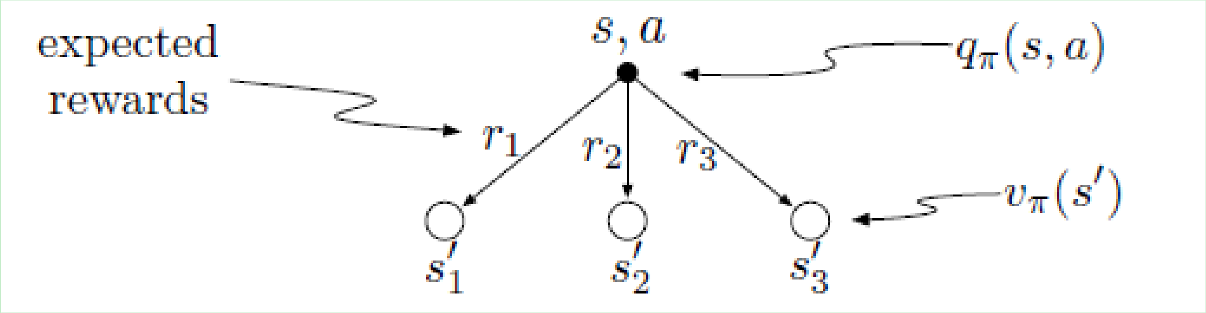


Figure 7: Backup diagram for q_π



(a)



(b)

Figure 8: (a) $v_\pi(s) = \sum_a \pi(a|s)q_\pi(s,a)$ (b) $q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$

The value of a state depends on the values of the actions possible in that state and on how likely each action is to be taken under the current policy.

We can use the Bellman Equation to solve for a value function by writing a system of linear equations. However, we can only solve small MDPs directly, but Bellman Equations will factor into the solutions we see later for large MDPs.

3.6 Optimal Policies and Optimal Value Functions

Solving a RL means finding a policy that achieves a lot of reward over the long run.

Definition 3.17 *optimal policy,* π_*

$$\pi_* \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S}$$

An optimal policy is defined as the policy with the highest possible value function in all states. At least one (deterministic) optimal policy always exists, but there may be more than one. The exponential number of possible policies makes searching for the optimal policy by brute-force intractable.

Definition 3.18 *optimal state-value function,* v_*

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

Definition 3.19 *optimal action-value function,* q_*

$$q_*(s) \doteq \max_\pi q_\pi(s, a), \quad \forall s \in \int, a \in \mathcal{A}$$

We can write q_* in terms of v_*:

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

Definition 3.20 *Bellman optimality equation for* v_**,*

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

Definition 3.21 *Bellman optimality equatoin for q_*,*

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

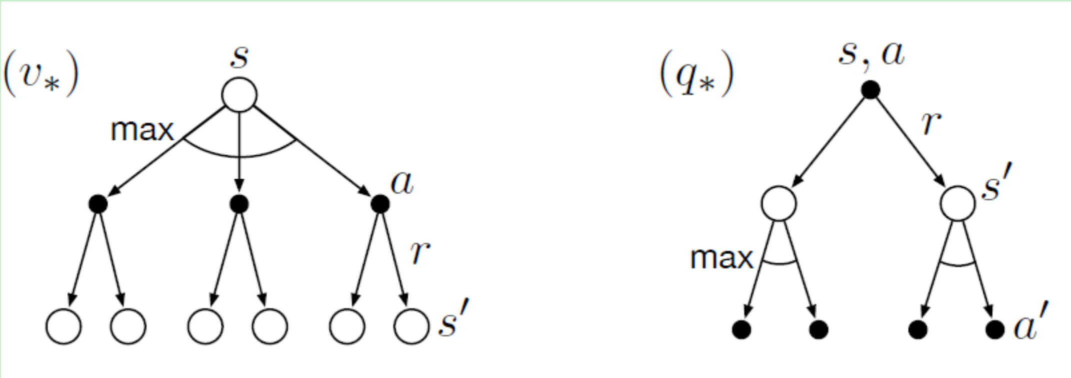


Figure 9: Backup diagram for v_* and q_*

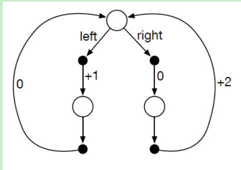
v_* is equal to the maximum of the term $\sum_{s',r} p(s', r|s,a)[r + \gamma v_*(s')]$ over all actions, while π_* is the particular action achieves the maximum.

Explicitly solving the Bellman optimality equation provides one route to finding π_*, and thus to solving the RL problem. However, this solution is rarely useful. On one hand, max_a is non-linear, so we cannot form linear equations as for Bellman equation. On the other hand, it needs an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least 3 assumptions:

- the dynamics of the environment are accurately known
- computational resources are sufficient
- the states have the Markov property

In RL, one typically has to settle for approximate solutions. Many RL methods can be taken as *approximately* solving the Bellman optimality equation.

Exercise Consider the contining MDP shown, the only decision to be made is in the top state with two actions (left, right). Find optimal policy when γ = 0.9.



Solution: Recall the definition of $v_\pi(s)$ and $q_\pi(s)$, we get

Left Policy

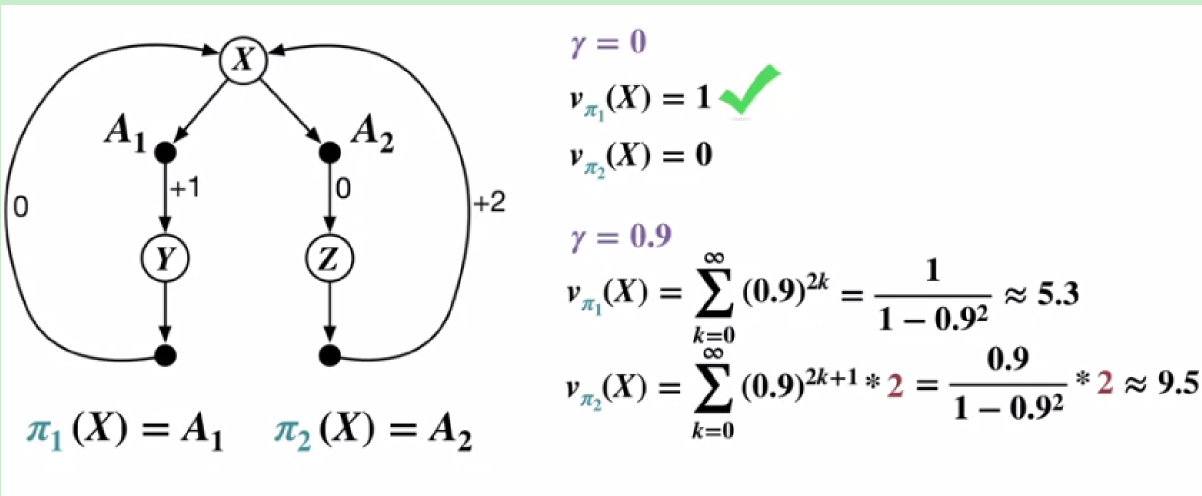
- $v(s_0) = 1 + 0.9 \cdot v(s_L)$
- $v(s_L) = 0 + 0.9 \cdot v(s_0)$
- $v(s_R) = 2 + 0.9 \cdot v(s_0)$

Right Policy

- $v(s_0) = 0 + 0.9 \cdot v(s_R)$
- $v(s_L) = 0 + 0.9 \cdot v(s_0)$
- $v(s_R) = 2 + 0.9 \cdot v(s_0)$

Solving above we find that all state values of Right Policy is greater than all state values of Left Policy. Thus, 'right' is the optimal policy for γ = 0.9.

