Reinforcement Learning

November 3, 2020

Abstract

Topics: goal-directed learning, MDP, value functions and backup diagrams, value iteration and policy iteration, model-free learning, function approximation and deep reinforcement learning.

Readings: Chapters 3-6, 9 of RL.

1 Motivations of RL

A more advanced sort of AI should be more human-like and be able to learn to achieve a preset goal by interacting with the environment. A baby learns to walk by seeing and touching the surrounding (the environment) and act on adults, chairs, desks and other entities, which give feedback to the baby (walking into a desk hurts). The goal is to walk and learning to walk is an example of goal-directed learning. Another example is learning to play chess: a computer agent or a human learns to play chess by playing the game and received feedback when winning or losing the game.

Such goal-directed learning task can hardly be handled by supervised learning or unsupervised learning alone, and reinforcement learning is needed. Different from supervised learning, the data from which an agent can learn are not given but generated as the agent interacts with the environment. The data points collected are most likely dependent, while many (not all) supervised learning algorithms assume that training examples are I.I.D. Lastly, the interactions between the agent and environment are better modeled as a sequential decision making problem, rather than a one-shot prediction in supervised learning. Unsupervised learning aims to discover structures from unlabeled data, while reinforcement learning aims to learn to achieve a goal.

There are connection between RL and supervised/unsupervised learning as well. Usually the policies are implemented as a classification model, such as a logistic regression or neural network. Unsupervised learning can be used to enhance state representation, e.g., when learning to play video game with a pre-trained deep network encoding images with raw pixels.

2 Markov Decision Processes

The agent-environment interactions and goal-directed learning can be mathematically described by Markov Decision Processes (MDP). An MDP consists of the following elements:

- State space $\mathcal{S}$: all states of an environment. A state of an environment, roughly speaking, is the configuration and status of the environment, such as the locations of objects and the agent in the environment. At any time t, the agent can sense the current state $S_t \in \mathcal{S}$. We use captital S_t to denote the variable representing the state at time t, and the lower-case s is a specific state so that $S_t = s$ means that the agent is at state s at time t. If we are not referring to any particular state, we just just S_t. How the states are represented is a question for the learning algorithm.

- Action space $\mathcal{A}(S_t)$: the set of all actions A_t that can be taken by the agent at the state S_t. $A_t = a$ means that the agent takes action $a \in \mathcal{S}_t$ at time t.

- Reward R_t: the agent will receive a reward, a numerical quantity, immediately as the result of taking action A_t under the state S_t. We assume that there are finite possible reward values, then R_t is a random variable following the distribution $\Pr(R_t = r|S_t = s, A_t = a)$. This is

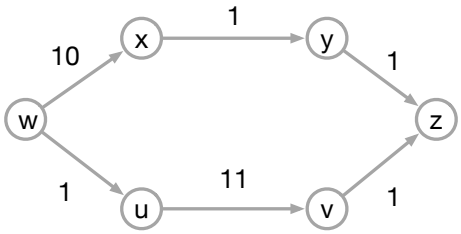

Figure 1: Finding the shortest path from node w to node z can be modeled as an MDP.

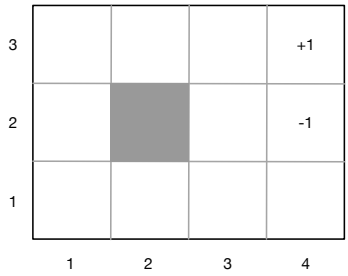

Figure 2: The example is reproduced from Figure 21.1 of the book "AI: a modern approach".

how a goal is encoded: by choosing the actions carefully, the agent aims to collect as high cumulative rewards, called "return", as possible, along the interactions with the environment.

- Transition probability: we consider the state S_t as a random variable that follows the distribution $\Pr(S_t = s|S_{t-1} = s', A_{t-1} = a)$. That is, the environment will take the agent to the state s following some probability when the action a is taken under the previous state s'. The transition probability indicates that the previous state s' fully summarizes the history of the interactions, so that no matter how s' was reached, taking action a at s will have the same probability $\Pr(S_t = s|S_{t-1} = s', A_{t-1} = a)$ to get to the state s. We say that the transition is *Makovian*.

- Discounting factor $0 \leq \gamma \leq 1$: it denotes the relative importance of short-term rewards against long-term rewards. It is meaning will become clear when we explain value functions.

