Notes of Reinforcement Learning: An Introduction

Yanqing Wu

Update: April 14, 2023

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 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 Multi-armed Bandits

- Evaluative: feedback depends solely on the taken action, indicating results (in RL)
- Instructive: feedback is independent of the taken action, indicating correct actions (in SL)
- 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

[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

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.

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*

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

$N_t(a)$, 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.

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.

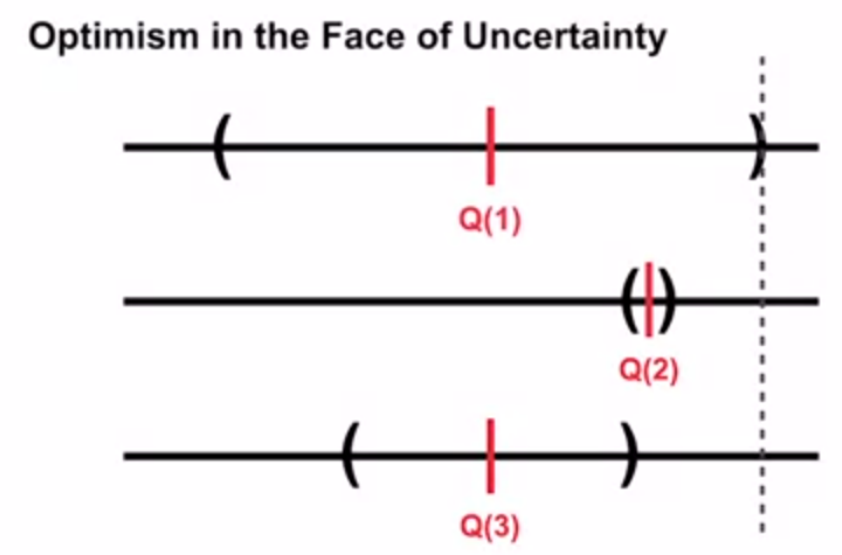


Figure 2: UCB Example with 3 actions. The interval represents the confidence level of the action. Larger confidence, smaller interval. Given the 3 actions, we select the one with highest upper bound. We update the interval based on received rewards.

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

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

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

$\overline{R_t} \in \mathbb{R}$ 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 (no interpretation on rewards), 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.

Associative search (contextual bandits)

- extend nonassociative bandit problem to the associative setting
- at each time step, the bandit is different
- learn a different policy for different bandits
- prepare for the full RL problems

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

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 (nonstationary) 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 work 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:

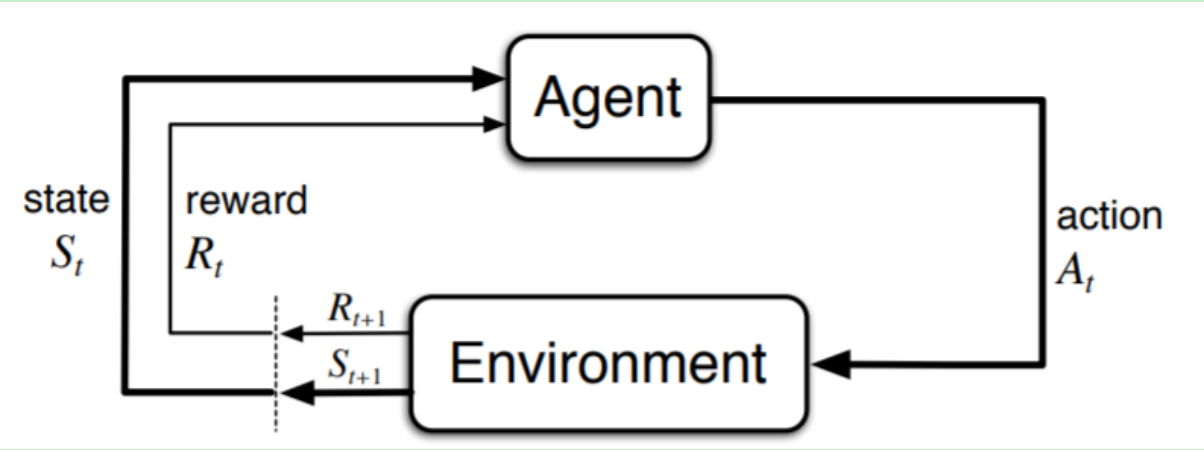


Figure 3: The agent-environment interaction in a MDP

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

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.

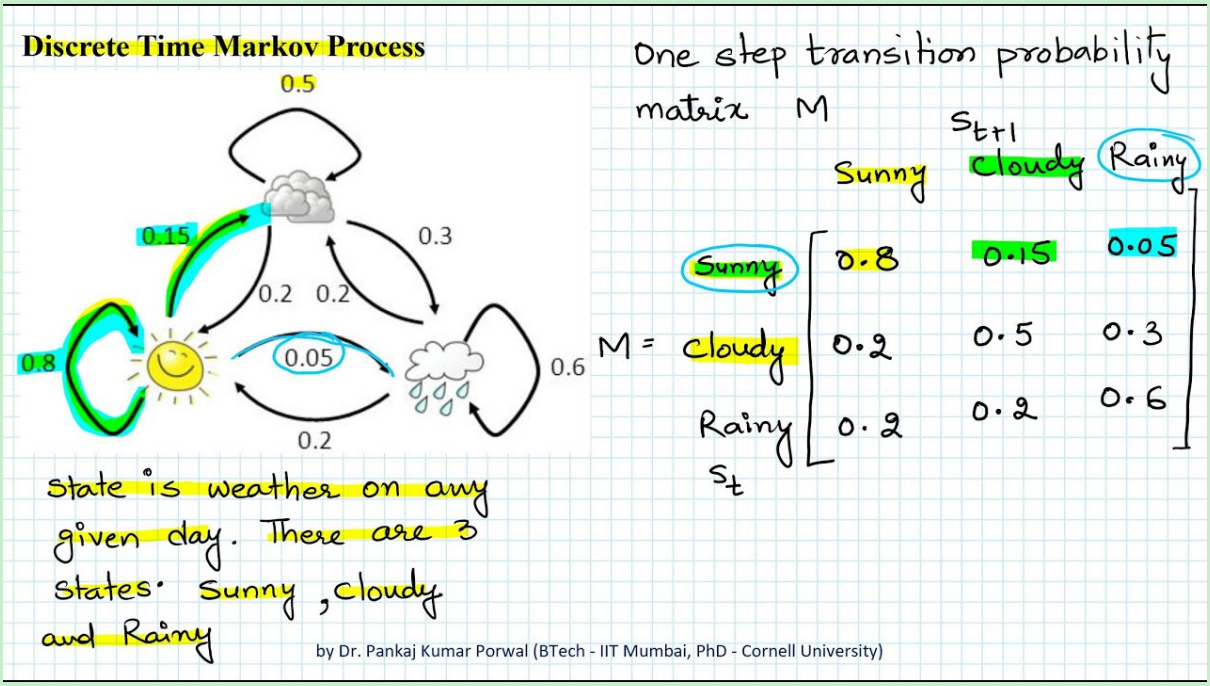
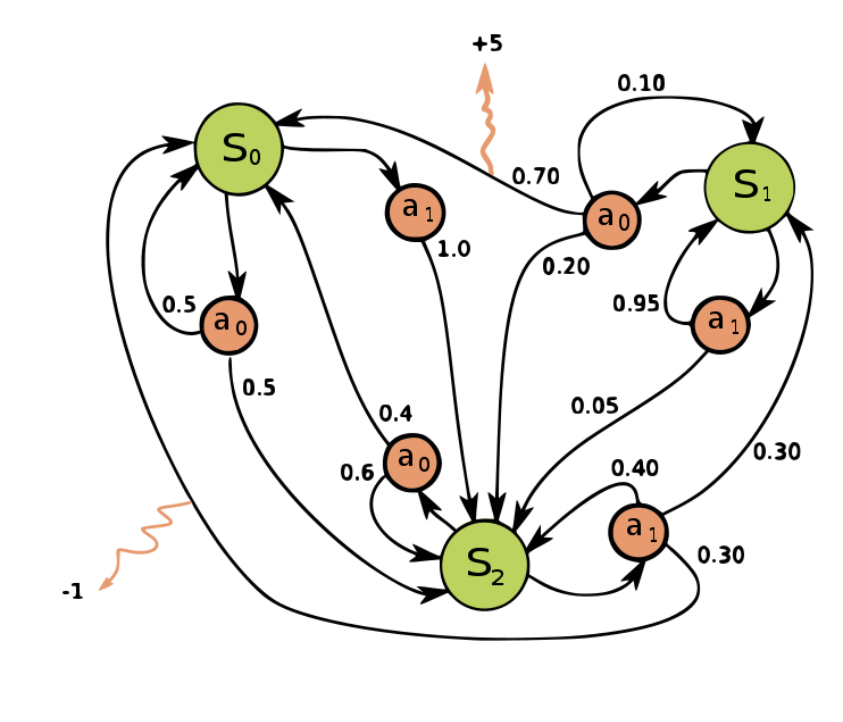


Figure 4: An example of state transition probability.



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*

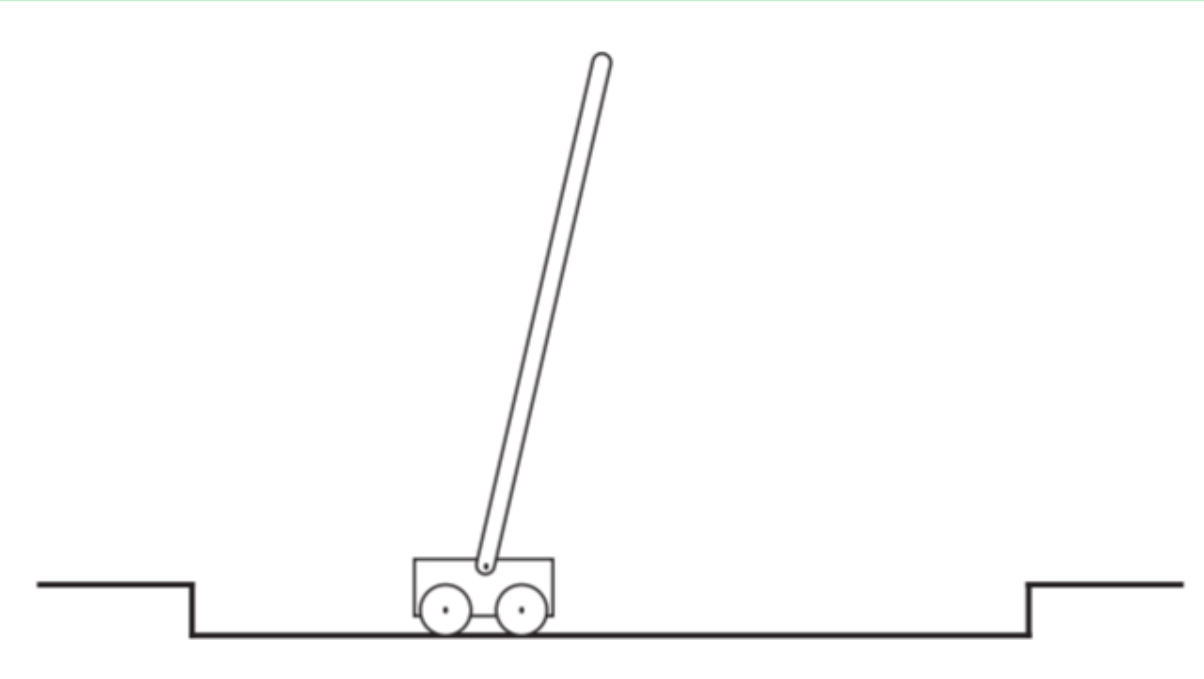
$$G_t = \sum_{k=0}^{\infty} R\gamma^k = R\frac{1}{1-\gamma}$$

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

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

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.

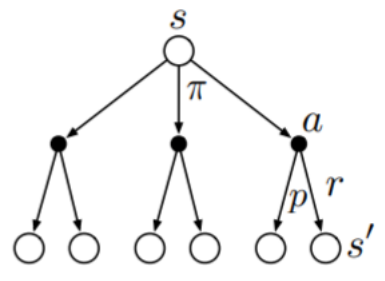


Figure 6: Backup diagram for v_π

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:*

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

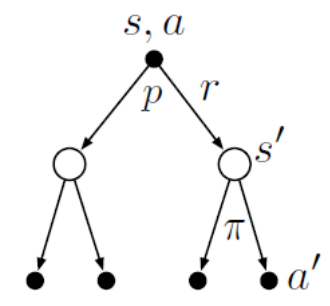
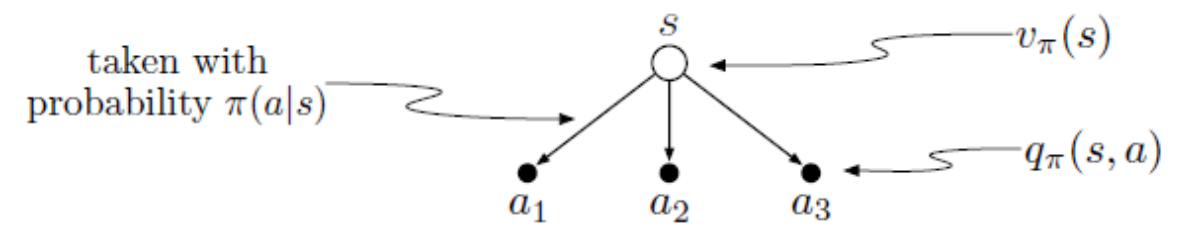
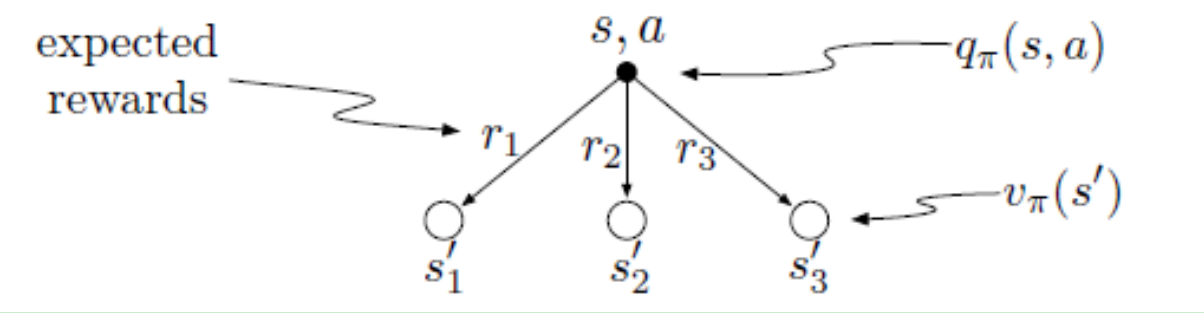


Figure 7: Backup diagram for q_π



(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_*,

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

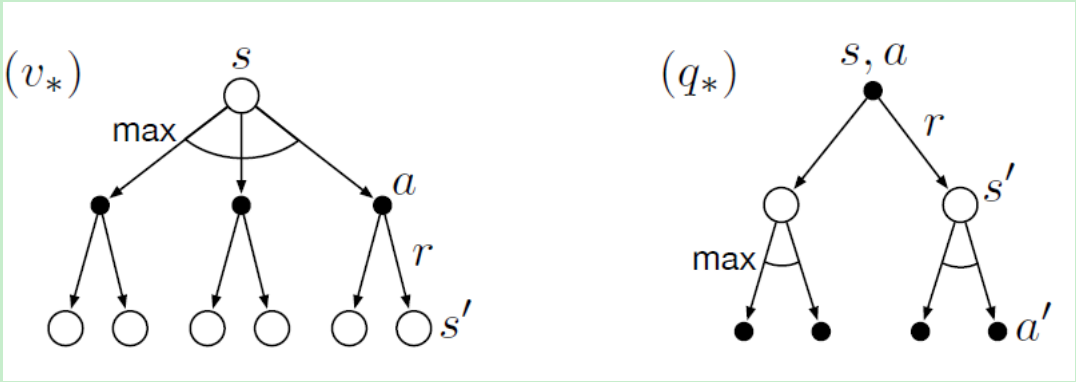


Figure 9: Backup diagram for v_* and q_*

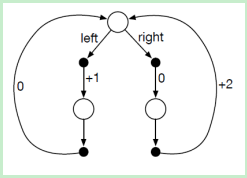
v_* is equal to the maximum of the term $\sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$ over all actions, while π_* is the particular action achieves the maximum.

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 $\gamma = 0.9$.