It is important to note that if we take 'left' action in s_0, then policy π would never take us to state s_R (same for 'right' and s_L); however, due to the definition of π_*, we must evaluate the value function for all states, even ones that would not be visited under a policy we are evaluating.



To explain the geometry series, the left is 1, $\gamma \cdot 0$, $\gamma^2 \cdot 1$, $\gamma^3 \cdot 0$ sequence and the right is 0, $\gamma \cdot 2$, $\gamma^2 \cdot 0$, $\gamma^3 \cdot 2$ sequence. Taking discounted sum into consideration, we have the geometry series.

3.7 Optimality and Approximation

It is difficult to reach optimal solutions due to constraints from computational resource and memory. We can often approximate the optimal solutions.

3.8 Summary

- RL is about learning from interaction how to behave in order to achieve a goal.
- RL *agent* and its *environment* interact over a sequence of discrete time steps.
- *actions*: choices made by the agent, *states*: the basis for making the choices, *rewards*: basis for evaluating the choices.
- everything inside agent is known and controllable; environment is incompletely controllable, and are partially known
- a *policy* is a stochastic rule that the agent selects actions as a function of states
- agent's objective is to maximize reward it receives in the long run
- when RL setup with agent, environment, states, actions, rewards, policy and formulated with well-defined transition probabilities it constitutes a Markov decision process (MDP)
- a finite MDP is an MDP with finite state sets, action sets and reward sets.
- *return* is the function of future rewards that the agent seeks to maximize (in expected value)
- undiscounted formulation is appropriate for *episodic tasks*
- discounted formulation is appropriate for ***tabular*** *continuing tasks* (but not for approximate continuing tasks; see chap 10.3-4)
- *value functions* v_π (state) and q_π (state-action pair) are the expected return from that state/state-action under policy π

- *optimal value functions* v_* and q_* are the largest expected return by any policy
- A policy whose value functions are optimal is an optimal policy
- !! A MDP can have many optimal policies, but can have only one unique optimal value function (v_* and q_*)
- Any policy that is *greedy* w.r.t v_* and q_* must be π_*
- The Bellman optimality equations are special consistency conditions that the optimal value functions must satisfy
- In RL most cases, their optimal solutions cannot be found but must be approximated in some way.

3.9 Learning Objectives (UA RL MOOC)

1. Understand Markov Decision Processes (MDP)

MDP adds associative perspective to bandit problem, the actions affect not only immediate reward but also future rewards. MDP consists of states, actions and rewards. MDP state must have Markov property (only present matters).

2. Describe how the dynamics of an MDP are defined

The dynamics of an MDP is the transition probability $p(s', r|s, a)$.

3. Understand the graphical representation of a Markov Decision Process 4. Explain how many diverse processes can be written in terms of the MDP framework

From my understanding, almost all real world problems can be written in MDP framework. The issue is that reward definition sometimes may not be accurate, and states, actions may not be finite.

5. Describe how rewards relate to the goal of an agent

The reward is a way of imparting to the agent what you want to achieved, not how you want to achieved.

6. Understand episodes and identify episodic tasks

Episodes have finite trajectory length and must end in a terminal state. Episodes are independent of previous episode. Episodic task is task with episodes that ends in terminal state.

7. Formulate returns for continuing tasks using discounting

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

8. Describe how returns at successive time steps are related to each other

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

9. Understand when to formalize a task as episodic or continuing

When the tasks can be naturally broken into episodes (like chess game), the task is episodic; otherwise it is continuing task (like walking robot).

10. Recognize that a policy is a distribution over actions for each possible state

Policy is either a deterministic one (mapping from state to action) or a stochastic one (with probability distribution from state to actions). The deterministic policy can be considered as a 100% probability distribution policy.

11. Describe the similarities and differences between stochastic and deterministic policies

see 10

12. Generate examples of valid policies for a given MDP

13. Describe the roles of state-value and action-value functions in reinforcement learning

state-value gives the total expected return of a state following a policy, and action-value gives the total expected return of a state with an action then following a policy. The scalar values (those expected returns) helps us to choose optimal policies, which solves an RL problem.

14. Describe the relationship between value functions and policies

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

15. Create examples of valid value functions for a given MDP

16. Derive the Bellman equation for state-value functions

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

17. Derive the Bellman equation for action-value functions

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

18. Understand how Bellman equations relate current and future values

We see that current values (in LHS) and future values (in RHS) of the Bellman equations.

19. Use the Bellman equations to compute value functions

20. Define an optimal policy

$$
\pi_* \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S}
$$

An optimal policy is defined as the policy with the highest possible value function in all states.

21. Understand how a policy can be at least as good as every other policy in every state

Let us say there is a policy π_1 which does well in some states, while policy π_2 does well in others. We could combine these policies into a third policy π_3, which always chooses actions according to whichever of policy π_1 and π_2 has the highest value in the current state. π_3 will necessarily have a value greater than or equal to both π_1 and π_2 in every state.

22. Identify an optimal policy for given MDPs

23. Derive the Bellman optimality equation for state-value functions

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

24. Derive the Bellman optimality equation for action-value functions

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

25. Understand how the Bellman optimality equations relate to the previously introduced Bellman equations

26. Understand the connection between the optimal value function and optimal policies

27. Verify the optimal value function for given MDPs

28. Does adding a constant to all rewards change the set of optimal policies in episodic tasks? (Yes, adding a constant to all rewards changes the set of optimal policies.) Does adding a constant to all rewards change the set of optimal policies in continuing tasks? (No, as long as the relative differences between rewards remain the same, the set of optimal policies is the same.)

Explanation: Let write R_c for total reward with added constant c of a policy as

$$R_c = \sum_{i=0}^{K}(r_i + c)\gamma^i = \sum_{i=0}^{K} r_i\gamma^i + \sum_{i=0}^{K} c\gamma^i$$

So if we have two policies with the same total reward (w/o added c):

$$\sum_{i=0}^{K_1} r_i^1\gamma^i = \sum_{i=0}^{K_2} r_i^2\gamma^i$$

but with different lengths $K_1 \neq K_2$, the total reward with added constant will be different, because the second term in R_c ($\sum_i^K c\gamma^i$) will be different.