Example 2.1. The shortest path problem on a graph is an example of MDP. In the directed graph in Figure 1, the states are the nodes on the graph $\mathcal{S} = \{w, x, y, u, v, z\}$. The action at each state is the possible next node(s) that can be reached from the current node, such as $\mathcal{A}(S_t = a) = \{x, u\}$. The reward of taking an action a at a node s is the negative of the cost on the edge (s, a), since minimizing the total cost of going from node w to node z is equivalent to maximizing the negative total cost. If we assume the agent can go to the node that it selects, then the transition probability is deterministic, i.e., $\Pr(S_t = s|S_{t-1} = s', a) = 1$, where s is the selected node.

Example 2.2. This is a more complex MDP. The goal is to find one of the exits (4,1) and (4,2) of the maze, starting from the cell (1,1). Each cell is a state and there are 11 states. The cell (2,2) is an obstacle. There are four actions at each state: Up, Down, Left, and Right. If an action leads to a wall or the obstacle, the agent will stay where it was before the action. Exiting from (4,1) will gain reward +1 and exiting from (4,2) will gain reward -1. These two states, (4,1) and (4,2) are called "terminal states", as the agent stops moving once a terminal state is reached. Since the agent wants to exit the maze as fast as possible, each transition into a state (except the terminal states) has reward -.0.04. The environment is stochastic: the agent can take its intended action with probability 0.8, but will be directed to one of the two directions at right angles to the intended action, each with probability 0.1.

This example shows the necessity of reinforcement learning. As the environment is not deterministic, a fixed sequence of actions can be a disaster. For instance, the sequence of actions $\{Up, Up, Right, Right, Right\}$ will have a non-zero chance to get into the state (4,2). Rather, the agent needs to learn how to act at each state.

Sometimes the reward distribution and transition distribution are combined into a joint distribution, called the "dynamics" of the MDP,

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

so that, by marginalization, we have

$$\Pr(S_t = s' | S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} \Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a), \tag{2}$$

$$\Pr(R_t = r | S_t = s, A_t = a) = \sum_{s' \in \mathcal{S}} \Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a). \tag{3}$$

Given an MDP, an agent's interactions with the environment consist of selecting an action A_t under state S_t and the environment responds with the next state S_{t+1} and reward R_{t+1} according to the environment dynamics. The following trajectory can be generated

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \ldots \tag{4}$$

A trajectory can be finite or infinite.

The goal of the agent is to maximized the return from a trajectory starting from any time t at any state S_t, defined as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{5}$$

For finite trajectories, γ may take value 1, meaning that all rewards are equally important. For infinite trajectories, it is important to set $\gamma < 1$ to ensure the infinite weighted sum converges to a finite value.

Example 2.3. In the maze example, the following trajectory will happen with probability 0.8^5

(1,1), Up, -0.04,
(1,2), Up, -0.04,
(1,3), Right, -0.04,
(2,3), Right, -0.04,
(3,3), Right, -0.04,
(4,3), NoAction, 1.

The return of this trajectory with $\gamma = 1$ (no discouting) is 0.8.

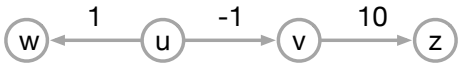

Figure 3: An MDP with deterministic transition probability. Starting from u, the agent can pick w or v as an action. Maximizing reward and maximizing return at state u result in rather different outcomes.

Example 2.4. In Figure 3, if the agent does not maximize the cumulative return G_t but the immediate reward R_t, then it will select action w at state u, since the action v leads to -1 reward. However, if $\gamma > 0$ is sufficiently large, maximizing the return will require the agent to take action v at u, since a larger reward by transiting from v to z can be obtained, with a un-discounted return of 9.