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

Left Policy

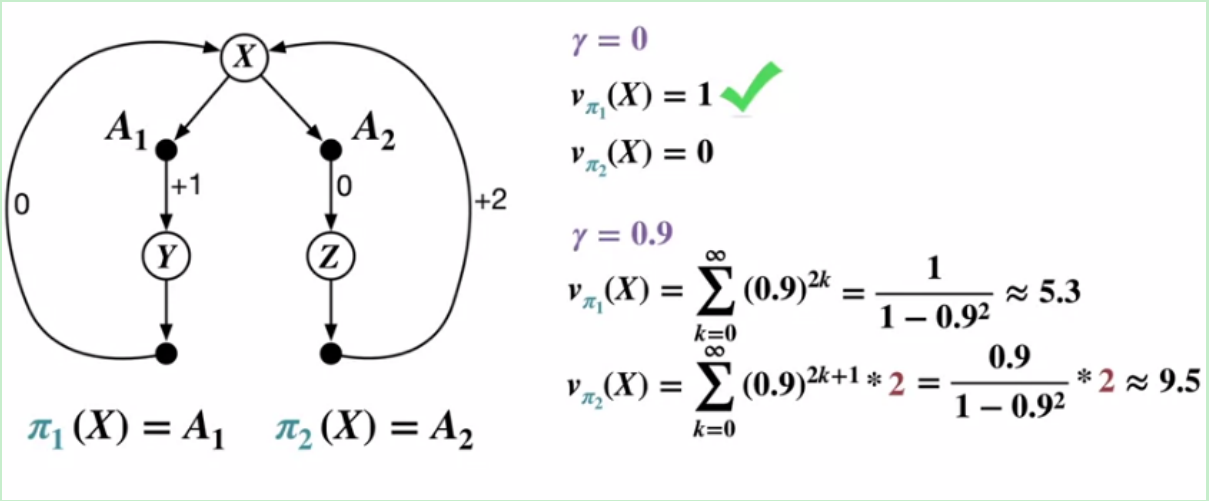
- $v(s_0) = 1 + 0.9 \cdot v(s_L)$
- $v(s_L) = 0 + 0.9 \cdot v(s_0)$
- $v(s_R) = 2 + 0.9 \cdot v(s_0)$

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 $\gamma = 0.9$.

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 goal of the agent is to maximize cumulative reward in the long run. In addition, The reward is a way of imparting to the agent what goal want to achieved, not how you want to achieved the goal.

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

Each value function are always associated with some policies. For example, the state value function:

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

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 length, 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 $v_k = v_\pi$ 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:

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

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

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 $\Delta < \theta a$

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()
```

```
        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.

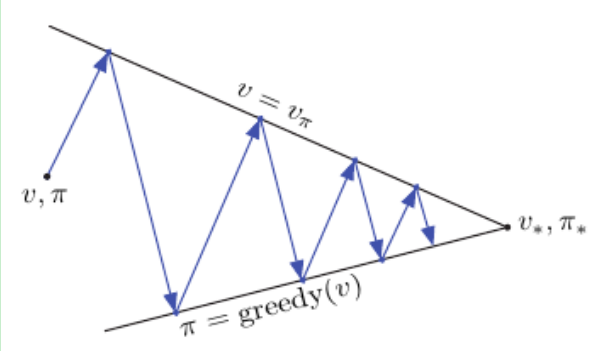
Asynchronous DP are in-place iterative DP that updates state value in random order. Async DP still needs to update all state values for convergence. We can also intermix policy iteration and value iteration, as for systematic DP, to produce an async truncated policy iteration.

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. 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 (greedy operation) 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.)

Problem	Bellman equation	Algorithm
Prediction (evaluation)	Bellman Expectation Equation	Iterative policy evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

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 bootstrapping 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

First *learning* method for estimating value functions and discovering optimal policies. Here we don't have any knowledge of the environment dynamics, we learn only by experience.

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 an 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 (because for $v(s)$, we need to do a one-step lookahead, which needs the model, i.e, transition probability $p(s', r|s, a)$); 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), $\forall sin\mathcal{S}$

$Q(s,a) \in \mathbb{R}$ (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 $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ 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 section (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:

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

To ensure the guarantee of convergence for MC, we made two unlikely assumptions:

- exploring start (not useful in real world interaction, some starting conditions aren't helpful)
- policy evaluation on infinite number of episodes.
  - soln 1: approximates q_{π_k} in each policy evaluation. Making bounds.
  - soln 2: early stop of policy evaluation. Example: value iteration (or in-place value iteration)

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.*

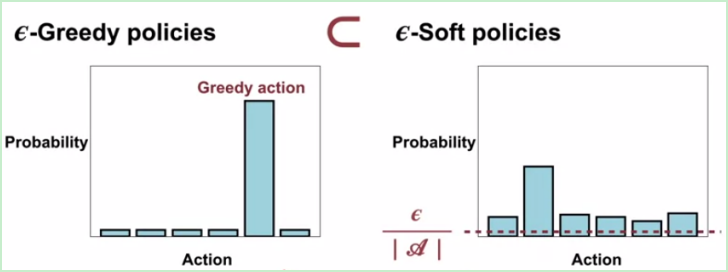
Definition 5.2 ***Off-policy method****: evaluate or improve a policy different from the one used to generate the data (make decisions). The policy being evaluated/improved is target policy, and the one generate behavior is behavior policy.*

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 $\epsilon > 0$

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 $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ 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.

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

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

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:

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

The ordinary IS formula:

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

where $\mathcal{T}(s)$ is the set of all time steps in every episode where state s has been visited. $T(t)$ is the first time of termination after timestep t, and $G(t)$ is the return after the t to $T(t)$ trajectory from behavior policy.

The weighted IS is:

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

To understand the difference, consider the estimates after observing a single return:

- for the weighted average, the ratio cancels out, and the estimate is equal to the return observed from behavior policy. It is a reasonable estimate but its expectation is $v_b(s)$ rather than $v_\pi(s)$, so it's biased.
- for the ordinary IS, the expectation is $v_\pi(s)$, so it's not biased but it can be a bit extreme. If the trajectory is 10 times more likely under π than under b, the estimate would be 10 times than the observed return, which would be far from the actual observed return. Note that the variance for the ordinary estimate can be unbounded (*infinite variance*).

5.6 Incremental Implementation

We want to implement those MC prediction methods on an episode-by-episode basis.

To do this, we will take inspiration from Chapter 2.4 where we incrementally computed Q estimates. For the on-policy method, the only difference is that here we average returns whereas in Chapter 2 we averaged rewards. For off-policy method, we need to distinguish between ordinary importance sampling and weighted importance sampling.

In ordinary IS, the returns are scaled by the importance sampling ratio, then simply averaged using ordinary IS formula:

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

For these methods, we can use the incremental methods from chapter 2, but using the scaled returns instead of reward.

For weighted IS, we have to form a weighted average of the returns, and use a slightly different incremental algorithm.

Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i (for example $\rho_{t:T(t)-1}$). We wish to form the estimate (for $n \geq 2$):

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

and keep it up to date as we obtain a single additional return G_n. In addition to keeping track of V_n, we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for $V_n (n \geq 1)$ is

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

and

$$C_{n+1} \doteq C_n + W_{n+1}$$

A complete pseudocode is on the book pg 110.

5.7 Off-policy Monte Carlo Control

In this book, we consider two classes of learning control methods:

- on-policy methods: estimate the value of a policy while using it for control
- off-policy methods: these two functions are separated (target vs. behavior policies)

Off-policy MC control follow the behavior policy while learning about improving the target policy. A complete pseudocode is on book pg 111.

A potential problem is that the method only learns from the tails of episodes, when all of the remaining actions are greedy. If non-greedy actions are common,

then learning will be slow, particularly for states appearing in early portions of long episodes.

Maybe incorporating TD-learning can help. Also if $\gamma \leq 1$, TD can help.

5.8 Discounting-aware Importance Sampling

idea: use partial termination (early break), up to horizon h, to reduce variance in off-policy methods.

flat partial return

5.9 Per-decision Importance Sampling

Similar ideas as discounting-aware, i.e., ignore irrelevant factors to reduce variance. It has the version for ordinary IS; it is unbiased and has lower variance. It is unclear if there exists weighted per-decision IS.

5.10 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

5.11 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).

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:*

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

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 successor
- **expected update**: a complete distribution of all possible successors

Definition 6.3 ***TD error***:

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

If the array V does not change during the episode, 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

- they do not require a model of the environment.
- online, fully incremental updates (no delay, no waiting for the end of the episode like MC)
- for a fixed policy π, TD(0) has been shown to converge to v_π
- 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. In non-batch form, TD(0) may be faster than MC because it is moving toward a better estimate (even if it's not getting all the way there).

!! MC will always find the estimate that minimizes RMSE on train data, whereas

batch TD(0) always find the estimate that would be correct for the maximum likelihood model of the Markov process.

Important: look at book examples.

6.4 Sarsa: On-policy TD control

We consider transitions from state-action pair to state-action pair. Recall that without a model, for on-policy methods, we must estimate $q(s, a)$; otherwise we need lookahead and a model for $v(s)$.

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.

Convergence properties of SARSA depend on the nature of the policy dependence to Q. SARSA converges with probability 1 to an π_* and q_* as long as all the state-value pairs are visited an infinite amount of time and the policy converges in the limit to the greedy policy.

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' (behavior policy)	π	π
Updating Q (target policy)	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***,

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

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

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}.

Expected SARSA might use a policy different from the target policy π to generate behavior, becoming an off-policy algorithm. If π is the greedy policy while behavior is more exploratory, then Expected SARSA is Q-learning.

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 Maximization Bias and Double Learning

All the control algorithms so far involve maximization in the construction of their target policy:

- in Q-learning, we use the greedy policy given the current action values
- in SARSA, the policy is often ε-greedy

6.7.1 Maximization Bias

In these algorithms, a max is implicitly used over the estimated values, which can lead to a significant positive bias. To see this, consider a single state s in

which for all actions the true value $q(s, a)$ is zero but whose estimated values $Q(s, a)$ are uncertain and thus distributed some above and some below zero. The max will always be positive, when the max over the true values is zero. It's *maximization bias*.

6.7.2 How to avoid maximization bias: double learning

One way to view the problem is that it is due to using the *same* samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divide the samples into two sets and learn two independent estimates, $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$.

- we use Q_1 to determine the maximizing action $A_* = \arg\max_a Q_1(a)$
- we use Q_2 to provide an estimate of its value $Q_2(A_*) = Q_2 \arg\max_a Q_1(a)$

This estimate will be unbiased in the sense that $\mathbb{E}[Q_2(A_*)] = q(A_*)$. We can reverse both to produce another unbiased estimate $Q_1(A_*) = Q_1 \arg\max_a Q_2(a)$ and we have **double learning**.

6.7.3 Double Q-learning

The idea of double learning extends naturally to algorithms for full MDPs. The behavior policy can use both estimates. There are also double versions of SARSA and Expected SARSA.

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

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

Init $Q_1(s, a)$ and $Q_2(s, a)$, $\forall s \in \S^+$, $a \in \mathcal{A}(s)$, such that that $Q(terminal, \cdot) = 0$

Loop for each episode:

Init S

Loop for each step of episode:

Choose A from S using the policy ε-greedy in $Q_1 + Q_2$

Take action A, observe R, S'

With 0.5 probability

$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A)]$

else:

$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A)]$

$S \leftarrow S'$

until S is terminal

6.8 Games, afterstates, and Other special cases

In chapter 1 we presented a TD algorithm for learning Tic-Tac-Toe. It learned something more like a state-value function, but it's neither an action-value nor a state-value in the usual sense. In a conventional state, the player has the option to play actions, but in tic-tac-toe we evaluate after the player has taken the action. It's not an action-value either because we know the state of the game after action is played. Let's call these afterstates and the value functions over these afterstate value-functions.

We can use these to evaluate values more efficiently.

6.9 Learning Objectives (UA RL MOOC)

Lesson 1: Introduction to Temporal Difference Learning

1. Define temporal-difference learning

We want to update the return, but not want to wait till the end of the upate. We want to incrementally update. TD is 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

We can update our estimate at each time step.

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

Unlike DP, TD can directly learn from experience. Unlike MC, TD can update the values on every step TD also converges faster than MC

6. Identify the empirical benefits of TD learning

TD converges faster to a lower final error in the random walk environment.

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 GPI can be used with TD to find improved policies

Recall GPI with MC improve policy after each episode. Instead, GPI with TD will improve the policy after just one policy evaluation step.

12. Describe the Sarsa Control algorithm

see chap 6.4 (SARSA, uses state-action transition)

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

see the Windy Gridworld example

14. Analyze the performance of a learning algorithm

see the Windy Gridworld example

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

15. Describe the Q-learning algorithm

Q-learning, like SARSA, solves the Bellman equation using samples from environment. But instead of using the standard Bellman equation, Q-learning uess the Bellman's optimality equation for action values. The optimality equations enable Q-learning to directly learn Q_* instead of switching between policy improvement and policy evaluation steps.

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

SARSA is sample-based version of *policy iteration* which uses Bellman equations for action values, that each depend on a fixed policy.

Q-learning is a sample-based version of *value iteration* which iteratively applies the Bellman optimality equation.

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

see Windy gridword

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

see Windy gridword

19. Understand the differences between Q-learning and Sarsa

SARSA (on-policy)

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

Q-learning: (off-policy)

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

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

In SARSA, the agent bootstraps off the value of the value of the action it's going to take next, which is sampled from its behavior policy. Q-learning instead bootstraps off of the *largest* action value in its next state. The is like sampling

an action under an estimate of the optimal policy rather than the behavior policy. Since Q-learning learns about the best action it could take rather the actions it actually takes, it is learning off-policy.

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

cliff walking example, Q-learning follows optimal policy, but its own ϵ-greedy action selection makes it fall off and result in lower reward.

Lesson 6: Expected Sarsa

22. Describe the Expected Sarsa algorithm

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

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

24. Understand how Expected Sarsa compares to Sarsa control

Expected SARSA has a more stable update target than SARSA. In general, Expected SARSA has much lower variance than SARSA (but at a cost of computation), as it removes the variance due to the random selection of A_{t+1}.

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

The expectation over actions is computed independently of the action actually selected in the next state. In fact, π in the Expected SARSA update rule does not need to be equal to b.

26. Explain how Expected Sarsa generalizes Q-learning

If the target policy is greedy w.r.t its action value estimates, then only the highest value action is considered in the expectation. Just like Q-learning, computing the maximum over actions in the next state.

27. Q-learning does not learn about the outcomes of exploratory actions because the update in Q-learning only learns about the greedy action

28. SARSA, Q-learning, and Expected SARSA have similar targets on a transition on a *terminal* state, because the target in this case only depends on reward.

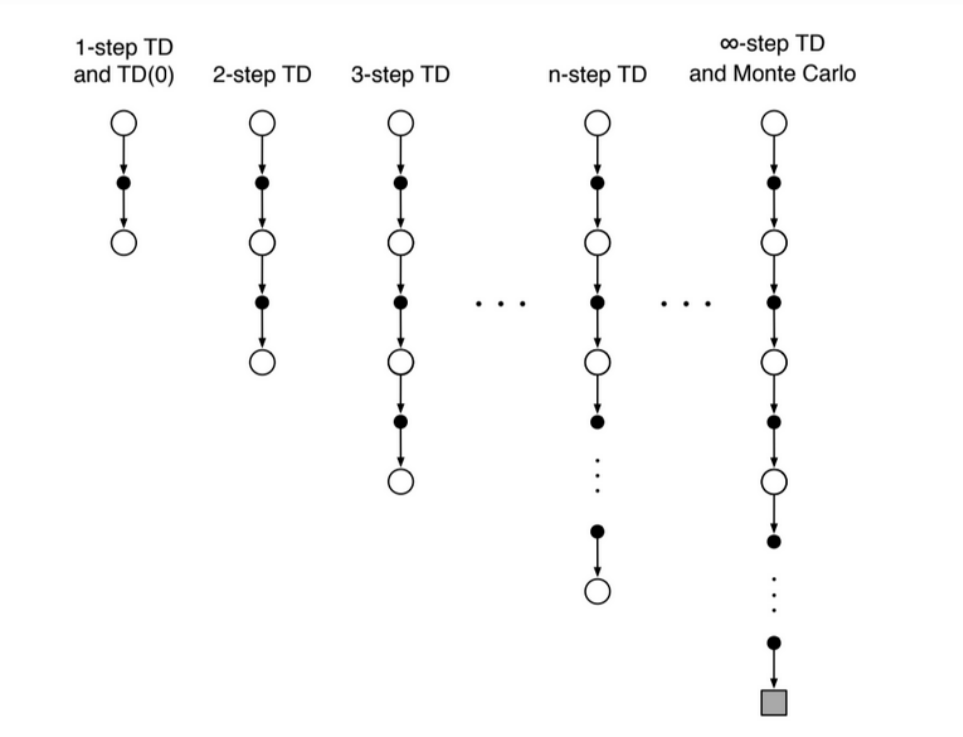
7 n-step Bootstrapping

- unify the MC and one-step tabular TD methods
- n-step TD methods generalize both by spanning a spectrum with MC on one end, and one-step TD at the other
- n-step methods enable bootstrapping over multiple time steps

7.1 n-step TD Prediction

Consider estimate v_π from sample episodes generated using π

- MC methods perform an update for each visited state based on the entire sequence of rewards from that state until the end of the episode
- one-step TD methods perform an update based on the next reward only
- n-step: in-between



(a)

Figure 10: Backup diagrams for n-step methods

Still TD: we bootstrap because we change an earlier estimate based on how it

differs from a later estimate (only that the later estimate is now n steps later)

7.1.1 Targets

notation: $G_t^{(n)} = G_{t:t+n}$, the return from time step t to $t+n$

$$
\begin{aligned}
n = 1 \quad (TD(0)) \qquad & G_t^{(1)} = R_{t+1} + \gamma V_t(S_{t+1}) && (7.1)\\
n = 2 \qquad & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}) && ()\\
\vdots \qquad & \vdots && ()\\
n = \infty \quad (MC) \qquad & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T && (7.2)
\end{aligned}
$$

- In MC, the target is the return
- In one-step TD, the target is the one-step return, that is the first reward plus the discounted estimated value of the next state
- $V_t : \mathcal{S} \to \mathbb{R}$ is the estimate at time t of v_π.