For example, consider two optimal policies, both generating the same cumulative reward of 10, but the first policy visits 4 states before it reaches a terminal state, while the second visits only two states. The reward can be written as

$$10 + 0 + 0 + 0 = 10$$

and

$$0 + 10 = 10$$

But when we add 100 to every reward:

$$110 + 100 + 100 + 100 = 410$$

and

$$100 + 110 = 210$$

Thus, now the first policy is better.

In the continuous case, the episodes always have length $K = \inf$. Therefore, they always have the same lenght, and adding a constant does not change anything, because the second term in R_c stays the same.

29. γ specifies the problem set. Once we have set the γ, we need to find the solution, so we do not want the agent to change the γ.

30. Adam: whenever I deal with continuous time system (like robotics), the first thing is: how do we discretize the time?

4 Dynamic Programming

In RL, DP refers to a collection of algorithms that can be used to compute optimal policies π_* given a perfect model of the environment as a Markov decision process (MDP). DP itself cannot be useful in RL due to its computational cost and requirement of a perfect model, but DP provides an essential foundation for the understanding of other algorithms. Other algorithms can be treated as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Starting with this chapter, we assume the environment is a finite MDP (i.e. S, A, R are finite and dynamics are given by a set of probabilities $p(s', r|s, a)$). A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As shown in Chapter 3, we can obtain π_* once we have found v_* or q_*, which satisfy the Bellman optimality equations:

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

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

4.1 Policy Evaluation (Prediction)

Definition 4.1 *policy evaluation (a.k.a prediction), compute the state-value function v_π for an arbitrary policy π*

Recall that $\forall s \in \mathcal{S}$,

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

If the environment's dynamics are completely known, then the last equation is a system of $|\mathcal{S}|$ simultaneous linear equations. For this case, iterative solution methods are most suitable. Consider a sequence of approximate value functions v_0, v_1, v_2, $\cdots$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$. The initial approximation, v_0, is chosen arbitrarily, and each successive approximation is obtained by using the Bellman equation v_π as an update rule ($\forall s$):

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

We see that v_k = v_π is a fixed point for this update rule as $k \to \infty$, v_k converges to v_π. This algorithm is called *iterative policy evaluation*. The update is called *expected update*. It is called *expected* because the updates are based on an expectation over all possible next states rather than on a sample next state.

In this algorithm, we use one array and update values in place.

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

Input π (the policy to be evaluated); $\theta > 0$ threshold determining accuracy of estimation. Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and V to 0.

Loop:

$\Delta \leftarrow 0$

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

$v \leftarrow V(s)$

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

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Policy evaluation of q_π:

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

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

4.1.1 Example Code

```
# V = np.zeros(len(env.S))
# pi = np.ones((len(env.S), len(env.A))) / len(env.A)
def evaluate_policy(env, V, pi, gamma, theta):
    delta = float('inf')
    while delta > theta:
        delta = 0
```

```
        for s in env.S:
            v = V[s]
            bellman_update(env, V, pi, s, gamma)
            delta = max(delta, abs(v - V[s]))
    return V

def bellman_update(env, V, pi, s, gamma):
    for state, pi_state in enumerate(pi):
        if s != state:  # not the state we want, skip
            continue
        tmp = 0
        for action, action_prob in enumerate(pi_state):
            transitions = env.transitions(state, action)
            for next_s, (reward, trans_prob) in enumerate(transitions):
                tmp += action_prob * trans_prob * (reward + gamma * V[
                                                    next_s])
        V[s] = tmp
```

4.2 Policy Improvement

Definition 4.2 *policy improvement theorem. Let π and π' be any pair of deterministic policies, where π' is identical to π except $\pi'(s) = a \neq \pi(s)$, s.t. $\forall s \in \mathcal{S}$,*

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

Then the policy π' must be better (or as good as) than π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$

$$v_{\pi'}(s) \geq v_\pi(s)$$

Greedy policy $\pi'(s)$:

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

Definition 4.3 *policy improvement, the process of making a new policy π' that improves on an original policy π, by making it greedy w.r.t the value function of π.*

If there are ties in policy improvement steps, each maximizing action can be given a portion of the probability of being selected in the new greedy policy.

4.3 Policy Iteration

It is trivial to see that we can ultimately reach an optimal policy through iterative improving policies and value functions [3].

Definition 4.4 *policy iteration, the way of finding an optimal policy.*

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi' \xrightarrow{E} v_{\pi'} \xrightarrow{I} \pi'' \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

[3]In large-scale reinforcement learning problems, it is typically impractical to run either of these steps to convergence, and instead the value function and policy are optimized jointly. By Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor

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

1. Initialization

   $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily $\forall s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation

   Loop:

   $\Delta \leftarrow 0$

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

   $v \leftarrow V(s)$

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

   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

   until $\Delta < \theta$

3. Policy Improvement

   $policyStable \leftarrow true$

   For each $s \in \mathcal{S}$

   $oldAction \leftarrow \pi(s)$

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

   If $oldAction \notin \{a_i\}$, which is the all equal best solutions from $\pi(s)$

   then $policyStable \leftarrow false$

   If $policyStable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

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

1. Initialization
   $Q(s, a) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily $\forall s \in \mathcal{S}, a \in \mathcal{A}$
2. Policy Evaluation

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

3. Policy Improvement

   $policyStable \leftarrow true$
   For each $s \in \mathcal{S}, a \in \mathcal{A}$
   $oldAction \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a Q(s, a)$
   If $oldAction \notin \{a_i\}$, which is the all equal best solutions from $\pi(s)$
   then $policyStable \leftarrow false$
   If $policyStable$, then stop and return $Q \approx q_*$ and $\pi \approx \pi_*$; else go to 2

Definition 4.5 *ε-soft, the probability of selecting each action in each state, is at least* $\frac{\varepsilon}{A(s)}$

4.3.1 Example Code

```
def improve_policy(env, V, pi, gamma):
    policy_stable = True
    for s in env.S:
        old = pi[s].copy()
```

```python
        q_greedify_policy(env, V, pi, s, gamma)
        if not np.array_equal(pi[s], old):
            policy_stable = False
    return pi, policy_stable

def policy_iteration(env, gamma, theta):
    V = np.zeros(len(env.S))
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    policy_stable = False

    while not policy_stable:
        V = evaluate_policy(env, V, pi, gamma, theta)
        pi, policy_stable = improve_policy(env, V, pi, gamma)
    return V, pi

def q_greedify_policy(env, V, pi, s, gamma):
    def argmax(_values):
        """
        Input:  (list) _values
        Return: (int)  the index of the item with the highest value.
        Breaks ties randomly.
        """
        top_value = float("-inf")
        ties = []
        for i in range(len(_values)):
            if _values[i] > top_value:
                ties = []
                ties.append(i)
                top_value = _values[i]
            elif _values[i] == top_value:
                ties.append(i)
            else:
                pass
        return np.random.choice(ties)
    for state, pi_state in enumerate(pi):
        if s != state:
            continue
        # take the action that result in max action-value
        q_values = []
        for action, action_prob in enumerate(pi_state):
            transitions = env.transitions(state, action)
            tmp = 0
            for next_s, (reward, trans_prob) in enumerate(transitions):
                tmp += trans_prob * (reward + gamma * V[next_s])
            q_values.append(tmp)
```

```
        pi[state] = int(0)
        pi[state][argmax(q_values)] = int(1)
```

4.4 Value Iteration

The policy evaluation of policy iteration requires multiple sweeps through the state set and convergence occurs only in the limit. We want to truncate policy evaluation without losing the convergence guarantees of policy iteration. One of the technique is value iteration.