2.1 Policies and value functions

Since all the rewards $R_{t+1}, \ldots$ are random variables depending on the actions to be taken and the states $S_{t+1}, \ldots$ that the agent will be taken to according to the transition probability, the return itself is a random variable. By taking different actions under the same state S_t, the dynamics will lead to different R_{t+1} and S_{t+1}, where the agent may also choose different actions, leading to

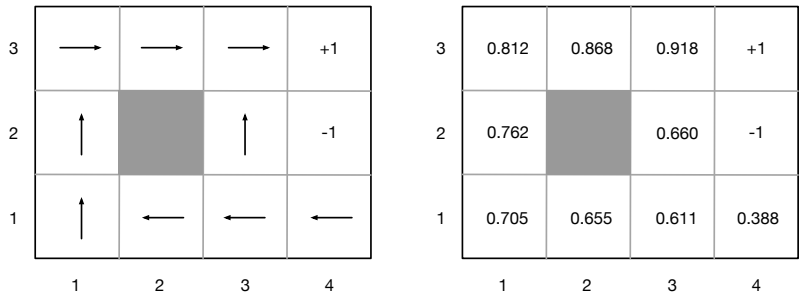

Figure 4: *Left*: An optimal policy of the maze example from Figure 2. *Right*: the state-value function of the optimal policy.

different reward R_{t+2}, and so on. The way that actions are chosen by an agent is called a *policy*, denoted by π. Formally, π is a probability distribution over the action space $\mathcal{A}(S_t)$:

$$0 \leq \pi(a|S_t) \leq 1, \sum_{a \in \mathcal{A}(S_t)} \pi(a|S_t) = 1. \tag{6}$$

An example (deterministic) policy is shown in the left panel of Figure 4. In general, at each state, a policy can have non-zero probabilities for multiple actions.

The agent's goal is to maximize the expected return with respect using an optimal policy. We define the expected return starting from a state s and acting according to a policy π as the *state-value function*, defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \ldots |S_t = s]. \tag{7}$$

The subscript π under the expectation is to denote that the expectation of the random variable $G_t = R_{t+1} + \gamma R_{t+2} + \ldots$ is taken over trajectories generated by following the policy π. To be more explicit, we can re-write the state-value function as:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s] && (8)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] && (9)\\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] && (10)\\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a)[r + \gamma v_\pi(s')]. && (11) \end{aligned}$$

The first and second equality is by the definition of the return G_t. The third equality is by writing down the expectation explicitly according to how R_{t+1} and G_{t+1} are generated through the interaction between the policy $\pi(a|s)$ and the dynamics of the MDP $\Pr(s', r|s, a)$. The last equality is by the definition of the state-value function. The summation over all triples (a, s', r) in the last equation defines an expectation, with the product probability distribution $\pi(a|s)\Pr(s', r|s, a)$ and random variable $r + \gamma v_\pi(s')$.

The equation is called the *Bellman equation* for v_π and is central to many reinforcement learning algorithms. The equation says: "the expected return obtained by starting from state s and following the policy π should be equal to the expected reward by taking action a according to π at state s, plus the discounted future expected return by following the policy π starting from the next state s'". The Bellman equation for v_π can be demonstrated using the *backup diagram* shown in the left panel of Figure 5. In the backup diagram, the label π denotes that an action a is sampled from $\pi(a|s)$. The label p denote the transition from s to s' when a is taken, and the label r is the reward received by taking action a at state s, with (s', r) sampled from the dynamics $\Pr(s', s|s, a)$ of the MDP.

A related value function is called *action value function* for policy π, defined as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s, A_t = a]. \tag{12}$$

$q_\pi(s, a)$ is thus the expected return when action a is taken at state s and π is followed afterward. The backup diagram for the action-value function is shown in the right panel of Figure 5. In

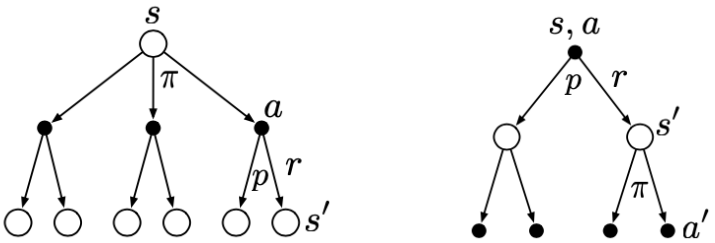

Figure 5: A circle represents a state and a solid dot represents an action. The arrows represent actions taken or transition to the next states. They are called "backup" diagrams because the values at the leave nodes will be backed up to the root, where the value functions will be computed.

this diagram, the labels of p, r, and π has the same meanings as the diagram for the state-value function. The action-value function can be more explicitly re-written as

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s'} \sum_{r} \Pr(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')]. \end{aligned} \tag{13}$$

This equation is the Bellman equation for q_π. The value of taking action a in state s and then following the policy π should be equal to the expected reward according to the dynamics of the MDP, plus a discounted expected return by taking the next action a' at the next state s' according to π and then follow the policy π.