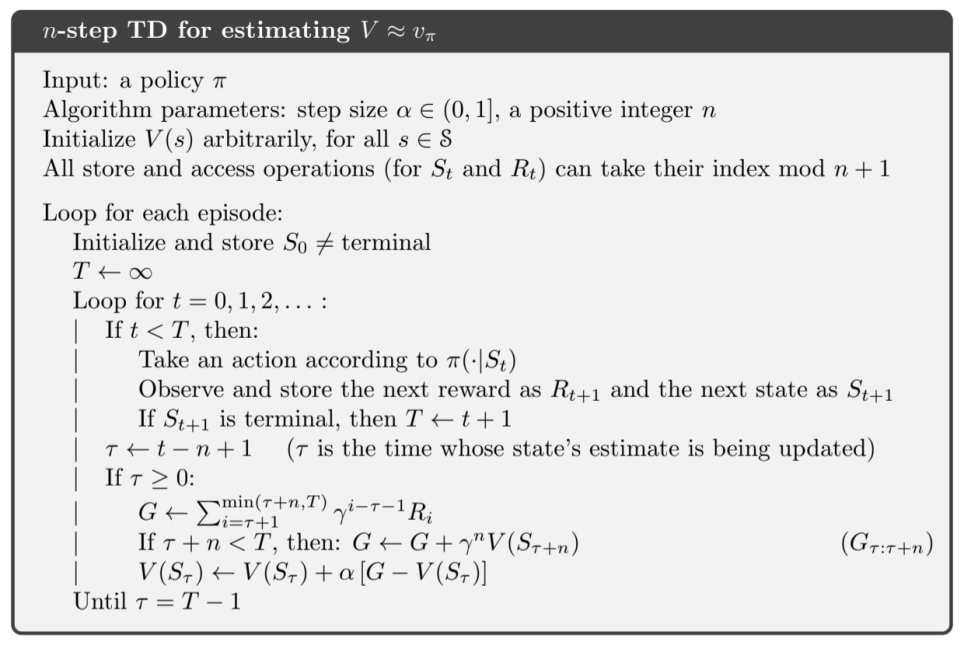
n-step target:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \qquad (7.3)$$

- All n-step returns can be considered approximations of the full return, truncated after n steps and then corrected for the remaining steps by $V_{t+n-1}(S_{t+n})$
- n-steps return for $n \geq 1$ involve future rewards that are not available from the transition $t \to t+1$; we will see eligibility traces for an online method not involving future rewards
- plugging this into the state-value learning

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_t^{(n)} - V_{t+n-1}(S_t)] \qquad (7.4)$$

- no changes are made during the first (n-1) steps of each episode. To make up for that, an equal number of updates are made after the termination of the episode, before starting the next.



(a)

Figure 11: n-step TD algorithm

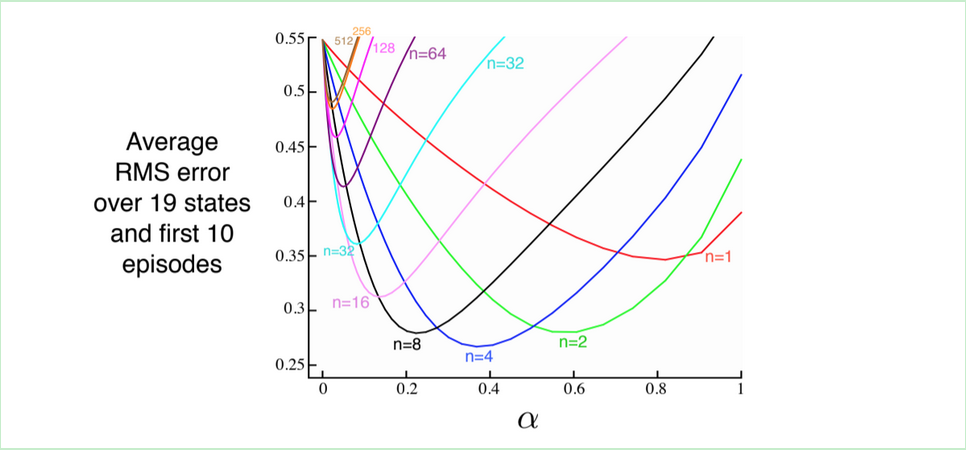
7.1.2 Error-reduction Property

The n-step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n}. An important property of the n-step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is in a worst-state sense. The worst error of the expected n-step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}.

$$\max_s |\mathbb{E}_\pi[G_t^{(n)}|S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{7.5}$$

This is called the **error reduction property**. We can use this to show that all n-steps methods converge to the correct predictions under appropriate technical conditions.

We use this to type of graph to show the effect of n



(a)

Figure 12: Performance of n-step TD methods as a function of α

7.2 n-step SARSA

- use n-step not just for prediction but also for control
- change from states to state-action pairs, and use a ε-greedy policy
- The backup diagrams for n-step SARSA are like those of n-step TD, except that the SARSA ones start and end in a state-action node. We redefine the n-step return in terms of estimated action-values instead of state-values

$$G_{t:t+n} = R_{t+1} + \gamma R_{T+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \qquad (7.6)$$

- and plug that into our GPI update

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha\big(G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\big) \qquad (7.7)$$

The values of all other states remain unchanged, $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$ for $s \neq S_t$ or $a \neq A_t$.

7.2.1 Expected SARSA

What about n-step version of expected SARSA? The backup diagram consists of a linear string of sample actions and states and the last element is a branch over all action possibilities, weighted by their probability of occurring under π.

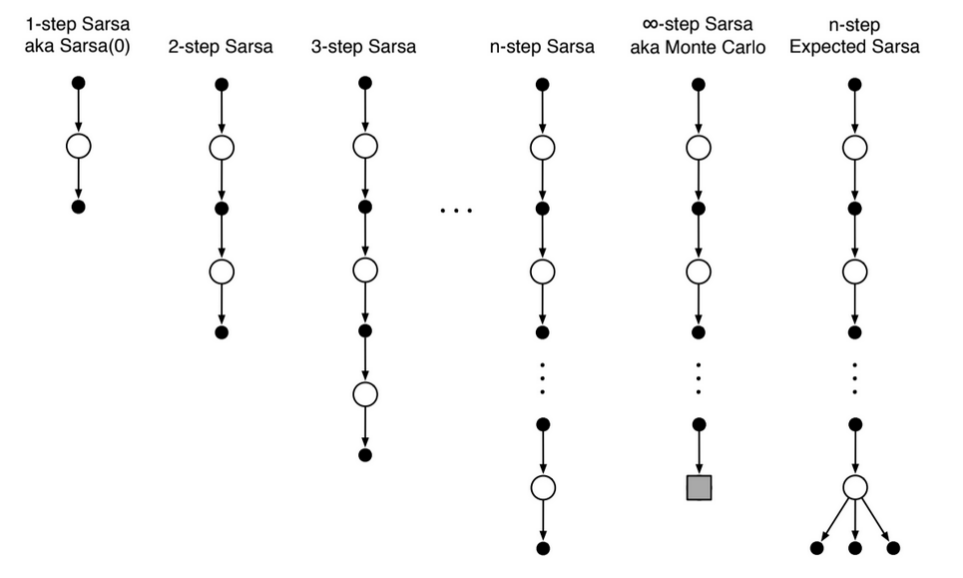
n-step Sarsa for estimating $Q \approx q_*$ or q_π

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

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

(a)

Figure 13: n-step SARSA



(a)

Figure 14: n-step SARSA backup diagram

The n-step return is here defined by:

$$G_{t:t+n} = R_{t+1} + \gamma R_{T+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}, A_{t+n}) \qquad (7.8)$$

Where $\bar{V}_t(s)$ is the expected approximate value of state s, using the estimated action values and the policy

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s,a)$$

- Expected approximate values are important!
- if s is terminal, then its approximated value is defined to be 0.

7.3 n-step Off-policy Learning (by importance sampling)

- To use the data from behavior policy b, we must take into account the relative probability of taking the actions that were taken
- In n-step methods, the returns are constructed over n steps so we are interested in the relative probability of just these n actions

For example, to make a simple version of n-step TD, we can weight the update for time t (made at time $t+n$) by the importance sampling ratio

$$\rho_{t:h} = \prod_{k=t}^{min(h,T-1)} \frac{\pi(A_k \| S_k)}{b(A_k \| S_k)} \tag{7.10}$$

If we plug this into our update, we get

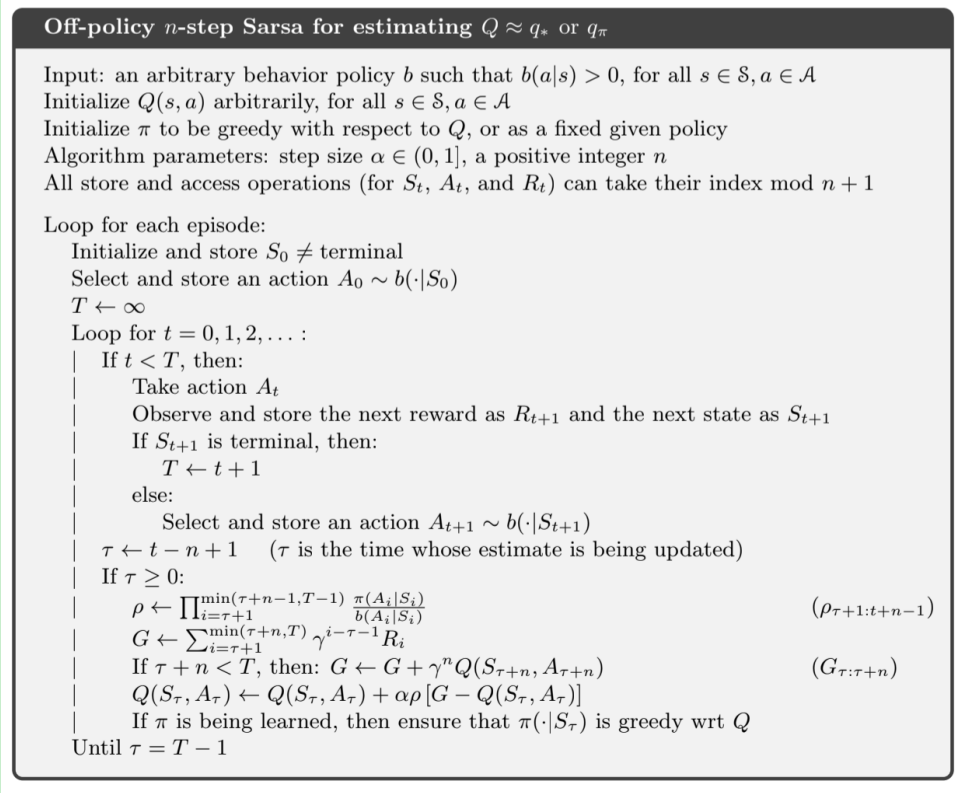
$$V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1}(G_{t:t+n} - V_{t+n-1}(S_t))$$

Similarly, the n-step SARSA update can be extended for the off-policy method

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

Note that the IS ratio here starts one step later than from n-step TD because we are updating a state-action pair (we know that we have selected the action so we need IS only for the subsequent actions).

The off-policy version for **expected SARSA** would be the same except that the IS ratio would use $\rho_{t+1:t+n-2}$ instead of $\rho_{t+1:t+n-1}$ because all possible actions are taken into account *in the last state*, the one actually taken has no effect and does not need to be corrected for.



Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

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

(a)

Figure 15: n-step SARSA off-policy algorithm

why Q-learning can be off-policy w/o IS?

Q-learning bootstraps off of the largest action value in its next state. This is like sampling an action under an estimate of the optimal policy rather than the behavior policy. Q-learning learns about the best action it could take rather the actions it actually takes, so it's off-policy.

why expected SARSA can be off-policy w/o IS?

the expectation over actions is computed independently of the action actually selected in the next state, so it's off-policy

why Q-learning doesn't use IS?

https://stats.stackexchange.com/questions/335396/why-dont-we-use-importa

For action values, the fist action does not play a role in the IS ratio (it has been taken!).

7.4 Per-decision Methods with Control Variates

The multi-step off-policy methods presented in 7.3 is conceptually clear but not very efficient. What about per-decision sampling (recall Chap 5.9).

1. The n-step return can be written recursively. For the n steps being ending at horizon h, the n-step return can be written $G_{t:h} = R_{t+1} + \gamma R_{t+1:h}$

2. Consider the effect of following behavior policy b is not the same as following policy π, so all the resulting experiences, including the first reward R_{t+1} and the next state S_{t+1}, must be weight by the importance sampling ratio for time t, $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

A simple approach would be to weight the right-hand side $R_{t+1} + \gamma G_{t+1:h}$ by ρ_t. A more sophisticated approach would be to use an off-policy definition of the n-step return ending in horizon h

$$G_{t:h} \doteq \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t)V_{h-1}(S_t) \tag{7.11}$$

where $t < h < T$, and $G_{h:h} \doteq V_{h-1}(S_h)$.

If $\rho_t = 0$ (trajectory has no chance to occur under the target policy), then instead of changing the target $G_{t:h}$ to be 0 and causing the estimate to shrink, the target is the same as the estimate and cause no change. The additional term $(1 - \rho_t)V_{h-1}(S_t)$ is called a **control variate** and conceptually is here to ensure the idea that if $\rho_t = 0$, then we should ignore the sample and don't change the estimate.

We can otherwise use the conventional learning rule that does not have explicit IS ratios, except the one in $G_{t:h}$.

7.4.1 For action values

For action values, the off-policy definition of the n-step return is a little different because the first action does not play a role in the IS ratio (it has been taken!).

The n-step on-policy return ending at horizon h can be written recursively, and

for action-values, the recursion ends with $G_{h:h} \doteq \bar{V}_{h-1}(S_h)$. An off-policy form with control variate is:

$$G_{t:h} \doteq R_{t+1} + \gamma\big(\rho_{t+1}G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1}Q_{h-1}(S_{t+1}, A_{t+1})\big) \quad (7.12)$$

$$= R_{t+1} + \gamma\rho_{t+1}\big(G_{t+1:h}Q_{h-1}(S_{t+1}, A_{t+1})\big) + \gamma\bar{V}_{h-1}(S_{t+1})\big) \quad (7.13)$$

for $t < h < T$.

- if $h < T$, then the recursion end with $G_{h:h} \doteq (S_h, A_h)$
- if $h = T$, it ends with $G_{T-1:T} \doteq R_T$. The resultant prediction algorithm is analogous to Expected SARSA.

Off-policy methods have higher variance and it's probably inevitable. Things to help reduce the variance:

- control variates
- adapt the step size parameter to the observed variance
- invariant updates

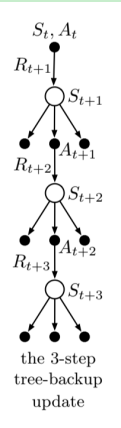
7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Off-policy without IS: Q-learning and Expected SARSA do it for the one-step case. Here, we present a multi-step algorithm: **the tree backup algorithm**.

A 3-step tree backup diagram

- 3 sample states and rewards
- 2 sample actions
- list of unselected actions for each state
- we have no sample for the unselected actions, and we bootstrap with their estimated values

So far, the target for the update of a node was combining the rewards along the way and the estimated values of the nodes at the bottom. Now we add to this the estimated values of the actions not selected. This is why it is called a tree



(a)

Figure 16: n-step tree

backup update: it is an update from the entire tree of estimated action values.

Each leaf node contributes to the target with a weight proportional to their probability of occurring under π.