Definition 4.6 *value iteration, stop policy evaluation after one sweep (one update after each state).*

$$\begin{aligned}
v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')], \quad \forall s \in \mathcal{S} \\
q_{k+1}(s,a) &\doteq \mathbb{E}[R_{t+1} + \max_{a'} \gamma q_k(s',a')] \\
&= \sum_{s',r} p(s',r|s,a)[r + \max_{a'} \gamma q_k(s',a')]
\end{aligned}$$

For arbitrary v_0, the sequence v_k converges to v_*. Note this equation is similar to the Bellman optimality equation of v_* and the update rule of v_{k+1}.

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

1. Initialization

   $\theta > 0, V(s) \forall s \in \mathcal{S}^+, V(term) = 0$

2. Policy Evaluation

   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 $\Delta < \theta$

3. Policy Improvement

Output a deterministic policy, $\pi \approx \pi_*$, s.t.

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

One value iteration sweep combines one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates.

4.4.1 Example Code

```
def value_iteration(env, gamma, theta):
    V = np.zeros(len(env.S))
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
```

```
            bellman_optimality_update(env, V, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    for s in env.S:
        q_greedify_policy(env, V, pi, s, gamma)
    return V, pi

def value_iteration2(env, gamma, theta):
    V = np.zeros(len(env.S))
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
            q_greedify_policy(env, V, pi, s, gamma)
            bellman_update(env, V, pi, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V, pi

def bellman_optimality_update(env, V, s, gamma):
    for state, pi_state in enumerate(pi):
        if s != state:  # not the state we want, skip
            continue

        state_values = [] # we later take the largest state value
        for action, action_prob in enumerate(pi_state):
            transitions = env.transitions(state, action)
            tmp = 0
            for next_s, (reward, trans_prob) in enumerate(transitions):
                tmp += trans_prob * (reward + gamma * V[next_s])
            state_values.append(tmp)
        V[state] = max(state_values)
```

4.5 Asynchronous Dynamic Programming

Systematic DP is intractable to large state set problems; it would take a lot of time for a single sweep.

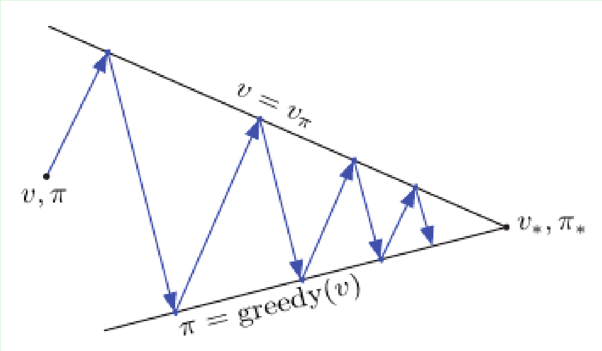
Asynchronous DP are in-place iterative DP that updates state value in random order. Async DP still needs to update all state values for convergence. We can also intermix policy iteration and value iteration, as for systematic DP, to produce an async truncated policy iteration.

The idea of async DP, and likewise in other RL algorithms, is that we *focus* the DP updates onto those states that are most relevant to the agent. In this way, we can speed up the process, but not less computation.

Sweepless DP algorithm is beyond the scope of this book.

4.6 Generalized Policy Iteration

Definition 4.7 ***Generalized Policy Iteration (GPI)****, the idea of letting policy evaluation and policy improvement processes interact, independent of the granularity of the two processes.*



For GPI examples, the value iteration only uses one sweep of policy evaluation, and async DP does sweep the in order. Almost all RL methods are GPI.

Monte Carlo sampling method, an alternative to learn a value function, estimates each state value independently by averaging large number of returns. Brute-force search is an alternative for finding π_* by computing with every π. Bootstrapping, the process of using the value estimates of successor states to improve current value estimate. This is more efficient (estimating value collectively) than Monte Carlo method (estimating each value independently).

4.7 Efficiency of Dynamic Programming

DP, in the worst case, the time takes to find an optimal solution is polynomial in the number of states and actions (even though the total number of deterministic policies is $|\mathcal{A}|^{|\mathcal{S}|}$. DP is better at large state spaces than competing methods (e.g. direct search, linear programming).

4.8 Summary

- DP requires complete and accurate model of the environment
- Policy evaluation: iterative computation of the value function for a given policy
- Policy improvement: the computation of an improved policy given the value function for that policy
- Two most popular DP methods: policy iteration and value iteration, by combining policy evaluation and policy improvement. These two can solve finite MDPs given complete knowledge of the MDP
- Classical DP: sweep whole state set, performing expected update on each state. The update is based on all possible successor states and their probabilities of occurring.
- Generalized Policy Iteration (GPI): interacting processes (policy evaluation and policy improvement) revolving around an approximate policy and an approximate value function
- Async DP: unlike classical/systematic DP, it update states in random order.
- **bootstrapping**: update estimates on the basis of other estimates (English definition of bootstrap: you pull yourself up by your boot-straps, meaning you achieved success by your own efforts.)

4.9 Learning Objectives (UA RL MOOC)

Only policies greedy with respect to the optimal value function are guaranteed to be optimal.

1. Understand the distinction between policy evaluation and control

policy evaluation computes the state value for π, while control is the dance of policy and value.

2. Explain the setting in which dynamic programming can be applied, as well as its limitations

DP can be applied in relatively large MDPs. Its limitations include (1) requiring complete model of the MDP.

3. Outline the iterative policy evaluation algorithm for estimating state values under a given policy

see chap 4.1

4. Apply iterative policy evaluation to compute value functions

5. Understand the policy improvement theorem

policy improvement theorem: the computation of an improved policy given the value function for that policy.

6. Use a value function for a policy to produce a better policy for a given MDP
7. Outline the policy iteration algorithm for finding the optimal policy

see chap 4.3

8. Understand "the dance of policy and value"

The PE and PI happens (mostly) alternatively, and reach optimal state value function and optimal policy eventually.

9. Apply policy iteration to compute optimal policies and optimal value functions

10. Understand the framework of generalized policy iteration

PE + PI. There can be multiple updates or no update for a state in each iteration. All state must be visited once.