2.2 Optimality policies and optimal value functions

Since the agent achieves the goal by maximizing the expected return, it should learn an optimal policy π^* so that

$$v_{\pi^*}(s) \geq v_\pi(s), \quad \forall s \in \mathcal{S}, \quad \forall \pi. \tag{14}$$

Note that there can be multiple optimal policies according to the above definition, but so long as a policy maximizes the state-value function at all states, it is optimal. There can be too many policies to be enumerated and evaluated to find the optimal ones. For example, in the maze example, there are 4^9 possible deterministic policies. We will see algorithms that can approximate the optimal policies in the next section. The algorithms will use the following Bellman optimality equation of an optimal policy π^*:

$$v_{\pi^*}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s', r} \Pr(s', r|s, a)[r + \gamma v_{\pi^*}(s')]. \tag{15}$$

That is, the value $v_{\pi^*}(s)$ of the optimal policy π^* should be achieved by taking the best action at state s, that leads to the maximum expected reward plus a discounted expected future return by following π^*. If either the selection of a or the policy for the next state s' were not optimal, the left hand side could not have been optimal.

The optimality Bellman equation follows the so-called "optimal substructure" in dynamic programming algorithms. In the shortest path problem, the current optimal action should have the smallest sum of immediate cost on the edge to take plus the length of the shortest path from the next node to the terminal node. See Figure 6. Another property of the Bellman optimality equation is that the value $v_{\pi^*}(s')$ of the next state s' can be re-used by other state that can transit to s'. This property is call "overlapping subproblems" in dynamic programming (considering finding the shortest path from s' as a subproblem that can be solved once and cached for future use).

Similarly, the Bellman optimality equation for q_{π^*} is

$$q_{\pi^*}(s,a) = \sum_{s', r} \Pr(s', r)[r + \gamma \max_{a' \in \mathcal{A}(s')} q_{\pi^*}(s', a')]. \tag{16}$$

Note that the action a in the argument of q_{π^*} may not be optimal for the state s. The above equation says, the maximum expected return by taking action a at state s should be equal to the

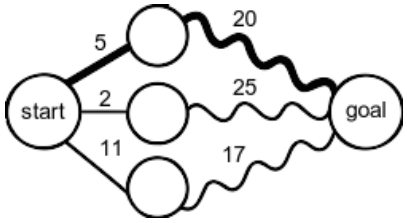

Figure 6: A demonstration of optimal substructure in dynamic programming. Source:https://en.wikipedia.org/wiki/Optimal_substructure

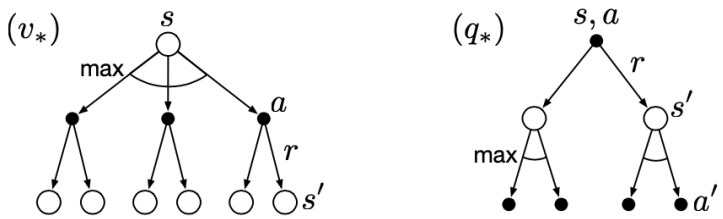

Figure 7: Backup diagrams for the Bellman optimality equations for v_π and q_π.

expected immediate reward plus the maximum future expected return by taking the best action in the next state s' and then follow the optimal policy. The above two Bellman optimality equations are demonstrated by the backup diagrams in Figure 7.

3 Dynamic programming

We can evaluate and optimize a policy for an MDP when the environment dynamics $\Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) = \Pr(s', r|s, a)$ of the MDP is fully known. The term "programming" means optimizing some policy, like in "linear programming", and the term "dynamic" means that the optimization is done for a sequence of interactions between an agent and an environment. There are three key algorithms, policy evaluation, policy iteration, and value iteration, that help find an optimal policy π^* with respect to an MDP.

3.1 Policy evaluation

Note that an optimal policy π^* is defined via its value function:

$$v_*(s) \geq v_\pi(s), \forall s \in \mathcal{S}, \forall \pi. \tag{17}$$

To know how well a policy π is doing and provide hints for improvement π towards the optimal policy π^*, one needs to evaluate the state-value function $v_\pi(s)$ for all states $s \in \mathcal{S}$, or the action-value function $q_\pi(s, a)$ for all state-action pairs (s, a). Policy evaluation is an iterative algorithm that uses the Bellman equations Eq. (11) or Eq. (13) to evaluate the value functions.