- each first-level action a contributes with a weight of $\pi(a|S_{t+1})$
- the action actually taken, A_{t+1}, does not contribute and its probability $\pi(A_{t+1}|S_{t+1})$ is used to weight all the second level action values
- each second-level action a' thus has a weight of $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$

The one-step return (target) is the same as Expected SARSA (for $t < T-1$):

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \tag{7.14}$$

And the two-step tree-backup return is (for $t < T-2$):

$$\begin{aligned} G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \\ &\quad + \gamma \pi(A_{t+1}|S_{t+1}) \big(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \big) \\ &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2} \end{aligned} \tag{3}$$

The latter form suggests the recursive form of the n-step tree backup return (for

$t < T - 1$):

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

(7.15)

We then use this target in the usual action-value update from n-step SARSA:

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha \big(G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \big) \qquad (7.16)$$

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations can take their index mod $n + 1$

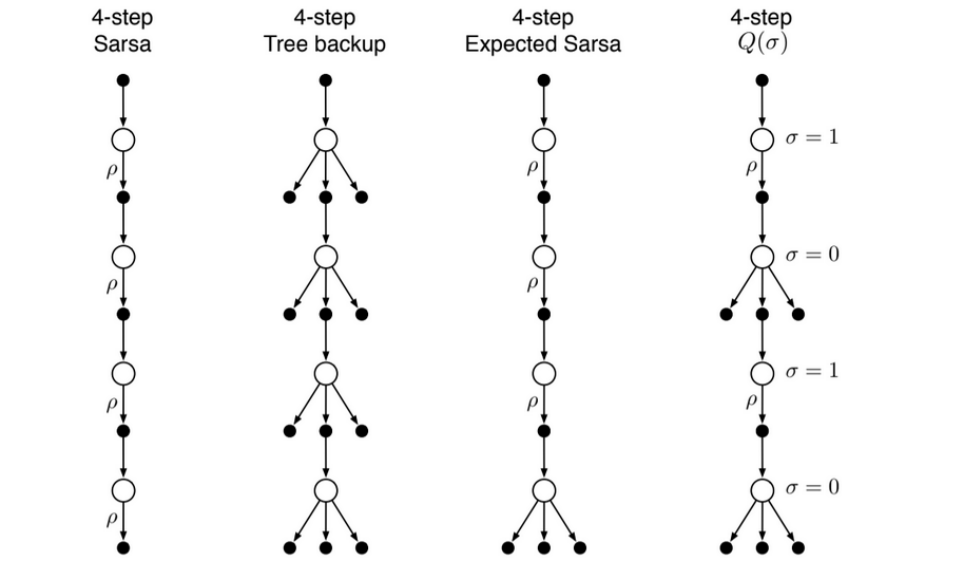
Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$:
  | Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
  | If S_{t+1}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

(a)

Figure 17: n-step tree backup algorithm

7.6 A Unifying Algorithm: n-step Q(σ)

So far we have considered 3 different kinds of action-value algorithms, the first 3 of the figure:



(a)

Figure 18: n-step backup diagrams. The rho's indicate half transitions on which IS is required in the off-policy case

- n-step SARSA has all sample transitions
- the tree backup has all state-to-action transitions branched w/o sampling
- n-step Expected SARSA has all sample transitions except for the last state which is fully branched with an expected value

Unification algorithm: last diagram.

Basic idea: decide on a step-by-step basis whether we want to sample as in SARSA or consider the expectation over all the actions instead, as in the tree backup update.

- let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t, 1 = full sampling, 0 = taking expectation
- the random variable σ_t might be set as a function of (s) or (s,a) at time t

This algorithm is called n-step Q(σ). To develop its equations, first we need to write the tree-backup n-step return in terms of the horizon $h = t + n$ and the expected approximate value $\bar{V}$:

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{h-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \bar{V}_{h-1}(S_{t+1}) - \gamma \pi(A_{t+1}|S_{t+1}) - Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \pi(A_{t+1}|S_{t+1}) \big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\big) + \gamma \bar{V}_{h-1}(S_{t+1}) \end{aligned} \tag{7.17}$$

After which this is exactly like the n-step return for SARSA with Control Variates, except with the action probability $\pi(A_{t+1}|S_{t+1})$ substituted for the IS ratio ρ_{t+1}. For Q(σ), we slide linearly between these two cases

$$\begin{aligned} G_{t:h} \doteq R_{t+1} + \gamma \big(\sigma_{t+1} \rho_{t+1} + (1 - \sigma_{t+1}) \pi(A_{t+1}|S_{t+1})\big) \big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\big) \\ + \gamma \bar{V}_{h-1}(S_{t+1}) s \end{aligned}$$

for $t < h \leq T$. The recursion ends with $G_{h:h} \doteq 0$ if $h < T$ or with $G_{T-1:T} = R_T$ if $h = T$

Once this return is well defined, we can plug it into the usual n-step SARSA upate.

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod $n+1$

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

(a)

Figure 19: n-step Q(σ)

7.7 Summary

- Range of TD methods between one-step TD and MC methods
- n-step methods look ahead to the next n rewards, states and actions
- all n-step methods require a delay of n steps before updating (because we need to know what happens in the next n steps) -> Eligibility Traces :)
- they also involve more computation than previous one-step methods (it's worthwhile, since one-step methods are limited)
- two approaches for off-policy learning have been explained:
  - with importance sampling. Simple but high variance
  - tree-backup updates (no IS). A natural extension of Q-learning to multi-

step case with stochastic target policies

8 Planning and Learning with Tabular Methods

- model-based: rely on *planning*; require a model of the environment (anything that an agent can use to predict how the environment will respond to its actions), e.g. dynamic programming and heuristic search
- model-free: rely on *learning*; use without a model, such as Monte Carlo and temporal-difference method

Similarities:

- both revolve around the computation of value functions
- both look ahead of future events, compute a backed up value and use it as an update target for an approximate value function

In this chapter we explore how model-free and model-based approaches can be combined.

8.1 Models and Planning

- distribution model: models that produce a description of all possibilities and their probabilities (e.g. MDP dynamics $p(s', r|s, a)$)
- sample models: models that produce just one of the possibilities, sampled according to the probabilities (e.g. HTHTHH flipping coin sequence)

Distribution models are stronger than sample models in that they can always be used to produce samples; however, sample models are easier to implement.

Definition 8.1 ***planning****, refer to any computational process that takes a* ***model*** *as input and produces or* ***improves a policy*** *for interacting with the modeled environment.*

$$\textit{model} \xrightarrow{\textit{planning}} \textit{policy}$$

- state-space planning: a search through the state space for an π_*, or an optimal path to a goal
- plan-space planning: a search through the space of plans

- includes evolutionary methods, and partial-order planning (ordering of steps is not completely determined at all stages of planning)
- hard to apply to the stochastic sequential decision problems
- not focused in this book

State-space planning methods' structure: (1) all state-space planning methods involve computing value functions to improve the policy, (2) they compute value functions by updates or backup operations applied to simulated experience.

$$\text{model} \rightarrow \text{simul. exp.} \xrightarrow{\text{backup}} \rightarrow \text{values} \rightarrow \text{policy}$$

State-space planning methods fits in the above structure, only differed by (1) update rule, (2) order of update, (3) how long the backed-up info is retained.

The heart of both planning and learning methods is the estimation of value functions by backing-up update operations.

- **planning** uses simulated experience generated by a model (e.g. DP)
- **learning** uses real experience generated by the environment (e.g. TD)

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in S$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain a sample next R, and S'

Apply one-step tabular Q-**learning** to S, A, R, S':

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

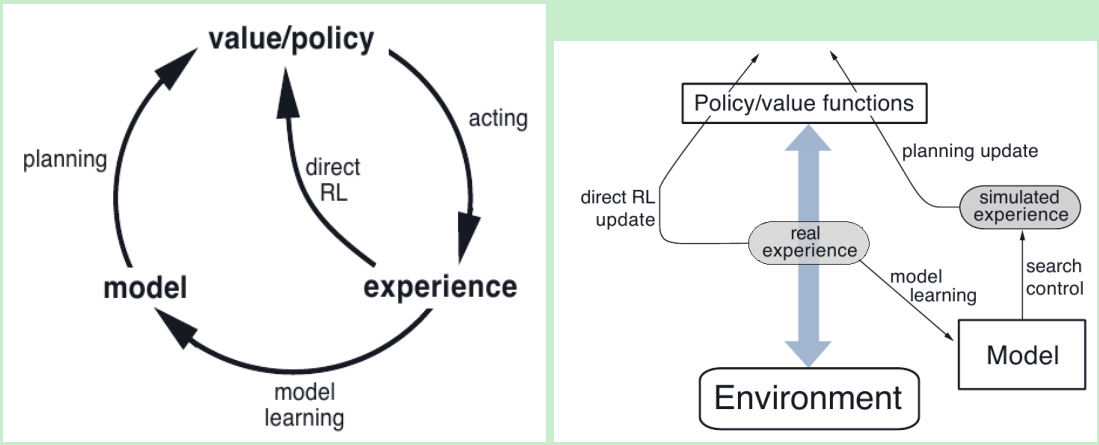
How does this differ from DP-value iteration? It does not **sweep** all states.

8.2 Dyna: Integrated Planning, Acting, and Learning

Problem: Both decision making and model learning are computation-intensive

Solution: To balance these two, we use a architecture to integrate the major functions in an online planning agent, called Dyna-Q.

- real experience: (1) *model-learning* (or indirect RL) to improve model



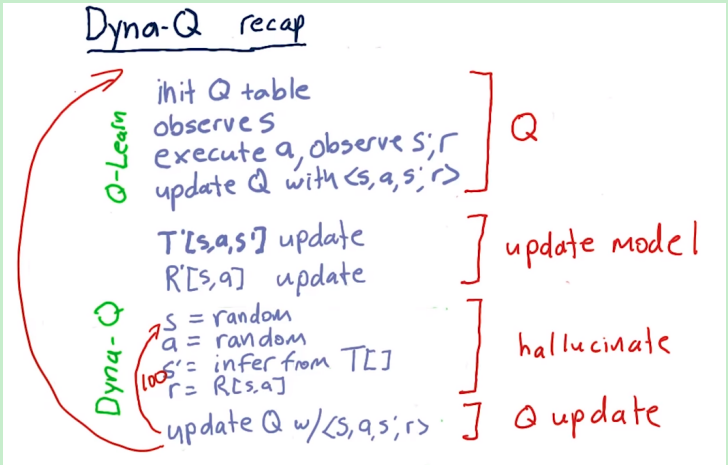
(a) (b)

Figure 20: (a) Dyna, balancing decision making and model learning (b) General Dyna Architecture

(more accurately match the real environment); (2) *direct RL* to improve value function and policy

- indirect methods: maximize the use of limited experience (better policy with fewer environmental interactions)
- direct methods are more simpler and are not affected by biases incurred by the model

- all planning, acting, model-learning, and direct RL occurring continually
  - planning here uses random-sample one-step tabular Q-**planning**
  - direct RL here uses one-step tabular Q-**learning**
- *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model



The Dyna Q uses both real world experience (which is expensive) and simulated (hallucinated) experience (which is cheap; more iterations are completed with simulated experience), thus accelerating training. The real experience is for learning, and simulated experience is for planning.

Tabular Dyna-Q

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

Loop forever:

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

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

(c) Take action A; observe resultant reward, R, and state, S'

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

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

(f) Loop repeat n times:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

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

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

8.3 When the Model Is Wrong

The model learning in the Dyna-Q may be incorrect when encountering stochastic environment. The model may not be exploratory enough to find the new (optimal) path when environment changes, sticking to the old path. We slightly modified the reward transition in Dyna-Q+ with $r + \kappa\sqrt{\tau}$ for small κ, where τ is the time steps that the (s,a) has not been tried (similar to how UCB incorporate time steps inside).

Read Blocking Task and Shortcut Task Examples in the book.

8.4 Prioritized Sweeping

Problem: search control select (s,a) uniform-randomly. Some (s,a) are not worth visiting

Solution: prioritized sweeping

! **Backward-focusing** of planning computations: in general, we want to work back not just from goal states but from any state whose value have changed.

As the frontier of useful updates propagates backward, it grows rapidly, producing many (s,a) that could be updated. But not all of them are equally useful. The value of some states have changed a lot, and the value of their predecessors are also likely to change a lot. It is then natural to prioritize them according to a measure of their urgency, and perform them in order of priority.

This is **prioritized sweeping**. A queue is maintained for every (s,a) whose value would change if updated, prioritized by the size of the change.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow policy(S,Q)$
(c) Take action A; observe resultant reward, R, and state, S'
(d) $Model(S,A) \leftarrow R, S'$
(e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
(f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
(g) Loop repeat n times, while $PQueue$ is not empty:
$S,A \leftarrow first(PQueue)$
$R,S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
Loop for all $\bar{S},\bar{A}$ predicted to lead to S:
$\bar{R} \leftarrow$ predicted reward for $\bar{S},\bar{A},S$
$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
if $P > \theta$ then insert $\bar{S},\bar{A}$ into $PQueue$ with priority P

(a)

Figure 21: Prioritized sweeping algorithm (deterministic environment)

8.4.1 What about stochastic environment

The model is maintained by keeping counts of the number of times each (s,a) has been experienced and what their next states were. We can update each (s,a) not with a sample update but with an **expected update**, taking into account the possibilities of the next states and their probability of occurring.

Using expected updates can waste a lot of computation on low-probability transitions. Often using samples help converge faster despite the added variance. Sample updates can be better because they break the overall backing-up computation into smaller pieces (individual transitions) which enables them to be more focused on the pieces that would have the greater impact.

We have focused on backward focusing, but this is only one strategy. Another could be to focus on states, according to how easy they can be reached from the states that are visited frequently under the current policy, which is **forward focusing**.

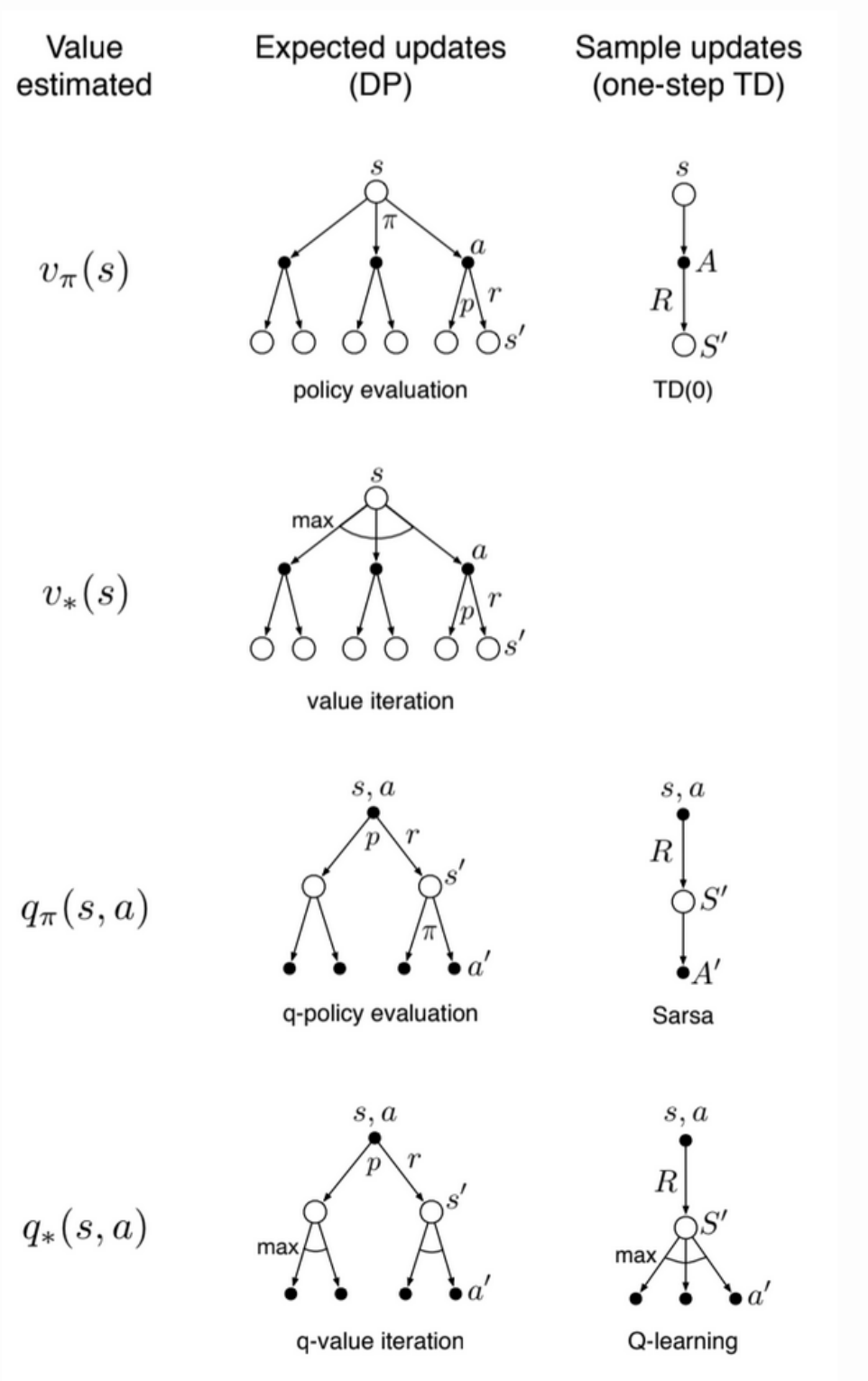
8.5 Expected vs. Sample update

We have considered many value-function updates. If we focus on one-step upates, they vary along 3 dimensions:

- Update state value or action values
- Estimate the value of the optimal policy or any given policy
- The updates are expected updates (consider all possible events) or sample updates (consider a sample of what might happen)

These 3 binary dimensions give rise to 8 cases, seven of which are specific algorithms. Any of these one-step method can be used in planning methods. The Dyna-Q use q_* sample update, but it can also use q_* expected update or q_π update.

When we don't have the environment model, we can't do an expected model so we use a sample. But are expected update better than sample updates when



(a)

Figure 22: Backup diagrams for one-step updates

available?

(1) they yield a better estimate because they are uncorrupted by sampling error;

(2) but they require more computation

8.5.1 computational requirements

For q_* approximation in this setup

- discrete states and actions
- table-lookup representation of the approximate value function Q
- a model in the form of estimated dynamics $\hat{p}(s', r|s, a)$

The expected update is:

$$Q(s, a) \leftarrow \sum_{s',r} \hat{p}(s', r|s, a)\big[r + \gamma \max_{a'} Q(s', a')\big]$$

The sample update is (Q-learning like):

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big[R + \gamma \max_{a'} Q(S', a') - Q(s, a)\big]$$

expected update vs.sample update: difference depends on stochasticity of the environment.