11. Outline value iteration, an important example of generalized policy iteration

see chap 4.4

12. Understand the distinction between synchronous and asynchronous dynamic programming methods

sync DP is the systematic DP, which updates all states in order. async DP updates state in any order, which makes it for faster convergence.

13. Describe brute force search as an alternative method for searching for an optimal policy

Brute-force search takes $|A|^{|S|}$ time. It compute for every combination of state-action mappings (policies).

14. Describe Monte Carlo as an alternative method for learning a value function

MC estimates state value by averaging large number of returns.

15. Understand the advantage of Dynamic programming and “bootstrapping” over these alternative strategies for finding the optimal policy

DP is fast and boostrapping uses more states’ info during estimation than the other alternatives, which is more efficient.

16. Why are DP algorithms considered planning methods?

Because they use a model to improve the policy (which is the definition of planning method).

5 Monte Carlo Methods

MC, is for estimating value functions and discovering optimal policies. Unlike methods in previous chapters, MC does not use complete knowledge of the environment. MC only requires *experience* - sample sequences of states, actions and rewards from actual or simulated interaction with an environment. That is, MC is a **model-free** learning method.

MC is based on averaging complete sample returns. Since we are considering sampling returns, we define MC only for episodic tasks (in this book) [4]. Once an episode ends, the value estimates and policies change. MC is thus a episode-by-episode update, not a step-by-step (online) update.

The procedure of MC is similar to DP: prediction problem (finding v_π and q_π) -> policy improvement -> control problem (solve by GPI). Except that, in DP, we compute value functions based on model, here we learn value functions from sample returns.

5.1 Monte Carlo Prediction

To learn v_π, we average the returns observed after visits to that state. The more returns are observed, the more closer to the expected value (recall the Law of Large Numbers). Each occurrence of s in an episode is called a *visit* to s. First-visit to s is the first time s is visited in an episode.

Note that in the following algorithms, we compute the returns backwards from timestep $T-1$ to 0.

[4]We can never sample an actual return value in continuous tasks. We may get around by using artificial time horizon, or TD (with eligibility traces); however, all these incur bias. see https://datascience.stackexchange.com/a/77789/137431

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

Input: policy π (to be evaluated) Initialize:

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

Loop forever (for each episode):

  Generate an episode following $\pi : S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$

  $G \leftarrow 0$

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

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

    if S_t not appears in $S_0, S_1, ..., S_{t-1}$:

      Append G to $Returns(S_t)$

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

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

Input: policy π (to be evaluated) Initialize:

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

Loop forever (for each episode):

  Generate an episode following $\pi : S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$

  $G \leftarrow 0$

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

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

    Append G to $Returns(S_t)$

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

The first-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visit to s.

The every-visit MC method averages the returns following all visits to s. This method extends more naturally to function approximation (chap 9) and eligibility traces (chap 12).

Advantage of MC over DP: (1) no need the model (i.e. the transition dynamics); (2) can work with sample episodes *alone*; (3) can estimate the target state only while ignoring all other states (MC does not bootstrap, meaning each state estimation is independent of other states).

5.2 Monte Carlo Estimation of Action Values

When without a model, $v(s)$ along cannot determine π; in this case, $q(s,a)$ is more useful. Thus, one of MC goals is to estimate q_*.

The only problem is that many (s,a) pairs may never be visited. For example, if we have a deterministic policy we only get one action per state (the one that the policy favor). Hence, we only observe returns for one action. This is a problem of **maintaining exploration**. One way to solve this is exploring starts. *exploring starts*: assume episodes start in a (s,a) pair, and that every pair has a nonzero probability to be selected at the start. (This is sometimes useful). A more common way to go about it, is to only consider stochastic policies where the probability of every action in every state is not 0.

MC Exploring Starts, for estimating $V \approx v_\pi$

Initialize:
  $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), $\forall s in \mathcal{S}$
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
Loop forever (for each episode):
  Choose a random (S_0, A_0) pair s.t. all pairs have nonzero probability
  Generate an episode(SAR trajectory) from (S_0, A_0), following π
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, ..., 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    if (S_t, A_t) pair not in the episode:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

We use exploring start for evaluating deterministic policy, and ϵ-greedy for evaluating stochastic policy.

5.3 Monte Carlo Control

We now look at how our MC estimation can be used in control. Meaning, to approximate optimal policies.

The idea is to follow generalized policy iteration (GPI), where we will maintain an approximate policy and an approximate value function. We continuously alter the value function to be a better approximation for the policy, and the policy is continuously improved (see previous chapter).

The policy evaluation part is done exactly as described in the previous chapter (MC Prediction), except that we are evaluating the state-action pair, rather than states.

The policy improvement part is done by taking greedy actions in each state. That is, for any action-value function q, and for every state s, the greedy policy chooses the action with maximal action-value:

$$\pi(s) \doteq \arg\max_a q(s,a)$$

5.4 Monte Carlo Control without Exploring Starts

To make sure that all actions are being selected infinitely often, we must continuously select them. There are 2 approaches to ensure this — on-policy methods and off-policy methods.

Definition 5.1 ***On-policy methods**: evaluate or improve the policy that is used to make decisions.*

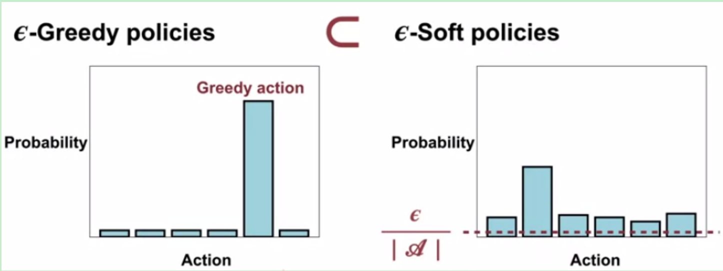
Definition 5.2 ***Off-policy method**: evaluate or improve a policy different from the one used to generate the data (make decisions). The policy being evaluated/improved is target policy, and the one generate behavior is behavior policy.*

On-policy method is a special case of off-policy method, where policy $\pi_t = \pi_b$.

Definition 5.3 ***soft policy***: $\pi(a|s) > 0 \quad \forall s \in \mathcal{S}$ *and* $\forall a \in \mathcal{A}(s)$.

Definition 5.4 ***ϵ-soft policy***: $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|} \quad \forall s \in \mathcal{S}$ *and* $\forall a \in \mathcal{A}(s)$ *and for some* $\epsilon > 0$.

For example, ϵ-greedy policies are examples of ϵ-soft policies. ϵ-soft policy gradually moves closer to a deterministic optimal policy.