Take computing v_π as an example, for iterations $k = 1, \ldots,$ policy evaluation computes

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a)[r + \gamma v_k(s')]. \tag{18}$$

We use $v_k(s)$ to denote the value of state s at the k-th iteration. The backup diagram of this update is similar to the left panel of Figure 5, except that the root node will have value $v_{k+1}(s)$ after the backup and the leaf nodes will have the values $v_k(s')$.

In terms of implementation, the value function $v_k(s)$ is regarded as a vector of dimension $|\mathcal{S}|$ and $\mathbf{v}_k$ and $\mathbf{v}_{k+1}$ are two arrays storing the old and new v vectors. The above iteration is to use the array $\mathbf{v}_k$ to compute the right hand side of Eq. (18), and store the results in the array $\mathbf{v}_{k+1}$, which becomes $\mathbf{v}_k$ when $k \to k + 1$ in the next iteration. The values of all states are updated at the same time and this is called "synchronous update". The vectorized version of the synchronous policy evaluation can be written as

$$\mathbf{v}_{k+1} \leftarrow \mathbf{r}_\pi + \gamma P_\pi \mathbf{v}_k, \tag{19}$$

where $\mathbf{r}_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a) r$ is the expected immediate reward when following the policy π at state s, and P_π is the matrix of state-to-state transition probabilities

$$P_\pi(s, s') = \sum_a \pi(a|s) \sum_r \Pr(s', r|s, a). \tag{20}$$

Example 3.1. Refer to Example 4.1 and Figure 4.1 of the RL book [1] for an example of policy evaluation.

The action-value function $q_\pi(s, a)$ for a policy π can be evaluated in a similar way, using the Bellman equation for q function. This is left as a homework question.

3.1.1 Asynchronous updates

The synchronous updates are suitable for MDPs with a state space of manageable size, say millions. When the number of states is exponentially large, a typical situation for real-world applications of reinforcement learning, updating all states can be expensive or even computationally infeasible. Even with smaller state space sizes, one needs to wait for a long time for an iteration to sweep all states to see some progress. A related drawback is that the storage requirement is high since each state requires two copies of the value function. Lastly, there can be some states that need no update as they already have converged $v_\pi(s)$. The asynchronous updates of the state-value function will address the above drawbacks and can be written as

$$v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a)[r + \gamma v(s')]. \tag{21}$$

In the above update, the state s is selected randomly, or by an exploring policy, or by a prioritized sweeping that focuses on the states that made the largest previous updates. The single value function v is used on both sides of the above update. The condition for asynchronous update to converge is that all the states are visited infinitely often during the update iterations.

Asynchronous updates for the q function will be used when we describe the Q-learning algorithm.

3.2 Policy improvement and policy iteration

The evaluation of a policy is not the ultimate goal. Rather, we will go further to use the evaluation to improve the current policy π to π', so that $\pi'(s) \geq \pi(s)$ for any state $s \in \mathcal{S}$, hoping for convergence of π to an optimal policy π^*. This update of the policy is called policy improvement. To simplify the analysis, assume that we work with deterministic policies that has selects a single action deterministically at any state. We will analyze ϵ-greedy stochastic policies later.

The best action at the state s, according to the current evaluation of the policy π is

$$a \triangleq \pi'(s) = \arg\max_a q_\pi(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \tag{22}$$

Then we can improve the deterministic policy π to π', which is identical to π except at the state s:

$$q_\pi(s, \pi'(s)) = \max_a q_\pi(s, a) \geq \sum_a \pi(a|s) q_\pi(s, a) = v_\pi(s). \tag{23}$$

Note: here we abuse the distribution π' to make it a deterministic function $\pi'(s) \mapsto a$. The above inequality holds because the maximum of a set of numbers is no less than the average of the set. As a result, selecting $\pi'(s)$ at s and then following the original policy π (denoted by the subscript under q) leads to an improvement at all states:

$$v_\pi(s) \leq v_{\pi'}(s), \forall s \in \mathcal{S}. \tag{24}$$

We call this result the "Policy Improvement Theorem", proved as follows.