Many possible next states = many computation for the expected update.

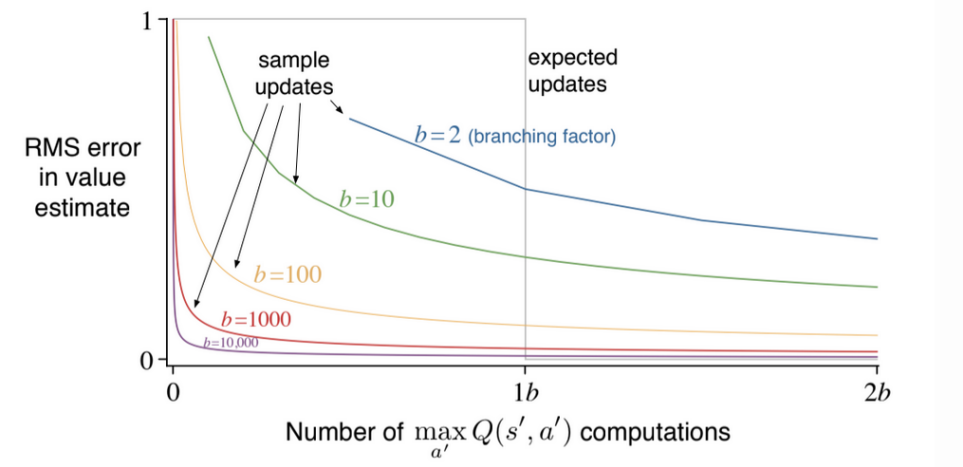
For a particular set (s, a), let b be the branching factor (the number of possible next states). Then an expected update of this pair requires roughly b times as much computation as a sample update.

conclusion: sample updates are better to expected updates on problems with a large branching factor and too many states

8.6 Trajectory Sampling

2 ways of distributing updates:

- **exhaustive sweep** (classical approach from DP): perform sweeps through the entire state space, updating each state once per sweep
- sample from the state space according to some distribution



(a)

Figure 23: Comparison of the efficiency of expected and sample updates

- uniform sample: Dyna-Q
- on-policy distribution (interact with the model following the current policy). Sample state transitions and rewards are given by the model, and sample actions are given by the current policy. We simulate explicit individual trajectories and perform update at the state encountered along the way -> **trajectory sampling**

Is the on-policy distribution a good one?

- it causes vast, uninteresting parts of the space to be ignored
- could be bad because we update the same parts over and over

conclusion:

- in the long run (and/or small problems), exhaustive sampling
  - Focusing on the on-policy distribution may hurt in the long run because the commonly occurring states all already have their correct value. Sampling them is useless.
- large problems, on-policy sampling/distribution

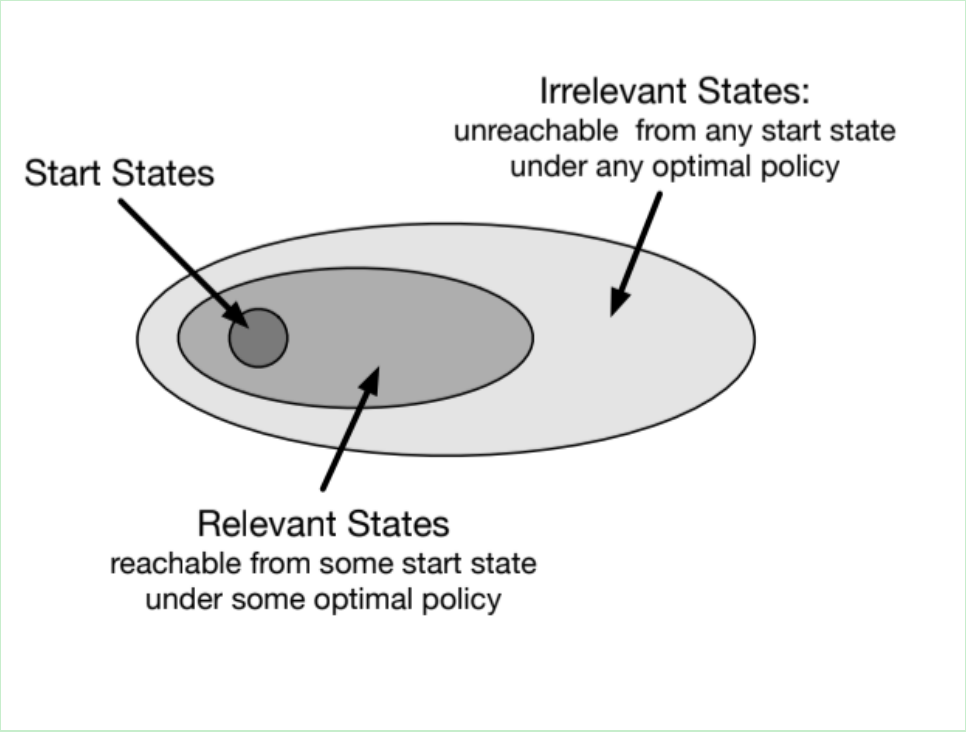
8.6.1 Real-Time Dynamic Programming (RTDP)

RTDP is an on-policy trajectory-sampling version of the value-iteration algorithm of DP. It is closely related to full-sweep policy iteration so we illustrate some advantages that on-policy trajectory sampling can have.

RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates.

RTDP is an example of an asynchronous DP algorithm. Asynchronous algorithms do not do full sweeps, they update state values in any order, using whatever value of other states happen to be available. In RTDP, th update order is dictated by the order states are visited in real or simulated trajectories.

8.6.2 Evaluation



(a)

Figure 24: Classification of states

- trajectories start from a designated set of start states
- we have a prediction problem for a given policy

- on-policy trajectory sampling allows to focus on useful states

8.6.3 Improvement

For Control, the goal is to find an optimal policy instead of evaluating a given policy, there might be states that cannot be reached by any of the optimal policies from any of the start states, so there is no need to specify optimal actions for those irrelevant states. We in fact only need an optimal partial policy.

Finding such an optimal partial policy can require visiting all state-action pairs, even those turning out to be irrelevant in the end. It is generally not possible to stop updating any state if convergence to an optimal policy is important.

For certain types of problems under certain conditions, RTDP is guaranteed to fin a policy that is is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all.

The task must be an undiscounted episodic task for MDP with absorbing goal state that generate zero reward.

At every step or a real or simulated trajectory, RTDP selects a greedy action and applies the expected value-iteration update operation to the current state. It can also update other states, like those visited in a limited-horizon look-ahead search from the current state.

More conditions for guarantee of optimal convergence:

- the initial value of every goal state is 0
- there exists at least one policy that guarantees a goal state will be reached from any start state
- all rewards from transitions from non-goal states are strictly negative
- the initial values are $\geq$ to optimal values

Proof: Barto, Bradtke and Singh 1995

Tasks having these properties are examples of stochastic optimal path problems, which are usually stated in term of cost minimization instead of reward

maximization, but we can just maximize the negative returns.

8.7 Planning at Decision Time

Two ways to plan:

- background planning (Dyna, DP)
- decision-time planning: begin and complete planning *after* encountering each state S_t, to produce a single action A_t.

8.8 Heuristic search

The classical state-space plannng methods are decision-time planning methods collectively known as heuristic search. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root, then the best is chosen as the current action, and all other backed-up values are discarded.

8.9 Rollout algorithms

skip

8.10 MCTS

skip

8.11 Learning Objectives (UA RL MOOC)

Lesson 1: What is a model?

1. Describe what a model is and how they can be used

- Model are used to store knowledge about the transition and reward dynamics
- Given S, A into model, model outputs R, S'
- A model allows for planning
- Planning refers to the process of using a model to improve a policy (use model to simulate experience, update value functions with the simulated experience, then improve policy with the updated value functions)

2. Classify models as distribution models or sample models

- Sample models procedurally generate samples, without explicitly storing the probability of each outcome (e.g. flip coin twice: HT)
- Distribution models contain a list of all outcomes and their probabilities (e.g. flip coin twice HT(1/4),HH(1/4),TT(1/4),TH(1/4))

3. Identify when to use a distribution model or sample model

Sample model can be computationally inexpensive. Distribution model contains more info, but it's hard to specify and can become large.

4. Describe the advantages and disadvantages of sample models and distribution models

Sample models require less memory

Distribution models can be used to compute the exact expected outcome (note sample models have to averaging many samples to get an approximate). Can be used to access risk.

5. Explain why sample models can be represented more compactly than distribution models

Consider rolling dice sample. The more dice there are, the larger the state space.

Lesson 2: Planning

6. Explain how planning is used to improve policies

Planning uses simulated experience from model to improve policies.

7. Describe random-sample one-step tabular Q-planning

(1) sample from model; (2) Q-learning update; (3) Greedy policy improvement

Lesson 3: Dyna as a formalism for planning

8. Recognize that direct RL updates use experience from the environment to improve a policy or value function

Direct RL, like Q-learning, directly learn from real world experience (environment)

9. Recognize that planning updates use experience from a model to improve a policy or value function

Indirect RL, like Q-planning, learn from simulated experience (generated by model) to improve value function and policy.

10. Describe how both direct RL and planning updates can be combined through the Dyna architecture

see 20 we do direct RL with real experience to improve policy or value functions; and we learn a model use real experience, then we sample (state, action) from the model which we use to get simulating experience, and we do planning update based on simulated experience to update policy or value functions

11. Describe the Tabular Dyna-Q algorithm

see Chap 8.2

12. Identify the direct-RL and planning updates in Tabular

direct RL update is the step (d), planning update is step (f)

13. Identify the model learning and search control components of Tabular Dyna-Q

10-13 see chap 8.2

14. Describe how learning from both direct and simulated experience impacts performance

It accelerate learning and it also very sample efficient.

15. Describe how simulated experience can be useful when the model is accurate

It can provide more samples for training, thus it's improve sample efficiency.

Lesson 4: Dealing with inaccurate models

16. Identify ways in which models can be inaccurate

Models are inaccurate when transitions they store are different from transitions that happen in the environment (like haven't explore enough or the environment is nonstationary / changing).

17. Explain the effects of planning with an inaccurate model

At first, the model is incomplete. As the agent interacts with the environment, the model stores more and more transitions. Then, the agent can perform updates by simulating transitions it's seen before. That means that as long as the agent has seen some transitions, it can plan with the model.

If the agent plan to with inaccurate model, then the value function or policy that the agent updates might change in the wrong direction. Planning with inaccurate model can make the policy worse w.r.t the environment.

18. Describe how Dyna can plan successfully with a partially inaccurate model

In the step (f), the model only knows the next state and reward from (s, a) it has already visited; therefore, Dyna-Q can only do planning updates from previously visited (s, a) pairs. Dyna-Q only plans the transition it has already seen. So in the first few timesteps of learning, Dyna-Q might do quite a few planning updates with the same transition. However, as Dyna-Q visits more (s, a) in the environment, its planning updates become more evenly distributed throughout the state action space.

19. Explain how model inaccuracies produce another exploration-exploitation trade-off

Explore to make sure its model is accurate, exploit the model to compute the optimal policy, assuming that the model is correct.

20. Describe how Dyna-Q+ proposes a way to address this trade-off

We add a bonus rewards for exploration, new reward = $r + \kappa\sqrt{\tau}$, where κ is small constant and τ is timesteps since transition was last tried.

21. (new research area?) We can have a separate thread (multi-threading) for planning specifically.

22. UCB method is similar to the bonus reward of Dyna-Q+: the difference is that they use differently (UCB can't work with nonstationary problems)

23. (new research area?) We start an easy and safe environment for the agent then we gradually make the environment more realistic/dangerous to the agent.

Part II: Approximate Solution Methods In the second part, we are dealing with arbitrarily large state spaces; in these cases, it is hard to find π_* or v_*, q_*. Our goal instead is to find a good approximate solution using limited computational resources.

The problem with large state spaces is threefold: memory, time and data. In many of our target tasks, almost every state encountered will never have been seen before. The solution is *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

To some extent, we need only combine RL methods with existing generalization methods. One of frequent use is *function approximation*, which takes examples from a desired function (e.g. a value function) and attempts to generalize from them to construct an approximation of the entire function.

Function approximation is an instance of supervised learning.

RL with function approximation involves a number of new issues (usually not in SL), such as nonstationarity, bootstrapping, and delayed targets.

Chap 9 (prediction): on-policy training, policy is given and only its value function is approximated

Chap 10 (control): an approximation to the optimal policy is found

Chap 11: off-policy learning with function approximation

Chap 12: *eligibility traces*, which dramatically improves the computational properties of multi-step RL methods in many cases.

Chap 13 (control): *policy-gradient*, a different approach to control, which approximates the optimal policy directly and need never form an approximate value function.

9 On-policy Prediction with Approximation

We start consider using function approximation in estimating the state-value function from on-policy data, which approximates v_π from experience generated using a known policy π.

The approximate value function is represented as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$, instead of a table. We write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$.

For example,

- $\hat{v}$ = linear function in features of the state, $\mathbf{w}$ = vector of feature weights
- $\hat{v}$ = function computed by a multi-layer ANN, $\mathbf{w}$ = vector of connection weights in all the layers
- $\hat{v}$ = function computed by a decision tree, $\mathbf{w}$ = all the numbers defining the split points and leaf values of the tree

Typically, the number of weights is much less than the number of states ($d << |S|$), and one weight takes care of many states.

Extending RL to function approximation also makes it applicable to partially observable problems (POMDP) (in which the full state is not available to the agent).

All theoretical results for methods using function approximation apply equally well to cases of partial observability.

Function approximation cannot augment the state representation with memories of past observations.

9.1 Value-function Approximation

Definition 9.1 ***individual update notation****: $s \mapsto u$ (where s is the state updated and u is the update target. The estimated value for s should be more like u.*

- MC: $S_t \mapsto G_t$
- TD(0): $S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$
- n-step TD: $S_t \mapsto G_{t:t+n}$
- DP: $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) | S_t = s]$

We want to extend a one-to-one mapping to one-to-many mapping, i.e., updating s will also update many other states.

Supervised learning methods learn to mimic input-output examples, and when the outputs are numbers, like u, the process is often called *function approximation*. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of each update as a training example.

In principle, we can use any method for supervised learning from examples, including ANN, decision trees, and various kinds of multivariate regression. However, in RL, we require the methods are able to handle nonstationarity cases as RL agents need to do online learning (interact with the environment to obtain data).

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

Assumptions for tabular setting:

- no need to specify an explicit objective for the prediction because the learned value will approx the true value
- an update at one state does not affect any other state

Now the assumptions do not hold. Making one state's estimate more accurate invariably(always) means making others' less accurate. We need to find which states we care most about. Therefore, we introduced the state distribution $\mu(s)$:

Definition 9.2 ***state distribution*** $\mu(s)$*:* $\mu \geq 0$, $\sum_s \mu(s) = 1$, *representing how much we care about the error (square distance between approx value* $\hat{v}(s, \mathbf{w})$ *and true value* $v_\pi(s)$*) in each state* s*.*

Since we are doing on-policy policy evaluation, the μ is from policy π. We can

also design μ, then it becomes off-policy prediction.

Definition 9.3 ***Mean Squared Value Error*** *($\overline{VE}$): a natural objective function,*

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in S} \underbrace{\mu(s)}_{weight} \underbrace{[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2}_{error}$$

Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training, this is called *on-policy distribution*. In continuing tasks, the on-policy distribution is the stationary distribution under π. The book talks about the episodic tasks (not including here).

!! NOTE that the RL community is not sure if $\overline{VE}$ is the right performance objective for RL. But this is the best we can find so far. Adam White: VE might not be the best because we are only approximating the exact true values. Instead, we can use Average reward + Policy Gradient methods for control.

9.2.1 on-policy distribution

In continuting tasks, the on-policy distribution is the stationary distribution under π.

In episodic tasks, it depends on how the initial states of episodes are chosen.

- $h(s)$ = prob. that an episode begins in each state s
- $\eta(s)$ = average num. of time steps spent in state s in single episode

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

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

9.3 Stochastic-gradient and Semi-gradient Methods

In GD, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, ..., w_d)^T$, and the approximate value function $\hat{v}(s, \mathbf{w})$

is a differentiable function of $\mathbf{w}$ for all $s \in S$. However, no $\mathbf{w}$ can gets all the states in real world.

GD vs SGD In both GD and SGD, we update a set of parameters in an iterative manner to minimize an error function.

- GD: we run through ALL the samples to do a single update for a parameter in a particular iteration (which is not feasible)
- SGD: we run through ONLY ONE or a SUBSET of samples to do the update. If we use a SUBSET, it is called Minibatch Stochastic Gradient Descent.

The solution to $\mathbf{w}$ is SGD: we minimize error on the *observed samples*. The weight vector of SGD is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w} - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2$$

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

Now we consider when we do not have the exact true value $v_\pi(S_t)$, but an approximate value U_t. Then

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

If U_t is an *unbiased* estimate ($\mathbb{E}[U_t|S_t = s] = v_\pi(s)$), then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (chap2). For example, Monte Carlo target $U_t \doteq G_t$ is an unbiased estimate of $v_\pi(S_t)$.

convergence to a local optimum is guaranteed depending on α.