On-policy first-visit MC control, estimates $V \approx v_\pi$

parameter: small ϵ > 0

Initialize:

$\pi(s) \leftarrow$ an arbitrary ϵ-soft policy

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

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

Loop forever (for each episode):

Generate an episode(SAR trajectory) following π

$G \leftarrow 0$

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

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

if (S_t, A_t) pair not in the episode:

Append G to $Returns(S_t, A_t)$

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

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

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

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A* \text{ (greedy)} \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A* \text{ (non-greedy)} \end{cases} \tag{2}$$

Soft policies cannot find optimal policy; instead, it can only find optimal ϵ-soft policy because we always give at least $\epsilon/|A|$ probability for each action, which cannot converge to an optimal deterministic policy. On the contrary, exploring start can find optimal policy. However, soft policy can perform generally well so we can abandon exploring start.

The optimal ϵ-soft policy is an ϵ-greedy policy.

5.5 Off-policy Prediction via Importance Sampling

Recall the exploration-exploitation dilemma. On-policy learning is actually a compromise: it learns action values (q) not for the optimal policy, but for a *near*-optimal policy that still explores. Since they cannot learn a optimal policy while behaving a exploratory policy.

The solution is off-policy: we use two policies, one for leaning optimal-policy, and one for exploration. We use episodes generated from the behavior policy to go and explore the environment, and then use this to update our target policy. We do this to estimate v_π and q_π.

There are other useful applications of off-policy learning, such as learning from demonstration and parallel learning. But facilitating exploration is one of the main motivators.

In the prediction step of GPI, the target and behavior policies are fixed. All we have are episodes following behavior policy π_b. **Assumption of *coverage***: $\pi(a|s) > 0$ implies $b(a|s) > 0$; policy b must be stochastic in states where $b \neq \pi$.

For off-policy learning to work, we need to align policy π and policy b through importance sampling.

Definition 5.5 ***Importance sampling**: a method of estimating expected values of one (target) distribution, given samples from another (behavior) distribution.*

We apply importance sampling to off-policy learning by using importance sampling ratio.

Definition 5.6 ***Importance sampling ratio***: *weight the returns based on the relative probability of their trajectories occurring under π and b.*

The probability of the state-action trajectory, from $S_t,\ A_t, S_{t+1}, A_{t+1}, ..., S_T$ occurring under policy π is:

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

Thus, the importance sampling ratio is:

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

Note after canceling the transition probabilities, the importance sampling ratio depends only on the two policies and the sequence, not on the MDP.

$$
\begin{aligned}
\mathop{\mathbb{E}}_{\pi}[X] \doteq \sum_{x\in X} x\pi(x) = \sum_{x\in X} x\pi(x)\frac{b(x)}{b(x)} &= \sum_{x\in X} x\,\underbrace{\frac{\pi(x)}{b(x)}}_{\rho(x)} b(x) \\
&= \sum_{x\in X} \underbrace{x\rho(x)}_{\text{new random var} X} \quad b(x) \\
&= \sum_{x\in X} X b(x) \\
&= \mathop{\mathbb{E}}_{b}[X\rho(X)]
\end{aligned}
$$

$$
\mathop{\mathbb{E}}_{b}[X\rho(x)] = \sum_{x\in X} x\rho(x)b(x) \approx \frac{1}{n}\sum_{i=1}^{n} x_i\rho_i(x), x_i \sim b \quad \text{(by def of } \mathbb{E})
$$

All we have from the behavior policy are returns G_t, we can use these returns to estimates $v_b(s) = \mathbb{E}[G_t|S_t = s]$, but not v_π. To adjust for the difference between v_b and v_π, we apply importance sampling ratio ρ:

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

5.6 Summary

- MC advantages over DP: (1) no need of model to learn; (2) MC can be used with simulation or sample models; (3) MC can focus on a small subsets of the states (can just keep visiting target states); (4) less influenced by violations of the Markov property because MC do not bootstrap.
- MC provide an alternative Policy Evaluation process. They average sample returns instead of using a model to compute $v(s)$.
- action-value functions can be used to improve the policy without a model.
- Off-policy prediction: learning the value function of π_{target} from data generated by a different π_b.
- ordinary importance sampling: unbiased estimates but large(inf) variance.
- weighted importance sampling: biased estimate with finite variance

5.7 Learning Objectives (UA RL MOOC)

Lesson 1: Introduction to Monte-Carlo Methods

1. Understand how Monte-Carlo methods can be used to estimate value functions from sampled interaction

Because value of a state is the expected return (expected cumulative future discounted reward), and MC estimate the $V(s)$ by averaging the returns from visits to those states. Due to the Law of Large Numbers, the estimated value converges to the true value once there are enough sample returns.

2. Identify problems that can be solved using Monte-Carlo methods

MC can be used to solve problems that is hard to define a MDP model. For example, we can use MC to solve Blackjack and Soap Bubble problem from the book.

3. Use Monte-Carlo prediction to estimate the value function for a given policy.

This is a MC prediction problem. We can solve the problem by MC first-visit method, or MC every-visit method.

Lesson 2: Monte-Carlo for Control

4. Estimate action-value functions using Monte-Carlo

When there is no model available, we can estimate action values. We can use MC Exploring Start for the problem. Or we can use the same methods for estimating $V(s)$, with MC first-visit/every-visit methods.

5. Understand the importance of maintaining exploration in Monte-Carlo algorithms

If we don't explore enough, then some (s, a) pairs are never visited. Without enough return, we cannot effectively estimate $q(s, a)$ by averaging.

6. Understand how to use Monte-Carlo methods to implement a GPI algorithm

Policy evaluation is done by averaging enough returns; policy improvement is that $\pi(s) \doteq argmax_a q(s, a)$.

7. Apply Monte-Carlo with exploring starts to solve an MDP

see Chap 5.2

Lesson 3: Exploration Methods for Monte-Carlo

8. Understand why exploring starts can be problematic in real problems

For example, we cannot let a self-driving car to start in all (s, a) situations. Some of them can be dangerous!

9. Describe an alternative exploration method for Monte-Carlo control

We can use on-policy ($\epsilon - soft$) methods, or off-policy (one behavior policy + one target policy) methods.

Lesson 4: Off-policy learning for prediction

10. Understand how off-policy learning can help deal with the exploration problem

Off-policy learning has one policy that is specific to explore the world (and generate behavior), and the other policy is to used the experience to learn.

11. Produce examples of target policies and examples of behavior policies

target policy can be a robotic arm, behavior policy is the expert.

12. Understand importance sampling

Importance sampling is a technique for estimating expected values under one distribution given samples from another.

13. Use importance sampling to estimate the expected value of a target distribution using samples from a different distribution

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

14. Understand how to use importance sampling to correct returns

With enough returns, we can lower the variance and eventually return the true value.

15. Understand how to modify the Monte-Carlo prediction algorithm for off-policy learning.

We use importance sampling for off-policy MC.