Proof.

$$
\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) && (25)\\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] && (26)\\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] && (27)\\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] && (28)\\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] | S_t = s] && (29)\\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] && (30)\\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) | S_t = s] && (31)\\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^2 v_\pi(S_{t+3}) | S_t = s] && (32)\\
&\leq \dots && (33)\\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] && (34)\\
&= v_{\pi'}(s). && (35)
\end{aligned}
$$

□

In the above proof, we use lower-case s to denote a given state, and upper-case, such as S_t and A_t, to denote random variables, of which an expectation is taken. For example, in the second line, A_t depends on π' and R_{t+1} and S_{t+1} in turn depends on A_t and the environment dynamics. We don't explicitly write down the expectation with respect to the dynamics, but implictly absorb it in the expectation signs.

3.2.1 Policy iteration

After the current policy π is improved to π', the policy evaluation algorithm can be run to evaluate π', and use the results to improve π' to π'', and so on, until there is no improvement can be done. The process generates a sequence of policies and value functions through alternative evaluation (E) and improvement (I) steps:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \dots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*. \tag{36}$$

At termination, the policy will be the optimal policy π^*, along with the corresponding value function v_*. This can be proved using the Contraction Mapping Theorem and the Bellman optimality equation [1].

Example 3.2. See Figure 4.2 of the RL book [1] for an example of policy iteration.

Any algorithms that alternates between policy evaluation and improvement steps are called the *Generalized Policy Improvement* (GPI) algorithm. More variations of GPI will be presented below with Monte Carlo and temporal difference.

3.3 Value iteration

In the policy iteration algorithm, running the embedded policy evaluation to convergence can be expensive or even infeasible, as it must go through all states in multiple iterations to guarantee the convergence of the evaluation, whether synchronous or asynchronous updates is used. Must one use the converged value functions for policy improvement? Figure 4.1 of the RL book shows that after only 3 iterations of policy evaluation, although the value function has not converged, the greedy policy extracted from the value function has already converged.

As a relaxation of the policy iteration, the policy improvement step can be executed right after one sweep of the states in the policy evaluation step:

$$
\begin{aligned}
v_{k+1}(s) &\leftarrow \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]\\
&= \max_a \sum_{s', r} \Pr(s', r | s, a)[r + \gamma v_k(s')]. && (37)
\end{aligned}
$$

[1] At convergence, the value function will not change, while the the policy is greedy with respect to that stablized value function. According to the Bellman optimality equation, the policy must be an optimal policy

Here the max operator is executing a policy improvement step and v_k is evaluated in a policy evaluation step. The expectation is to derive the $q_k(s,a)$ function from the evaluated v_k function. This equation is just turning the Bellman optimality equation Eq. (15) to value assignment. The Bellman optimality equation is applied iteratively over all states and it converges to the optimal policy π^* and optimal value function v_*. In order to avoid using two arrays for the value function and waiting from a long time for sweeping all the states, asynchronous updates can be used to update the value function, followed by the policy improvement step through the max operator:

$$v(s) = \max_a \sum_{s',r} \Pr(s', r|s, a)[r + \gamma v(s')]. \tag{38}$$

The action-value function $q(s,a)$ can be evaluated and improved in a similar way:

$$q(s,a) = \sum_{s',r} \Pr(s', r|s, a)[r + \gamma \max_{a'} q(s', a')]. \tag{39}$$

We will see that the Q-learning algorithm is an approximation of this update formula.

4 Monte Carlo Methods

Computing the value function (prediction) or optimizing a policy (control) using an environment's dynamics $\Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$ may not be feasible. In some cases, the true dynamics are unknown, such as in the Blackjack example in [1], and in the real-world supply chain management or portfolio management. In other cases, although the dynamics can be explicitly defined, but the computation of the expectation under the dynamics is computationally infeasible (e.g., in the Go game, enumerating all possible next states after one action takes exponential time, depending on the opponent's move). Monte Carlo methods are to make the policy evaluation and optimization feasible by taking the average of only finite samples rather than use the dynamics to evaluate the expectation in the Bellman equalities. This will be our first genuine reinforcement "learning" algorithm since the dynamics are not given but have to be estimated from sample data through agent-environment interactions.