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

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$

Parameter: $\alpha > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, S_1, A_1, ..., R_T, S_T$ using π

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

Take action A; observe resultant reward, R, and state, S'

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

On the other hand, Bootstrapping targets, such as n-step returns $G_{t:t+n}$ or the DP target, all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased. These are not the true gradient-descent methods, but $semi$-gradient methods.

Although semi-gradient (bootstrapping) methods do not converge as robustly as true gradient methods, they offer important advantages: (1) enable much faster learning; (2) enable learning to be continual and online; (3) provides computational advantages. A typical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_t, \mathbf{w})$

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

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$ where $\hat{v}(terminal, \dot{}) = 0$

Parameter: $\alpha > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

Loop for each episode:

  Init S

  Loop for each step of episode:

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

    Take action A; observe reward, R, and state, S'

    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{v}(S_t, \mathbf{w}) - \hat{v}(S, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$

    $S \leftarrow S'$

  until S is terminal

The target U_t in semi-gradient methods is biased, so the $\mathbf{w}$ may not converge to a local optimum.

state aggregation a simple form of generalizing function approximation in which states are grouped together, with one estimated value for each group.

9.4 Linear Methods

One of the most important special cases of function approximation is that, $\hat{v}(\dot{;}\mathbf{w})$ is a linear function of the weight vector $\mathbf{w}$. Corresponding to every state s, there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), ..., x_d(s))^T$, with the same number of components as $\mathbf{w}$.

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

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

The vector $\mathbf{x}(s)$ is a *feature vector* representing state s. Each component $x_i(s)$ is a *feature*.

The gradient of the approximation value function w.r.t $\mathbf{w}$ is then $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$. Thus, we can simplify the general SGD update to

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(s)$$

Understand the fixed point of linear TD learning Let us start with TD update with linear function approximation on the first line, recall the value of a state $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s)$

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t) \\
&= \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}_t \quad \text{simplify notation} \\
&= \mathbf{w} + \alpha[R_{t+1} + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t]\mathbf{x}_t \quad \text{due to } \hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \\
&= \mathbf{w} + \alpha[\underbrace{R_{t+1}\mathbf{x}_t}_{\boldsymbol{b}} - \underbrace{\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T}_{\boldsymbol{A}} \mathbf{w}_t] \quad \text{transpose doesn't change scalar}
\end{aligned}$$

The TD update can be rewritten as the expected update + a noise term, so it is largely dominated by the behavior of the expected update.

$$\begin{aligned}
\mathbb{E}[\Delta \mathbf{w}_t] &= \alpha(\boldsymbol{b} - \boldsymbol{A}\mathbf{w}_t) \\
\boldsymbol{b} &= \mathbb{E}[R_{t+1}\mathbf{x}_t] \\
\boldsymbol{A} &= \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \quad \text{expct. over features}
\end{aligned}$$

Definition 9.4 ***TD Fixed Point*** *$\mathbf{w}_{TD}$, the weight is converged when the expected TD update = 0. TD fixed point is the solution to the semi-gradient TD(0).*

$$\mathbb{E}[\Delta \mathbf{w}_{TD}] = \alpha(\boldsymbol{b} - \boldsymbol{A}\mathbf{w}_{TD}) = 0$$

When A is invertible, we can rewrite $\mathbf{w}_{TD} = \boldsymbol{A}^{-1}\boldsymbol{b}$. We say $\mathbf{w}_{TD}$ is a solution to this linear system; the solution is called TD fixed point.

$\mathbf{w}_{TD}$ minimizes an objective that is based on this $\boldsymbol{A}$ and $\boldsymbol{b}$. This objective extends the connection between TD and Bellman equations to the function approximation setting.

Recall in the tabular setting, TD is described as a sample based method for solving the Bellman equation. Linear TD similarly approximates the solution to the Bellman equation, minimizing what is called the projected Bellman error.

Describe a theoretical guarantee on the mean squared value error at the TD fixed point We want to understand the relationship between the solution found by TD and the minimum value error solution.

$$\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} min_{\mathbf{w}} \overline{VE}(\mathbf{w})$$

The difference between these two errors depends on γ and the quality of features.

Why isn't the TD fixed point error equal to the minimum value error solution? This is because bootstrapping under function approximation. If our function approximator is good, then our estimate of the next state will be accurate.

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

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$ where $\hat{v}(terminal, \cdot) = 0$

Parameter: $\alpha > 0$, $n > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

All store and access operations (S_t and R_t) can their index mod $n + 1$

Loop for each episode:

Init and store $S_0 \neq$ terminal

$T \leftarrow \inf$

Loop for t = 0,1,2,...:

If $t < T$,then:

Take an action according to $\pi(\cdot|S_t)$

Observe and store R_{t+1} and S_{t+1}

If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

If $\tau \geq 0$:

$G \leftarrow \sum_{i=\tau+1}^{min(\tau+n,T)} \gamma^{i-\tau-1} R_i$

If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G - \hat{v}(S_\tau, \mathbf{w})]\nabla\hat{v}(S_\tau, \mathbf{w})$

until $\tau = T - 1$

9.5 Feature Construction for Linear Methods

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

Linear methods,

- pro: guarantee convergence, data efficient, computation efficient
- con: does not consider interactions between features

9.5.1 Polynomials

9.5.2 Fourier Basis

9.5.3 Coarse Coding

The features used to construct value estimates, are one of the most important parts of a RL agent.

Recall linear value function approximation $v_\pi(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$. Recall that a tabular representation can be expressed as a binary (0 or 1) feature vector. Each state is associated with a different feature. Agent-presented state is 1, all other features are 0. The tabular case is a special case of linear function approximation, where the feature vector is an indicator/one-hot encoding of the state.

This becomes a problem when the feature space is large. Recall that we can use state aggregation to associate nearby states with the same features.

From state aggregation to coarse coding: although state aggregation can have arbitrary shapes, but it does not allow **overlapping**. We can remove this restriction, and we have coarse coding. Therefore, coarse coding is a generalization of state aggregation.

Definition 9.5 ***binary feature****: 1-0 valued feature, 1 for present, 0 for absent.*

Definition 9.6 ***coarse coding****: representing a state with binary features that overlap.*

Changing the shapes and sizes of features impacts generalization and discrimination, and so affects the speed of learning and the value functions we can represent.

Coarse coding parameters affect discrimination and generalization, which in turn affect learning accuracy.

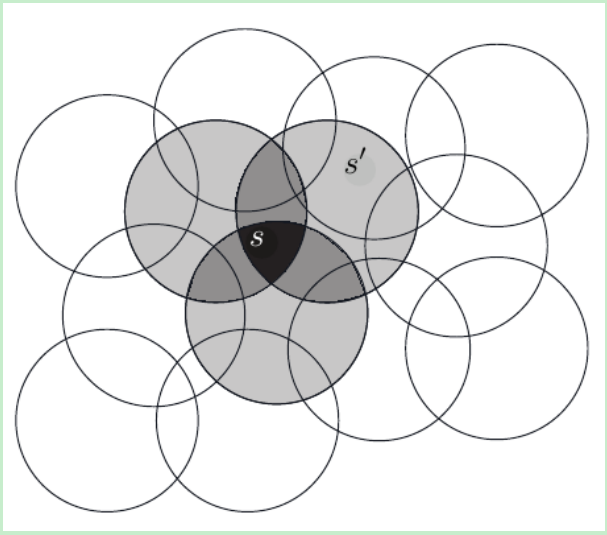


Figure 25: Coarse coding. Generalization from s to s' depend on the number of their features whose receptive fields overlap

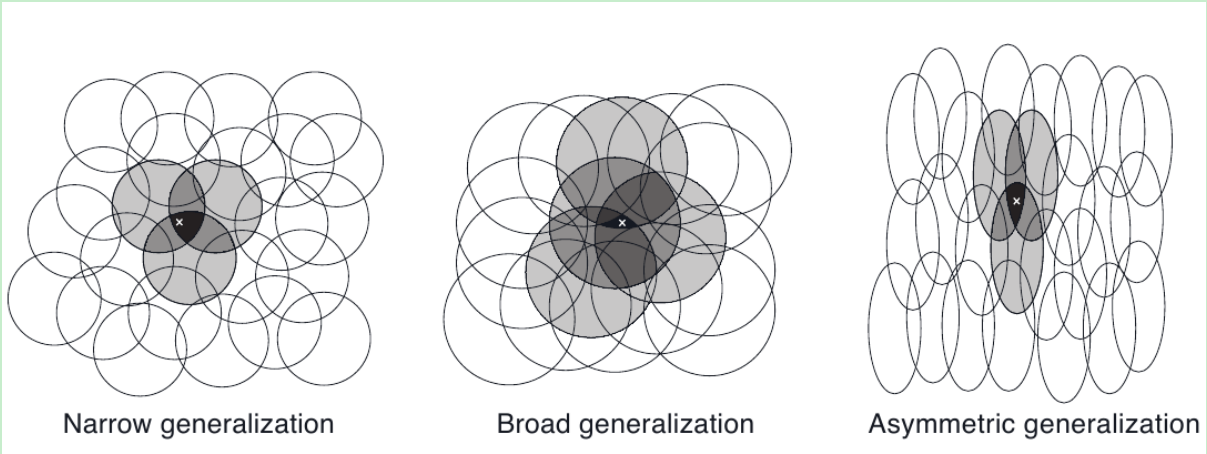


Figure 26: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

9.5.4 Tile Coding

Definition 9.7 ***Tile coding****, a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient.*

This may be the most practical feature representation for modern sequential digital computers.

In the coding, the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. The tiles of a partition do not overlap.

Recall one tiling is just like state aggregation. Tile coding = multiple overlapped state aggregation. To improve discrimination ability, we use multiple tilings so that we have many small intersections.

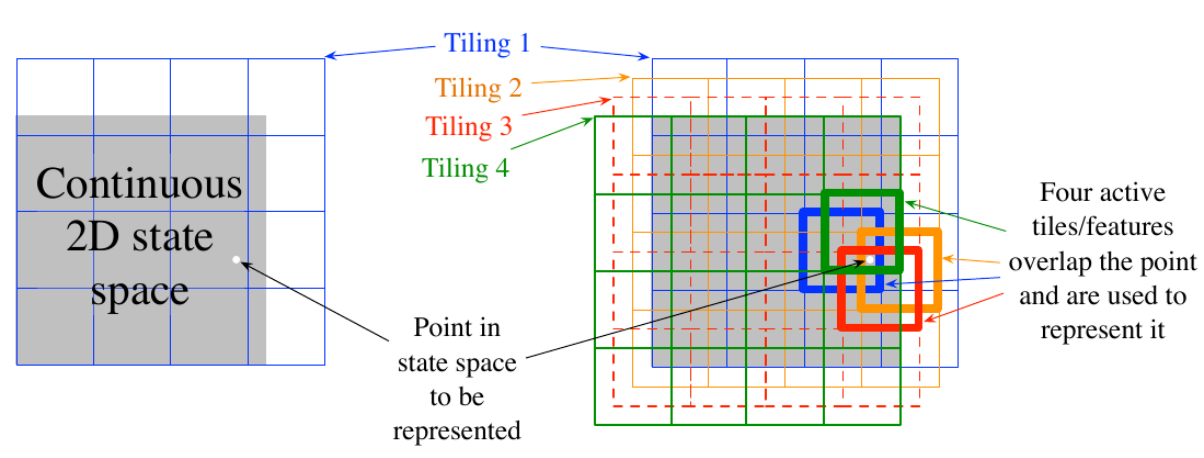


Figure 27: Multiple, overlapping grid-tilings on a limited 2D space. Thee tilings are offset from one another by a uniform amount in each dimension.

To get the strength of coarse coding, we need overlapping receptive fields. The advantages of tile coding

- the total active features at one time is the same for any state
- computational advantages from its use of binary feature vectors (just sum all active features)
- save memory

The choice of how to offset the tilings from each other affects generalization.

Tile asymmetrical offsets are preferred. If the tilings are uniformly offset, then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding. In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles.

Another useful trick of reducing memory requirements is *hashing* (Like one-to-many mapping).

9.6 Selecting Step-Size Parameters Manually

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

where τ is the number of experience, and $\mathbf{x}$ is the random feature vector.

See Ex 9.5

9.7 Nonlinear Function Approximation: Artificial Neural Networks

!! Note that neural network, like coarse coding/tile coding, is also a way of feature representation. State aggregation, coarse coding and tile coding are fixed representation. NN are flexible representation (due to learning)

This section is like a summary of the NN course.

ANN is a network of interconnected units that have some of the properties of neurons.

A feedforward ANN has no loops in its network, i.e., there are no paths within the network by which a unit's output can influence its input. If an ANN has at least one loop in its connections, it is a recurrent rather than a feedforward

ANN.

Hidden layers: layers that are neither input/output layers.

A real-valued weight is associated with each link.

The units in ANN are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function (activation function). **Nonlinearity is essential**: if all the units have linear activation functions, the entire network is equivalent to a network without hidden layers (linear+linear = linear).

Universal Approximation Property An ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy.

Training the hidden layers of an ANN is a way to auto create features appropriate for a given problem so that hierarchical representations can be produced w/o replying exclusively on hand-crafted features.

ANN typically learn by a stochastic gradient method. Each weight is adjusted in a direction aimed at improving the network's overall performance as measured by an objective function to be either min/maximized (that is, to estimate the partial derivative of an objective function w.r.t each weight, given the current values of all network's weights).

Backpropagation algorithm consists of alternating forward and backward passes through the network. (In later chapter, we discuss train ANN with RL instead of backpropagation).

Backpropagation may not work well for deep ANN. Because (1) the large number of weights can lead to overfitting; (2) the partial derivatives computed by its backward passes either decay rapidly (slow learning) or grow rapidly (unstable learning).

Overfitting is a problem for any function approximation method that adjusts

functions with many degrees of freedom on the basis of limited training data.

Methods for reducing overfitting / improve training

- cross validation: stop training when performance begins to decrease on validation data different from the training data
- regularization: modifying the objective function to discourage complexity of the approximation
- weight sharing: introducing dependencies among the weights to reduce the number of degrees of freedom
- dropout: during training, units are randomly removed from the network along with their connections
- deep belief networks: deepest layers are trained first using unsupervised learning
- batch normalization: normalizes the output of dep layers before they feed into the following layer
- deep residual learning: sometimes it is easier to learn a function differs from the identify function than to learn the function itself (by adding shortcut, or skip connections)
- deep convolution neural network

Compute the gradient of a single hidden layer neural network

- step 1: define a loss on the parameters of the NN
- step 2: find the parameters which minimize this loss function

Suppose we have input layer $\boldsymbol{s}$, hidden layer $\mathbf{x}$, output layer $\boldsymbol{y}$, input-hidden weight $\boldsymbol{A}$, hidden-output weight $\boldsymbol{B}$.

- $\mathbf{x} \doteq f_A(\boldsymbol{\psi})$, $\boldsymbol{\psi} \doteq \boldsymbol{s}\boldsymbol{A}$
- $\hat{\boldsymbol{y}} \doteq f_B(\boldsymbol{\theta})$, $\boldsymbol{\theta} = \mathbf{x}\boldsymbol{B}$

Gradient w.r.t $\boldsymbol{B}$:

$$\begin{aligned}\frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \boldsymbol{B}_{\boldsymbol{jk}}} &= \frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \hat{\boldsymbol{y}}_{\boldsymbol{k}}} \frac{\partial \hat{\boldsymbol{y}}_{\boldsymbol{k}}}{\partial \boldsymbol{B}_{\boldsymbol{jk}}} \\ &= \frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \hat{\boldsymbol{y}}_{\boldsymbol{k}}} \frac{\partial f_{\boldsymbol{B}}(\theta_k)}{\partial \theta_k} \frac{\partial \theta_k}{\partial \boldsymbol{B}_{jk}} \\ &= \underbrace{\frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \hat{\boldsymbol{y}}_{\boldsymbol{k}}} \frac{\partial f_{\boldsymbol{B}}(\theta_k)}{\partial \theta_k}}_{\delta_k^{\boldsymbol{B}}} \boldsymbol{x}_{\boldsymbol{j}}\end{aligned}$$

Note that the weight $\boldsymbol{A}$ also affect $\mathbf{x}$, so a chain rule set up is needed here. Gradient w.r.t $\boldsymbol{A}$:

$$\begin{aligned}\frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \boldsymbol{A}_{\boldsymbol{ij}}} &= \delta_k^{\boldsymbol{B}} \frac{\partial \theta_k}{\partial \boldsymbol{A}_{ij}} \\ &= \delta_k^{\boldsymbol{B}} \boldsymbol{B}_{jk} \frac{\partial \mathbf{x}_j}{\partial \boldsymbol{A}_{ij}} \\ &= \delta_k^{\boldsymbol{B}} \boldsymbol{B}_{jk} \frac{\partial f_{\boldsymbol{A}}(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial \boldsymbol{A}_{ij}} \\ &= \underbrace{\delta_k^{\boldsymbol{B}} \boldsymbol{B}_{jk} \frac{\partial f_{\boldsymbol{A}}(\psi_j)}{\partial \psi_j}}_{\delta_j^{\boldsymbol{A}}} \boldsymbol{s}_i\end{aligned}$$

$$\frac{\partial \theta_k}{\partial \boldsymbol{A}_{ij}} = \boldsymbol{B}_{jk} \frac{\partial \mathbf{x}_j}{\partial \boldsymbol{A}_{ij}}$$

$$\frac{\partial \mathbf{x}_j}{\partial \boldsymbol{A}_{ij}} = \frac{\partial f_{\boldsymbol{A}}(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial \boldsymbol{A}_{ij}}$$

Notice that both gradients can be rewritten in a similar form. They have a term δ that contains an error signal times their input.

$$\frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \boldsymbol{A}_{\boldsymbol{ij}}} = \delta_j^{\boldsymbol{A}} \boldsymbol{s}_i$$

$$\frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \boldsymbol{B}_{\boldsymbol{jk}}} = \delta_k^{\boldsymbol{B}} \boldsymbol{x}_{\boldsymbol{j}}$$

one-hidden layer backprop algorithm

for each i/o pair (s, y) in dataset:

$\delta_k^{\boldsymbol{B}} = \frac{\partial L(\hat{\boldsymbol{y}}_{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{k}})}{\partial \hat{\boldsymbol{y}}_{\boldsymbol{k}}} \frac{\partial f_{\boldsymbol{B}}(\theta_k)}{\partial \theta_k}$

$\nabla_{\boldsymbol{B}}^{jk} = \delta_k^{\boldsymbol{B}} \mathbf{x}_j$

$\boldsymbol{B} = \boldsymbol{B} - \alpha_{\boldsymbol{B}} \nabla_{\boldsymbol{B}}$

$\delta_j^{\boldsymbol{A}} = \delta_k^{\boldsymbol{B}} \boldsymbol{B}_{jk} \frac{\partial f_{\boldsymbol{A}}(\psi_j)}{\partial \psi_j}$

$\nabla_j^{\boldsymbol{A}} = \delta_j^{\boldsymbol{A}} \boldsymbol{s}_i$

$\boldsymbol{A} = \boldsymbol{A} - \alpha_{\boldsymbol{A}} \nabla_{\boldsymbol{A}}$

First get prediction $\hat{y}$ (forward pass), then compute the gradients backwards from the output. Specifically, we first compute $\delta_{\boldsymbol{B}}$ and the gradient for B, then we use this gradient to update the parameters $\boldsymbol{B}$, the step size $\alpha_{\boldsymbol{B}}$. Next we update the parameters $\boldsymbol{A}$, we compute $\delta_{\boldsymbol{A}}$ which uses $\delta_{\boldsymbol{B}}$. Notice that by computing the gradients backwards, we avoid re-computing the same terms; In fact, this is the main idea behind backpropgation: it is simple GD with efficient strategy to compute gradients.