6 Temporal-Difference Learning

TD learning is a learning method that's specialized for prediction learning (the scalable model-free learning). TD learning is learning a prediction from another, later, learned prediction (i.e., learning a guess from a guess). The TD error is the difference between two predictions, the temporal difference; otherwise TD learning is the same as supervised learning, backpropagating the error. You can think one step TD (make a prediction, wait one step, see the result) as traditional supervised learning (supervisor tells you the result after one step).

Note prediction problem means policy evaluation, control problem means finding π_*.

TD combines some of the features of both MC and DP. TD does not require a model and can learn from interactions (like MC), and TD can bootstrap, thus learn online (without waiting till the end of episodes) (like DP).

TD(λ) unifies DP, MC, TD.

Prediction problem = policy evaluation (i.e., estimating v_π for a given π)

Control problem = finding an optimal policy

6.1 TD Prediction

Definition 6.1 ***constant-α MC:***

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[G_t - V(S_t)]}_{\textit{MC error}}$$

Unlike MC which has to wait until the end of an episode to update, TD only has to wait one time step.

Definition 6.2 ***TD(0)**, or one-step TD:*

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]}_{\textit{TD error}}$$

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Init $V(s)$ $\forall s \in S+$, $V(term.) = 0$

Loop for each episode:

  Init 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

- MC target is an estimate, because a sample return is needed for real expected return
- DP target is an estimate, because the next state value is unknown and the current estimate is used
- TD target is an estimate because (1) samples the expected values; (2) and uses current estimate V instead of the true v_π

TD methods combine the sampling of MC with the bootstrapping of DP. TD and MC updates are *sample update* because they involve looking ahead to a sample successor state.

- sample update: based on a single sample sample successor
- expected update: a complete distribution of all possible successors

Definition 6.3 ***TD error***:

$$\delta_t \doteq \underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\textit{TD Target}} - V(S_t)$$

MC error can be written as a sum of TD errors (think of it this way, TD updates immediately at each time step, MC updates after an episode (which include a lot of timesteps)

$$
\begin{aligned}
\text{MC error} = G_t - V(S_t) &= \underbrace{R_{t+1} + \gamma G_{t+1}}_{G_t} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\
&= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2}) \\
&= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k
\end{aligned}
$$

6.2 Advantages of TD Prediction Methods

TD methods have an advantage over DP methods in that they do not require a model of the environment.

TD methods over MC methods is that they are naturally implemented in an online, fully incremental fashion.

TD converges faster than MC.

6.3 Optimality of TD(0)

Definition 6.4 ***batch updating***: *updates are made only after processing each complete batch of training data.*

- max-likelihood estimate: find the parameter value whose probability of generating the data is greatest
- certainty-equivalence estimate: equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated

In batch form, TD(0) is faster than MC methods because it computes the true certainty-equivalence estimate.

6.4 Sarsa: On-policy TD control

This week, you will learn about using temporal difference learning for control, as a generalized policy iteration strategy. You will see three different algorithms based on bootstrapping and Bellman equations for control: Sarsa, Q-learning and Expected Sarsa. You will see some of the differences between the methods for on-policy and off-policy control, and that Expected Sarsa is a unified algorithm for both. You will implement Expected Sarsa and Q-learning, on Cliff World.

We consider transitions from state-action pair to state-action pair.

Definition 6.5 ***SARSA***,

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

If S_{t+1} *is terminal, then* $Q(S_{t+1}, A_{t+1}) = 0$.

Since the update rule uses every element of the quintuple of events,

$$(S_t, A_t, R, S_{t+1}, A_{t+1})$$

, that make up a transition from one state-action pair to the next. Hence, the algorithm name *SARSA*.

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Init $Q(s, a)$, $\forall s \in \S^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

  Init S

  Choose A from S using policy derived from Q (eg ϵ-greedy)

  Loop for each step of episode:

    Take action A, observe R, S'

    Choose A' from S' using policy derived from Q (eg 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

The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state S' and the current policy's action A'. It estimates the return for state-action pairs assuming the current policy continues to be followed.

6.5 Q-learning: Off-policy TD control

Definition 6.6 ***Q-learning***,

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

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Init $Q(s, a)$, $\forall s \in \S^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

  Init S

  Loop for each step of episode:

    Choose A from S using policy derived from Q (eg 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

The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state S' and the greedy action a. In other words, it estimates the *return* (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy.

	SARSA	Q-learning
Choosing A'	π	π
Updating Q	π	b

where π is, for example, a ϵ-greedy policy ($\epsilon > 0$ with exploration), and b is a greedy policy (e.g. $\epsilon = 0$, no exploration)

1. Given that Q-learning is using different policies for choosing next action A' and updating Q. In other words, it is trying to evaluate π while following another policy b, so it's an off-policy algorithm
2. In contrast, SARSA follows π all the time, hence it is an on-policy algorithm.

6.6 Expected Sarsa

Expected SARSA like Q-learning except that instead of the maximum over next state-action pairs, it uses the expected value (taking into account how likely each action is under the current policy).

Definition 6.7 ***Expected SARSA,***

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

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

Expected SARSA moves deterministically in the same direction as SARSA moves in expectation. Expected SARSA is more complex computationally than SARSA, but in return, it eliminates the variance due to the random selection of A_{t+1}.

6.6.1 TD control and Bellman equations

TD control algorithms are based on Bellman equations.

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

SARSA, on-policy, uses the sample based version of the Bellman equation $R + \gamma Q(S_{t+1}, A_{t+1})$, learns q_π.

Expected SARSA, on/off-policy, uses the same Bellman equation as SARSA, but samples it differently. It takes an expectation over the next action values. $R + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a')$

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

Q-learning, off-policy, uses the Bellman optimality equation $R+\gamma max_{a'}Q(S_{t+1}, a')$, it learns q_*.

SARSA can do better than Q-learning when performance is online, because on-policy control methods account for their own exploration.

6.7 Learning Objectives (UA RL MOOC)

Lesson 1: Introduction to Temporal Difference Learning

1. Define temporal-difference learning

A way to incrementally estimate the return through bootstrapping

2. Define the temporal-difference error

$$\delta_t \doteq \underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD Target}} - V(S_t)$$

3. Understand the TD(0) algorithm

see chap 6.1

Lesson 2: Advantages of TD

4. Understand the benefits of learning online with TD

5. Identify key advantages of TD methods over Dynamic Programming and Monte Carlo methods

6. Identify the empirical benefits of TD learning

7. TD can be used both in continuing tasks and episodic tasks (because TD updates at every time step!)

8. Both TD(0) and MC methods converge to the true value function asymptotically, given that the environment is Markovian.

9. TD and MC use sample updates, DP uses expected updates.

10. Suppose we have current estimates for the value of two states: $V(A) = 1, V(B) = 1$ in an episodic setting. We observe the following trajectory: A, 0, B, 1, B, 0, T where T is a terminal state. Apply TD(0) with step-size, $\alpha = 1$, and discount factor, $\gamma = 0.5$. What are the value estimates for state A and state B at the end of the episode?