In general, to evaluate the expectation of a random variable X under an distribution $\Pr(X)$, Monte Carlo (MC for short) methods use the average of m samples of X, denoted by $x^{(1)}, \ldots, x^{(m)}$, to estimate the true expectation:

$$\mathbb{E}[X] \approx \frac{1}{m} \sum_{i=1}^{m} x^{(i)}. \tag{40}$$

4.1 Monte Carlo prediction

The prediction problem is to evaluate a given policy π. Recall that in dynamic programming, the Bellman equation Eq. (11) is used to update the state-value function $v_\pi(s)$. MC prediction first follows the policy π to sample m trajectories, each of which looks like

$$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T. \tag{41}$$

One such trajectory is called an "episode" of executing the policy π. The sampled trajectories can have different lengths.

The expectation $\mathbb{E}_\pi[R_{t+1} + \cdots + R_T]$ on the right hand side of Eq. (11) for $v_\pi(s)$ is then replaced by the average of all returns G_t when starting from the state $S_t = s$:

$$G_t = \sum_{k=t+1}^{T-1} R_k, \quad S_t = s. \tag{42}$$

In plain English, $v_\pi(s)$ is the average of all sample returns starting from the state s.
Remark: it seems that we have to maintain a list of returns for each state. Since the algorithm is online, meaning that one trajectory is processed after the previous one, and there is an efficient incremental update of $v_\pi(s)$. We will discuss that after we present the MC control algorithms, which use the incremental update as well.

There are two versions of the MC estimation of $v_\pi(s)$. One is to average the returns collected at every time steps t when s appears in any episodes, including multiple appearance of s in a single episode. This version is called "every-visit" estimation. The other is called "first-visit" estimation, with the average taken over the time steps t when s first appears in an episode, with the remaining appearance(s) of s in the same episode discarded. These two estimations can result in rather different estimations, as shown in one of the homework questions. See the first-visit MC prediction Algorithm in Section 5.1 of the RL book.

The action-value function $q_\pi(s, a)$ can be estimated in a similar way, except that the returns G_t are collected after the appearance of the pair (s, a) at some time step in the sample episodes. There can also be every-visit and first-visit estimations. The problem of estimating $q_\pi(s, a)$ is less data-efficient than estimating $v_\pi(s)$ since there are more parameters ($|\mathcal{S}| \times |\mathcal{A}|$) of the q_π function than the v_π function ($|\mathcal{S}|$) to be estimated. If the sample episodes does not have an (s, a) pair appear somewhere, the action-value $q_\pi(s, a)$ cannot be estimated. In the extreme case, when π is deterministic, only one action a will appear along with each state s in all sample episodes. The estimation of $q_\pi(s, a)$ turns out to be critical for Monte Carlo control to be discussed next, as the dynamics $\Pr(s', r|s, a)$ is not available to recover $q_\pi(s, a)$ from $v_\pi(s)$ for policy improvement.

The advantage of MC prediction over dynamic programming is that no environment dynamics is required and it learns from samples. One disadvantage of MC is that it requires the sample trajectories to be of finite length, as the return G_t has to be calculated starting from R_{t+1} to R_T and T has to be finite. Therefore, Monte Carlo methods can't be used to learn from continuous MDPs. The temporal difference (TD) methods, to be discussed in the next section, will address this shortcoming.

4.2 MC on-policy control with Exploration Starts

The control problem is to optimize a policy. At the heart of policy optimization is the Policy Improvement Theorem. The minimum requirement for the Policy Improvement Theorem to work is that all (s, a) pairs are visited infinitely often during GPI. This can be guaranteed during trajectory generation, using *on-policy* or *off-policy* control.

Definition 4.1 (On-policy and off-policy)**.** On-policy methods attempt to evaluate and improvement the target policy π that is also the behavior policy used to sample trajectories, whereas off-policy methods evaluate or improve a target policy π using data generated using a behavior policy b, which is different from the target policy.

One significant difference between control using dynamic programming (DP) and MC methods is the availability of the environment dynamics $\Pr(s', r|s, a)$ for finding q_π using v_π (try to find a better policy π' so that $a = \arg\max_{a'} q(s, a')$ given the state-value function $v_\pi(s)$ only). As a result, q_π rather than v_π needs to be estimated for policy improvement.