9.8 Least-Squares TD

-> Direct computation of the TD fixed point, instead of the iterative method.

Recall that TD(0) with linear FA converges asymptotically to the TD fixed point:

$$\mathbf{x}_{TD} \doteq \mathbf{A}^{-1}\mathbf{b}$$

- $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$
- $\mathbf{A} \doteq \mathbb{E}\left[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\right]$

The Least Squares TD Algorithm (LSTD) computes directly estimates for $\mathbf{A}$ and $\mathbf{b}$:

- $\hat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{t+1}\mathbf{x}_k$
- $\hat{\mathbf{A}}_t \doteq \sum_{k=0} t - 1\mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^T + \varepsilon\mathbf{I}$

where:

- $\mathbf{I}$ is the identity matrix
- $\varepsilon\mathbf{I}$ ensures that $\hat{\mathbf{A}}$ is always invertible

It seems that these estimates should be divided $t-1$ and they should, but we don't care because when we use them we effectively divide one by another.

LSTD estimates the fixed point as:

$$\mathbf{w}_t \doteq \hat{\mathbf{A}}_t^{-1}\hat{\mathbf{b}}_t$$

Complexity

- The computation involves the inverse of A, so complexity $O(d^3)$
- Fortunately our matrix has a special form of the sum of outer products, so the inverse can be computed incrementally with only $O(d^2)$. This is using the Sherman-Morrisson formula
- Still less efficient than the $O(d)$ of incremental version, but can be handy depending on how large d is.

9.9 Memory-Based Function Approximation

- parametric approach (prev. discussed): training examples is (1) used to update the parameters; (2) then discarded
- memory-based approach: stores training example w/o updating the parameters. If a value update is needed, some examples is taken from the storage, and then discarded.

Some examples: nearest neighbor, weighted average, locally weighted regression

properties of memory-based methods

- no limit of functional form
- more data = more accuracy
- experience effect more immediate to neighboring states
- retrieving neighboring states can be long, if there are many data in memory

9.10 Kernel based Function Approximation

kernel = kernel function = $k(s, s')$, measures the strength of the generalization between s and s'.

kernel regression = memory-based method, that computes a kernel weighted average of all examples in storage

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s') g(s')$$

where $\mathcal{D}$ is example storage, $g(s')$ is target value for a stored state s'.

Common kernel:

- Gaussian radial basis function
- in practice, $k(s, s') = \mathbf{x}(s)^T \mathbf{x}(s')$

9.11 Looking deeper at on-policy learning: Interest and Emphasis

problem: we treat all states equally important. we are interests in some states than others, how to distinguish them?

The on-policy distribution is defined as the distribution of states encountered in the MDP while following the target policy. Generalization: Rather than having one policy for the MDP, we will have many. All of them are a distribution of states encountered in trajectories using the target policy, but they vary in how the trajectories are initiated.

Interest (I_t) = how much we are interested in evaluating s at time t.

Emphasis (M_t) = non-negative scalar multiplies the learning update and thus emphasizes or de-emphaizes the learning done at time t.

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

(TODO: this feels like a concept from the Optimization course, interest and momentum)

9.12 Learning Objectives (UA RL MOOC)

Lesson 1: Estimating Value Functions as Supervised Learning

1. Understand how we can use parameterized functions to approximate value functions

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

2. Explain the meaning of linear value function approximation

$\hat{v}(s, \mathbf{w}) = \sum w_i x_i(s)$

3. Recognize that the tabular case is a special case of linear value function approximation.

Consider each state has a corresponding feature (one-hot encoding)

4. Understand that there are many ways to parameterize an approximate value function

ANN, decision tree etc

5. Understand what is meant by generalization and discrimination

generalization: updates to the value estimate of one state influence the value of other states

discrimination:the ability to make the values for two states different to distinguish between the values for these two state.

6. Understand how generalization can be beneficial

Generalization can speed up learning.

7. Explain why we want both generalization and discrimination from our function approximation

Because we want generalization to speed up learning, and discrimination to distinguish really different states.

8. Understand how value estimation can be framed as a supervised learning problem

Supervised learning involves approximating a function given a dataset of (input, target) pairs. For Monte-Carlo (S_t, G_t). For TD $(S_t, R_t + \gamma\hat{v}(S_t, \mathbf{w}))$

9. Recognize not all function approximation methods are well suited for reinforcement learning

Some are not suitable because they are designed for a fixed batch of data (offline learning, contrast to RL's online learning), or not designed for temporally correlated data (data in RL is always correlated).

Lesson 2: The Objective for On-policy Prediction

10. Understand the mean-squared value error objective for policy evaluation

We cannot guarantee perfect approximation for every state's value, so we need to define an objective, a measure of the distance between our approximation and the true values.

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \S} \underbrace{\mu(s)}_{\text{weight}} \underbrace{[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2}_{\text{error}}$$

11. Explain the role of the state distribution in the objective

$\mu(s)$ specifies how much we care for each state.

- It is a probability distribution
- It has higher values for states that are visited more often
- It serves as a weighting to minimize the error more in states that we care about

12. Understand the idea behind gradient descent and stochastic gradient descent

We use SGD to optimize the objective $\overline{VE}$. (1) the gradient indicates the direction to increase/decrease the weight vectors; (2) the gradient gives the direction of steepest ascent

13. Outline the gradient Monte Carlo algorithm for value estimation

see chap 9.3

14. Understand how state aggregation can be used to approximate the value function

Group multiple states into one grouped state. One feature for each grouped state.

15. Apply Gradient Monte-Carlo with state aggregation

see chap 9.3

Lesson 3: The Objective for TD

16. Understand the TD-update for function approximation

17. Highlight the advantages of TD compared to Monte-Carlo

TD enables faster learning, continual and online learning; provides computational advantages

18. Outline the Semi-gradient TD(0) algorithm for value estimation

see chap 9.3; semi-gradient TD as an approximation to SGD.

19. Understand that TD converges to a biased value estimate

The TD target depends on our estimate of the value in the next state. This means our update could be biased because the estimate in our target may be inaccurate.

20. Understand that TD converges much faster than Gradient Monte Carlo

TD can learn during the episode and has a lower variance update.

Lesson 4: Linear TD

21. Derive the TD-update with linear function approximation

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(s)$$

22. Understand that tabular TD(0) is a special case of linear semi-gradient TD(0)

linear TD is a strict generalization on both tabular TD and TD with state aggregation. Recall the feature vector is like one-hot encoded vector, where the i-th=1, else 0 corresponds to i-th weight.

23. Highlight the advantages of linear value function approximation over non-linear

- linear methods are simpler to understand and analyze mathematically
- with good features, linear methods can learn quickly and achieve good prediction accuracy

linear semi-gradient TD (aka TD with linear function approximation)

Lesson 1: Feature Construction for Linear Methods

24. Describe the difference between coarse coding and tabular representations

Tabular states can be represented with a binary one-hot encoding. Tabular representations become infeasible when state space is large (a state associate with a feature). We then introduced state aggregation to resolve this, where one feature represents many states. Coarse coding is a further generalization of state aggregation, which allows overlapping of features (receptive fields).

25. Explain the trade-off when designing representations between discrimination and generalization

The size, shape, and number of features (receptive fields) affect the generalization and discrimination, thus affecting learning accuracy. For example, larger circle can generalize better, but discrimination is worse.

26. Understand how different coarse coding schemes affect the functions that can be represented

Recall the step-function approximation in the textbook.

27. Explain how tile coding is a (computationally?) convenient case of coarse coding

Since grid is uniform, it's easy to compute which cell the current state is in.

28. Describe how designing the tilings affects the resultant representation

The symmetry/asymmetry offsets affect tile coding's generalization and discrimination.

29. ...

Lesson 2: Neural Networks

30. Define a neural network

- NN = input layer + hidden layer(s) + output layer
- layer = node(s); layers are linked, each link is a real-valued weight
- node contains non-linear function (active function)

31. Define activation functions

activation function = non-linear function (Sigmoid, ReLU, step function)

32. Define a feed-forward architecture

A network without link from output to input.

33. Understand how neural networks are doing feature construction

Tile coding has fixed parameters before learning, NN has both fixed and learned parameters. So NN can use both prior knowledge and knowledge from data to do feature construction.

34. Understand how neural networks are a non-linear function of state

Non-linear activation function resulting in a non-linear function of the inputs.

35. Understand how deep networks are a composition of layers

We can have many hidden layers (1) to allow composition of features. Composition can produce more specialized features by combining modular components; (2) to obtain abstractions, DNN compose many layers of lower-level of abstractions with each successive layer contributing to increasingly abstract

representation.

36. Understand the trade-off between learning capacity and challenges presented by deeper networks

Challenges mostly from overfitting and training efficiency.

Lesson 3: Training Neural Networks

37. Understand the importance of initialization for neural networks

If we choose a bad init parameters, we either have zero gradient or stuck in local optima.

38. Describe strategies for initializing neural networks

Method I: randomly sample the initial weights from a normal distribution with small variance, and normalize the weights.

$$\mathbf{w}_{init} \sim \frac{N(0,1)}{\sqrt{n_{inputs}}}$$

This way, each neuron has a different output from other neurons within its layer. This provides a more diverse set of potential features. By keeping the variants small, we ensure that the output of each neuron is within the same range as its neighbors. By normalizing the weights, we prevent larger variance as number of neurons grows.

Adam Algorithm are Method II + Method III.

Method II: Update Momentum M

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t) + \lambda M_t$$
$$M_{t+1} \leftarrow \lambda M_t - \alpha \nabla_{\mathbf{w}} L$$

The momentum term summarizes the history of the gradients using a decaying sum of gradients with decay rate λ. If recent updates in the same direction, then we have large momentum, making larger step; if recent updates in the opposite direction, then we kill the momentum.

Method III: vector step sizes

Instead of global step size, this uses different step sizes for different weights.

39. Describe optimization techniques for training neural networks

cross validation, regularization, dropout, residual learning etc

10 On-policy Control with Approximation

In this chapter we move from prediction problem to control problem, now with parametric approximation of the action-value $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$.

We learn the semi-gradient SARSA algorithm as the natural extension of semi-gradient TD(0) to action values and to on-policy control.

Once we have genuine function approximation we have to give up discounting and switch to a new 'average-reward' formulation of the control problem, with new 'differential' (just difference) value functions.

10.1 Episodic Semi-gradient Control

From chap 9 $S_t \mapsto U_t$ to chap 10 $S_t, A_t \mapsto U_t$

The general GD update for action-value prediction is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t)$$

Episodic semi-gradient one-step SARSA:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_t + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t)$$

10.2 Semi-gradient n-step SARSA

10.3 Average Reward: A New Problem Setting for Continuing Tasks

Episodic, discounted, average reward (New!)

why average reward In discounted setting, the only way to ensure the agents actions maximize reward over time is to keep increasing discount factor γ towards 1 (see the LeftRight8ring example). The γ can't be 1; otherwise, the

return will be infinite. Moreover, larger γ can result in larger and more variables sums (longer time horizon $\rightarrow$ larger discount factor $\rightarrow$ wider the range of return $\rightarrow$ larger variance), which might be hard to learn. So we introduce a new setting: average reward.

Definition 10.1 ***average reward*** $r(\pi)$, *no discounting considered, the agent cares the same for delayed rewards and immediate rewards. It defines the quality of a policy* π.

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

Note that the expectation conditioned on initial state S_0 and all actions according to π. The 2nd and 3rd equations hold if the MDP is ***ergodic***, that is, the *steady-state distribution* $\mu_\pi(s) \doteq \lim_{t\to\text{inf}} P\{S_t = s | A_{0:t-1} \sim \pi\}$ exists and is independent of S_0. In other words, in the long run, the expectation f being in a state depends only on the policy and the MDP transition probabilities (the effect of early decisions is temporary).

The steady state distribution μ_π is the special distribution under which you remain in the same distribution if you select actions according to π.

We consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Definition 10.2 ***differential return***, *diff return.*

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

- The differential return can only be used to compare actions if the same policy is followed on subsequent timesteps.
- we use average reward to compare policies.
- Only true $r(\pi)$ results in convergence

Bellman equations for differential value functions:

$$
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)[r - r(\pi) + v_\pi(s')] \\
q_\pi(s, a) &= \sum_{r,s'} p(s', r|s, a)[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a')] \\
v_*(s) &= max_a \sum r, s' p(s', r|s, a)[r - max_\pi r(\pi) + v_*(s')] \\
q_*(s, a) &= \sum_{r,s'} p(s', r|s, a)[r - max_\pi r(\pi) + max_{a'} q_*(s', a')]
\end{aligned}
$$

10.4 Deprecating the Discounted Setting

- the discounted setting is useful for the tabular case where the return for each state can be separated and averaged,
- and in continuing tasks, we could measure the discounted return at each time step;
- however, the average of the discounted rewards is proportional to the average reward, so the ordering of the policies don't change, making discounting useless

The root cause of the difficulties with discounting is that *we lost the policy improvement theorem with function approximation*. It is no longer true that if we change the policy to improve the discounted value of one state then we are guaranteed to have improved the overall policy. This lack of theoretical guarantees is an ongoing area of research.

10.5 Differential Semi-gradient n-step SARSA

To generalize to n-step bootstrapping, we need an n-step version of the TD error. n-step return:

$$G_{t:t+n} = R_{t+1} - \bar{R}_{t+1} + R_{t+1} - \bar{R}_{t+2} + ... + R_{t+n} - \bar{R}_{t+n} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1})$$

$\bar{R}_t$ is an estimate of $r(\pi)$, $n \geq 1$ and $t + n < T$. If $t + n \geq T$, then we define

$G_{t:t+n} = G_t$ as usual.

The n-step TD error is then:

$$\delta_t = G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w})$$

And then we can apply the usual semi-gradient SARSA update.

10.6 Learning Objectives (UA RL MOOC)

Lesson 1: Episodic Sarsa with Function Approximation

1. Explain the update for Episodic Sarsa with function approximation

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_t + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t)$$

2. Introduce the feature choices, including passing actions to features or stacking state features

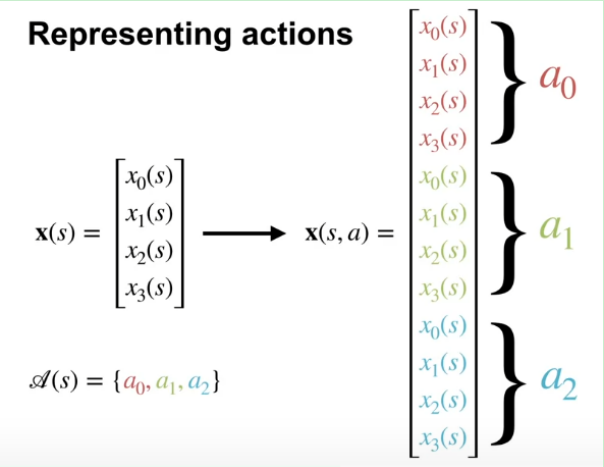


Figure 28: feature stacking

3. Visualize value function and learning curves

4. Discuss how this extends to Q-learning easily, since it is a subset of Expected Sarsa

Expected SARSA:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_t + \gamma \sum_{a'} \pi(a'|S_{t+1})\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t)$$

Q-learning:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_t + \gamma \max_{a'} \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Lesson 2: Exploration under Function Approximation

5. Understanding optimistically initializing your value function as a form of exploration

Recall Optimistic Initial values in the tabular setting to encourage early exploration. Note that it's unclear how to optimistically init values with nonlinear function approximators like NN.