Solution: My mistake was that I thought I need to update all the way to the end, then backpropagate the state value back. Actually, here are the transition pairs, (S, R, S') = $(A, 0, B)$, $(B, 1, B)$, and $(B, 0, T)$. We can update the value estimate at each time step.

For example, after observing $(A, 0, B)$, we have

$$V(A) \leftarrow V(A) + \alpha[R + \gamma V(B) - V(A)]$$

Note that in above, B is the next state S_{t+1}.

$$V(A) = 1 + 1[0 + 0.5 * 1 - 1]$$

So, $V(A) = 0.5$ and $V(B) = 1$ (remains the same).

Lesson 4: TD for control

11. Explain how generalized policy iteration can be used with TD to find improved policies

12. Describe the Sarsa Control algorithm

13. Understand how the Sarsa control algorithm operates in an example MDP

14. Analyze the performance of a learning algorithm

Lesson 5: Off-policy TD control: Q-learning

15. Describe the Q-learning algorithm

16. Explain the relationship between Q-learning and the Bellman optimality equations.

17. Apply Q-learning to an MDP to find the optimal policy

18. Understand how Q-learning performs in an example MDP

19. Understand the differences between Q-learning and Sarsa

20. Understand how Q-learning can be off-policy without using importance sampling

21. Describe how the on-policy nature of SARSA and the off-policy nature of Q-learning affect their relative performance

Lesson 6: Expected Sarsaz

22. Describe the Expected Sarsa algorithm

23. Describe Expected Sarsa's behaviour in an example MDP

24. Understand how Expected Sarsa compares to Sarsa control

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

26. Explain how Expected Sarsa generalizes Q-learning

7 Some Notes

7.1 On-policy vs. Off-policy

These two concepts are introduced in Chapter 5.

On-policy methods can be a special case of off-policy, where the target policy and behavior policy are the same.

on-policy	off-policy
simpler, considered first	greater variance, slower converge
Agent can pick actions	Agent can't pick actions
most obvious setup :)	learning with exploration playing without exploration
Agent always follows own policy	Learning from expert (expert is imperfect) Learning from sessions (recorded data)

	on-policy	off-policy
value based	Monte Carlo Learning	
	TD(0)	Q-Learning
	SARSA	DQN
	Expected SARSA	Double DQN
	n-Step TD/SARSA	Dueling DQN
	TD(\lamda)	
policy based	REINFORCE	
	REINFORCE with Advantage	
actor-critic	A3C	DDPG
	A2C	TD3
	TRPO	SAC
	PPO	IMPALA

7.2 argmax vs. max

$$\arg\max_x f(x) = f^{-1}\max_x f(x)$$

$$\max f(x) = f(\arg\max f(x))$$

$\arg\max f(x)$ is the function **argument** x at which the maximum of f occurs, and $\max f(x)$ is the **maximum** value of f.

For example, $f(x) = 100-(x-6)^2$, then $\max_x f(x) = 100$ and $\arg\max_x f(x) = 6$.

7.3 Gradient Descent vs. Gradient Ascent

The gradient of a continuous function f = the vector that contains the partial derivatives $\frac{\partial f(p)}{\partial x_i}$ computed at that point p. The gradient is finite and defined if and only if all partial derivatives are also defined and finite. The gradient formula:

$$\nabla f(x) = [\frac{\partial f(p)}{\partial x_1}, \frac{\partial f(p)}{\partial x_2}, \cdots, \frac{\partial f(p)}{\partial x_{|x|}}]^T$$

When using the gradient for optimization, we can either conduct gradient descent or gradient ascent.

Gradient Descent Gradient Descent is an iterative process through which we optimize the parameters of a ML model. It's particularly used in NN, but also in logistic regression and support vector machines (SVM). It is the most typical method for iterative minimization of a cost function. Its major limitation consists of its guaranteed convergence to a local, not necesarily global, minimum.

A hyperparameter (i.e. pre-defined parameter) α (learning rate), allows the fine-tuning of the process of decent. In particular, we may descent to a global minimum. The gradient is calculated with respect to a vector of parameters for the model, typically the weight w. In NN, the process of applying gradient descent to the weight matrix is called backpropagation of the error.

Backpropagation uses the sign of the gradient to determine whether the weights should increase or decrease. The sign of the gradient allows us to direction of the closet minimum to the cost function. For a given α, we iteratively optimize the vector w by computing

$$w_{n+1} = w_n - \alpha \nabla_w f(w)$$

At step n, the weights of the NN are all modified by the product of the hyperparameter α times the gradient of the cost function, computed with those weights. If the gradient is positive, then we decrease the weights; if the gradient is negative, then we increase the weights.

Gradient Ascent Gradient ascent works in the same manner as gradient descent. The only difference is that gradient ascent maximize functions (instead of minimization).

$$w_{n+1} = w_n + \alpha \nabla_w f(w)$$

Gradient descent works on **convex functions**, while gradient ascent works on **concave functions**.

Summary

- The gradient is the vector containing all partial derivatives of a function in a point
- We can apply gradient descent on a convex function, and gradient ascent on a concave function
- Gradient descent finds the nearest minimum of a function, gradient ascent finds the nearest maximum
- We can use either form of optimization for the same problem if we can flip the objective function.

Gradient vs. Derivative

- A directional derivative = a slope in an arbitrary specified direction.
- A directional derivative is a rate of change of a function in any given direction.

- Gradient = a vector with slope of the function along each of the coordinate axes.
- Gradient indicates the direction of **greatest** change of a funciton of more than one variable.
- Gradient vector can be interpreted as the 'direction and rate of fastest increase'.

Differential vs. Derivative

- Differential is a subfield of calculus that refers to infinitesimal difference in some varying quantity
- Differential represents an equation that contains a function and one or more derivatives of that function
- The function which represents the relationship between the dependent and the independent variables is unknown

- The derivative of a function is the rate of change of the output value with respect to its input value
- Derivative represent the instantaneous change in the dependent variable with respect to its independent variable
- The function which represents the relationship between the variables is known

Loss/Cost/Objective function

- **Loss function** is usually a function defined on a data point, prediction and label, and measures the penalty. For example: square loss in linear regression, hinge loss in SVM, 01 loss in theoretical analysis
- **Cost function** is usually more general. It might be a sum of loss functions

over all training set plus some model complexity penalty. For example: MSE and SVM cost function

- **Objective function** is the most general term for any function that you optimize during training. For example, a probability of generating training set in maximum likelihood approach is a well defined objective function, but it is not a loss function nor cost function

We may say that, a loss function is a part of cost function which is a type of an objective function.

From wikipedia, in mathematical optimization and decision theory, a loss function or cost function (sometimes also called an error function) is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function. An objective function is either a loss function or its opposite (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case it is to be maximized.

Surrogate vs. Approximation I think surrogate and approximation can be used interchangeably.

Surrogate loss function is used when the original loss function is inconvenient for calculation.

References

(2021). Minimax.