Lesson 3: Average Reward

6. Describe the average reward setting

Instead of using discount factor, we consider all reward equally.

7. Explain when average reward optimal policies are different from discounted solutions

Recall the LeftRight8ring example, the optimal policies changes with the choice of γ. However, the average reward optimal policies gives the same result

8. Understand how differential value functions are different from discounted value functions

Discounted value functions uses discount factor on future rewards, while differential value functions subtract a constant $r(\pi)$ from all rewards.

11 Off-policy Methods with Approximation

- The extension to function approximation is significantly different and difficult for off-policy learning, than it is for on-policy learning
- tabular off-policy methods
  - extend to semi-gradient algorithms
  - do not converge robustly compared to when they under on-policy trainings.
- convergence problem
- notion of learnability
- introduce new algorithms for off-policy case with stronger convergence
- off-policy
  - prediction case: both π and b are static and given, we learn either $\hat{v} \approx v_\pi$ or $\hat{q} \approx q_\pi$.
  - control case: $\hat{q}$ are learned, π and b changes during learning.)
- **challenges**
  - (mismatched) target of the update (solution: semi-gradients, importance sampling)
  - (mismatched) distribution of the updates, like distribution shift in offline RL? (solution: (1) importance sampling; (2) true gradient methods)

11.1 Semi-gradient Methods

Rule change from tabular form to semi-gradient form: replace (update to array/table V, Q) $\rightarrow$ (update to weight vector $\boldsymbol{w}$ using $\hat{v}, \hat{q}$ and its gradient).

11.2 Examples of Off-policy Divergence

12 Eligibility Traces

13 Policy Gradient Methods

Action-valued method: learns the values of actions, then select actions based on estimated action values. If no action-value estimate, then no policy.

On the contrary, a **parameterized policy** can select actions w/o consulting a value function (a value function may still be used to learn the policy parameter, but isn't required for action selection).

Policy gradient method: Let $J(\boldsymbol{\theta})$ be the scalar performance measure w.r.t policy parameter. We want to maximize performance (so gradient ascent, note the **plus** sign):

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

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure w.r.t θ_t. PG methods not necessarily learn an approximate value function.

actor-critic method: learn approximations of both policy (actor) and value functions (critic).

- episodic case: performance = the value of the start state under the parameterized policy $J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(S_0)$
- continuing case: performance = average reward rate

13.1 Policy Approximation and its Advantages

In PG methods, policy can be parameterized in any way, as long as the policy $\pi(a|s,\theta)$ is differentiable w.r.t parameters.

Definition 13.1 ***soft-max in action preferences***: *action preference* $h(s,a,\boldsymbol{\theta})$ *can be parameterized arbitrarily.*

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

For example, the preference can be linear $h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^T\boldsymbol{x}(s,a)$

Advantages of PG:

- Action-value methods have no natural way of finding stochastic optimal policies, whereas PG methods can
- policy may be simpler to approximate
- policy parameterization is sometimes a good way to impart prior knowledge
- action probabilities change smoothly
- stronger convergence guarantees for PG than action-value methods

13.2 The Policy Gradient Theorem

Definition 13.2 ***policy gradient theorem***

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

In the episodic case, the constant of proportionality is the average length of an episode, and in the continuing case it is 1 (which is equality actually). μ is the distribution under π.

13.3 REINFORCE: Monte Carlo Policy Gradient

The RHS of the PG theorem is a sum over states weighted by how often the states occur under the target policy π.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta})] \\
&= \mathbb{E}_\pi[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}] \\
&= \mathbb{E}_\pi[q_\pi(S_t,A_t) \frac{\nabla \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}] \\
&= \mathbb{E}_\pi[G_t \frac{\nabla \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}]
\end{aligned}$$

Therefore, we have the REINFORCE update:

$$\begin{aligned}\boldsymbol{\theta_{t+1}} &\doteq \boldsymbol{\theta_t} + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta_t})}{\pi(A_t|S_t, \boldsymbol{\theta_t})} \\ &= \boldsymbol{\theta} + \alpha G_t \nabla ln \pi(A_t|S_t, \boldsymbol{\theta})\end{aligned}$$

The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return (because it causes the parameter to move most in the directions that favor actions that yield the highest return), and inversely proportional to the action probability (because we want to prevent same actions to not be constantly selected, resulting in local optimum).

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

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, ..., T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla ln \pi(A_t|S_t, \boldsymbol{\theta})$

13.4 REINFORCE with Baseline

We include an arbitrary baseline $b(s)$ (as long as it doesn't vary with a) to (1) strictly generalize REINFORCE; (2) reduce variances w/o introducing bias; (3) dynamically adjust to different states for action selection, (and thus faster learning).

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta})$$

Therefore, the update rule is

$$\begin{aligned}\boldsymbol{\theta_{t+1}} &\doteq \boldsymbol{\theta_t} + \alpha(G_t - b(S_t))\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta_t})}{\pi(A_t|S_t,\boldsymbol{\theta_t})} \\ &= \boldsymbol{\theta} + \alpha(G_t - b(S_t))\nabla ln\pi(A_t|S_t,\boldsymbol{\theta})\end{aligned}$$

A natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t, \boldsymbol{w})$.

REINFORCE with baseline (episodic) for estimating π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\boldsymbol{w})$
Algorithm parameter: step size $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weight $\boldsymbol{w} \in \mathbb{R}^d$
Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, ..., T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G - \hat{v}(S_t,\boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}}\delta\nabla\hat{v}(S_t,\boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\gamma^t\delta\nabla ln\pi(A_t|S_t,\boldsymbol{\theta})$

13.5 Actor-Critic Methods

why go from baseline methods to AC? In REINFORCE with baseline, the learned $\hat{v}(s)$ estimates only the first state of each state transition. It is made prior to the transition's action and thus cannot be used to assess that action. In AC methods, the $\hat{v}(s)$ is also applied to the second state of the transition (which forms a one-step return), so we can assessing that action. This way of assessing actions is called a *critic*.

Note that one-step return introduces bias, but it is often superior to the actual return for lower variance and easier computation. We can fix the bias with n-step returns and eligibility traces.

Definition 13.3 ***one-step AC*** *replace the full return of REINFORCE with the*

one-step return. The baseline of REINFORCE may be a $\hat{v}(s)$, but AC must use $\hat{v}(s)$ as the baseline.

$$\begin{aligned}\boldsymbol{\theta_{t+1}} &\doteq \boldsymbol{\theta_t} + \alpha(\underbrace{G_{t:t+1}}_{\textit{one-step return}} - \hat{v}(S_t, \boldsymbol{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta_t})}{\pi(A_t|S_t, \boldsymbol{\theta_t})} \\ &= \boldsymbol{\theta_t} + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta_t})}{\pi(A_t|S_t, \boldsymbol{\theta_t})}\end{aligned}$$

Since we are using one-step return, the one-step AC is full online and incremental.

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

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \boldsymbol{w})$
Algorithm parameter: step size $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weight $\boldsymbol{w} \in \mathbb{R}^d$
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S != $terminal$ (for each time step):
    $A \sim \pi(\cdot|S, \theta)$
    Take action A, observe S', R
    $\delta \leftarrow R + \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}}\delta\nabla\hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla ln\pi(A_t|S_t, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

AC with Eligibility Traces (episodic) is available in the book. We just declare eligibility trace vectors for both $\boldsymbol{\theta}$ and $\boldsymbol{w}$, and a trace-decay rates λ. We decay the the γ for both ET vectors at each update.

13.6 Policy Gradient for Continuing Problems

Recall in chap 10.3, we introduce the average reward for continuing problems and a steady-state distribution $\mu(s)$.

$$\sum_s \mu(s) \sum_a \pi(a|s, \boldsymbol{\theta}) p(s'|s, a) = \mu(s') \quad \forall s' \in \S$$

The pseudocode of AC with ET (continuing) is similar to AC with ET (episodic), except that we (1) replace reward with (reward - mean reward), and the mean reward is update as $\overline{R} \leftarrow \overline{R} + \alpha^{\overline{R}}\delta$; (2) remove γ decay.

13.7 Policy Parameterization for Continuous Actions

Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution.

Recall the probability density function is:

$$p(s) \doteq \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

To use PDF in policy parameterization

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} exp(-\frac{(a-\mu(s, \boldsymbol{\theta}))^2}{2\sigma(s, \boldsymbol{\theta})^2})$$

where we often declare that

$$\mu(s, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu^T \boldsymbol{x}_\mu(s)$$

$$\sigma(s, \boldsymbol{\theta}) \doteq exp(\boldsymbol{\theta}_\sigma^T \boldsymbol{x}_\sigma(s)) > 0$$

13.8 Summary

- In contrary to action-value methods, policy gradient methods perform action selections w/o consulting action-value estimates

- Advantages of learn&store policy parameters
  - learn specific probabilities for taking the actions
  - learn appropriate levels of exploration
  - approach deterministic policies asymptotically
  - can handle continuous action spaces
- policy gradient theorem provides a strong theoretical foundation
- REINFORCE directly follows PG theorem
- REINFORCE w/ baseline reduces variance w/o new bias
- AC, critic introduces bias, but bootstrapping (TD) is superior than MC (substantially reduce variance)

13.9 Learning Objectives (UA RL MOOC)

Lesson 1: Learning Parameterized Policies

1. Understand how to define policies as parameterized functions

Parameterized policy: $\pi(a|s, \boldsymbol{\theta})$. Parameterized value function: $\hat{q}(s, a, \boldsymbol{w})$

Constraints on the policy parameterization

- $\pi(a|s, \boldsymbol{\theta}) \geq 0 \quad \forall a \in \mathcal{A}, s \in \S$
- $\sum_{a \in \mathcal{A}} \pi(a|s, \boldsymbol{\theta}) = 1 \quad \forall s \in \S$

2. Define one class of parameterized policies based on the softmax function

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b \in \mathcal{A}} e^{h(s,a,\boldsymbol{\theta})}}$$

h function is action preference (it can be parameterized in any way), the exponential function guarantees the probability is positive for each action. The denominator normalizes the output of each action s.t. the sum over actions is one. Note action preferences != action values; only the difference between preferences is important.

3. Understand the advantages of using parameterized policies over action-value

based methods

- Parameterized policies can autonomously decrease exploration over time. Specifically, the policy can start off stochastic to guarantee exploration; as learning progresses, the policy can naturally converge towards a deterministic greedy policy
- They can avoid failures due to deterministic policies with limited function approximation
- Sometimes the policy is less complicated than the value function

Lesson 2: Policy Gradient for Continuing Tasks

4. Describe the objective for policy gradient algorithms

Our objective:

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

- $\sum_{s',r} p(s', r|s, a) r$ is $\mathbb{E}[R_t|S_t = s, A_t = a]$ expected reward if we start in state s and take action a
- $\sum_a \pi(a|s, \boldsymbol{\theta}) \sum_{s',r} p(s', r|s, a) r$ is $\mathbb{E}[R_t|S_t = s]$ expected reward of state s
- $r(\pi)$ is $\mathbb{E}_\pi[R_t]$ overall average reward by considering the fraction of time we spend in state s under policy π.

Optimizing the average reward objective (policy gradient method):

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

The main challenge is modifying policy changes the distribution $\mu(s)$; in contrast, recall in $\overline{VE}$ objective, the distribution is fixed. We do gradient ascent for PG, while gradient descent in optimizing the mean squared value error.

5. Describe the results of the policy gradient theorem

Note that chain rule of gradient requires to estimate $\nabla\mu(s)$, which is difficult to estimate since it depends on a long-term interaction between the policy and the environment. The PG theorem proves we don't need that, and following is our

result:

$$\nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla \pi(a|s, \boldsymbol{\theta}) q_\pi(s, a)$$

$\sum_a \nabla \pi(a|s, \boldsymbol{\theta}) q_\pi(s, a)$. This is a sum over the gradients of each action probability, weighted by the value of the associated action. $r(\pi)$ takes the above expression and sum that over each state. This gives the direction to move the policy parameters to most rapidly increase the overall average reward.

6. Understand the importance of the policy gradient theorem

Lesson 3: Actor-Critic for Continuing Tasks

7. Derive a sample-based estimate for the gradient of the average reward objective

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta_t})}{\pi(A_t|S_t, \boldsymbol{\theta_t})} q_\pi(S_t, A_t)$$

8. Describe the actor-critic algorithm for control with function approximation, for continuing tasks

Full algorithm see chap 13.6

- actor: parameterized policy
- critic: value functions, evaluating the actions selected by the actor

In the update rule, we don't have access to q_π, so we have to approximate it. For example, one-step bootstrap return TD method:

$$q_\pi(s, a) = R_{t+1} - \overline{R} + \hat{v}(S_{t+1}, \boldsymbol{w})$$

. To further improve, we subtract a baseline to reduces the update variance (results in faster learning):

$$q_\pi(s, a) = R_{t+1} - \overline{R} + \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$$

. Note this is equals the TD error δ.

Lesson 4: Policy Parameterizations

9. Derive the actor-critic update for a softmax policy with linear action preferences

$$\boldsymbol{w} = \boldsymbol{w} + \alpha^{\boldsymbol{w}}\delta \underbrace{\nabla \hat{v}(S, \boldsymbol{w})}_{\boldsymbol{x}(s)} \quad \text{just like semi-gradient TD}$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta \quad \underbrace{\nabla ln\pi(A|S, \boldsymbol{\theta})}_{\boldsymbol{x}_h(s,a) - \sum_b \pi(b|s,\boldsymbol{\theta}) x_h(s,b)}$$

10. Implement this algorithm

11. Design concrete function approximators for an average reward actor-critic algorithm

12. Analyze the performance of an average reward agent

13. Derive the actor-critic update for a gaussian policy

Probability density means that for a given range, the probability of x lying in that range will be the area under the probability density curve.

Gaussian Distribution

$$p.d.f = p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

μ = mean of distribution, σ = standard deviation = $\sqrt{variance}$. Gaussian Policy

$$p.d.f = \pi(a|s, \boldsymbol{\theta}) \doteq \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} exp(-\frac{(a-\mu(s, \boldsymbol{\theta}))^2}{2\sigma(s, \boldsymbol{\theta})^2})$$

$\mu(s, \boldsymbol{\theta})$ can be any parameterized function. σ controls the degree of exploration; usually initialized to be large s.t a wide range of actions are tried. As learning process, we expect the variance to shrink and the policy to concentrate around the best action in each state. The agent can reduce the amount of exploration through learning.

For θ,

$$\nabla ln\pi(a|s, \boldsymbol{\theta}_\mu) = \frac{1}{\sigma(s, \boldsymbol{\theta})^2}(a - \mu(s, \boldsymbol{\theta}))\boldsymbol{x}(s)$$

For σ,

$$\nabla ln\pi(a|s, \boldsymbol{\theta}_\sigma) = (\frac{(a - \mu(s, \boldsymbol{\theta}))^2}{\sigma(s, \boldsymbol{\theta})^2} - 1)\boldsymbol{x}(s)$$

Advantages of Continuous Actions

- it might not be straightforward to choose a proper discrete set of actions
- continuous actions allow use to generalize over actions

14. Apply average reward actor-critic with a gaussian policy to a particular task with continuous actions

For discrete actions, we use softmax policy parameterization; for continuous actions, we use Gaussian policy.

14 Psychology (skip)

15 Neuroscience (skip)

16 Application and Case Studies (skip)

17 Frontiers

17.1 General Value Functions and Auxiliary Tasks

17.2 Temporal Abstraction via Options

17.3 Observation and State

18 Things to consider for an algorithm

18.1 associative vs. non-associative

18.2 stationary vs. non-stationary

18.3 unbiased vs. biased

learning curve, parameter setting (robustness)

ability to learn

initialization (easy / hard) how?

update rule

memory computation

complexity

convergence guarantee

18.4 online vs. offline

18.5 short term vs. long term

18.6 exploitation vs. exploration

18.7 constant vs. non-constant step-size parameters

18.8 partial observability

related to non-stationary

18.9 episodic vs. continuing tasks

finite vs. infinite

size of state / actions / reward space

18.10 deterministic vs. stochastic policy

18.11 linear vs. nonlinear function approximation

18.12 exact (tabular) vs. approximate solution

18.13 discrete vs. continuous

space / task

18.14 expected vs. sampled return

18.15 forward vs. backward view

18.16 prediction vs. control

Q: do control algorithms always use state-action pairs?

18.17 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(λ)	
policy based	REINFORCE	
	REINFORCE with Advantage	
actor-critic	A3C	DDPG
	A2C	TD3
	TRPO	SAC
	PPO	IMPALA

19 Random Notes

19.1 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 $\arg\max_x f(x) = 6$.

19.2 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.