The idea is to adopt the GPI (Generalized Policy Improvement) algorithm, introduced in the last section. Like policy iteration (which is an example of GPI), MC control alternates between policy evaluation and improvement steps until convergence:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \dots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*. \tag{43}$$

The evaluation steps can be done via first-visit or every-visit MC prediction for the action-value function. The improvement step modifies π at *some* state s by setting deterministically

$$\pi_{k+1}(s) = \arg\max_a q_{\pi_k}(s, a), \tag{44}$$

so that for *any* states s,

$$q_{\pi_k}(s, \pi_{k+1}(s)) = \max_a q_{\pi_k}(s, a) \geq v_{\pi_k}(s). \tag{45}$$

By the policy improvement theorem, π_{k+1} is better than π_k (i.e., $v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s)$).

Similar to value iteration, an evaluation step does not have to run to the convergence of the value function v_π before an improvement step can be made. For the policy improvement theorem to work, all (s, a) pairs need to be evaluated infinitely often during the GPI iterations to guarantee $v_{\pi_k}(s) \leq q_{\pi_k}(s, \pi_{k+1}(s))$. The algorithm in Section 5.3 of [1] gives a strategy called Exploring Starts, so that each (s, a) pair will have a positive probability to serve as (S_0, A_0) in a sample

trajectory. One trajectory contributes to the updates of multiple (s, a) pairs so long as (s, a) appears in the trajectory. A running average of the returns for each (s, a) pair is maintained for finding $q(s, a)$, which does not evaluate a fixed policy but a mixture of many policies as $\pi(s)$ is updated for each s appearing in the trajectory.

4.3 MC on-policy control using ϵ-greedy exploration

MC control with Exploration Starts is not feasible when the number of (s, a) pairs is too large or are hard to enumerate. A practical algorithm that will visit all (s, a) pairs, in the expectation sense, is the MC control with ϵ-greedy exploration. A policy is ϵ-greedy if it takes the optimal action with probability $1-\epsilon$, and randomly takes any action with probability $\frac{\epsilon}{|\mathcal{A}|}$. ϵ-greedy policies will explore all actions a available at a state with at least $\frac{\epsilon}{|\mathcal{A}|} > 0$ probability. For the details of the algorithm, see Section 5.4 of [1].

We prove that the minimum requirement of the Policy Improvement Theorem is satisfied by an ϵ-greedy policy.

Theorem 4.1 (Policy improvement for ϵ-greedy policies)**.** *Suppose we have evaluated the action-value function $q_\pi(s, a)$ for a policy π, and π' is the ϵ-greedy policy derived from $q_\pi(s, a)$:*

$$\pi'(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + (1-\epsilon) & \text{if } a = \arg\max_{a'} q_\pi(a'|s), \\ \frac{\epsilon}{|\mathcal{A}|} & \text{for other actions.} \end{cases} \tag{46}$$

$$\begin{aligned} \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] &= \sum_a \pi'(a|s) q_\pi(a, s) && (47) \\ &= \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s, a) + (1-\epsilon) \max_a q_\pi(s, a) && (48) \\ &\geq \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s, a) + (1-\epsilon) \sum_a \frac{\pi(a|s) - \epsilon/|\mathcal{A}|}{1-\epsilon} q_\pi(s, a) && (49) \\ &= \sum_a \pi(a|s) q_\pi(s, a) && (50) \\ &= v_\pi(s). && (51) \end{aligned}$$

The first and the last line is by the definitions of the value functions q and v. The second line is by the definition of the ϵ-greedy policy π'. The third line is due to $\frac{\pi(a|s)-\epsilon/|\mathcal{A}|}{1-\epsilon}$ is a distribution over actions ($\sum_a \frac{\pi(a|s)-\epsilon/|\mathcal{A}|}{1-\epsilon} = 1$).

By replacing $q_\pi(s, \pi'(s))$ with $\mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$ in the proof of Policy improvement Theorem for deterministic greedy policies, we can claim that $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$.

It is requires for $\epsilon \to 0$ as the Evaluation-Improvement cycles of GPI algorithm go, so that the ϵ-greedy policy will converge to a deterministic policy. For example, $\epsilon_k = 1/k$ where k is the number of the Evaluation-Improvement cycles and ϵ_k is the ϵ used for that cycle. It can be proved that the policy that GPI converges to is an optimal policy.

MC control with ϵ-greedy exploration is an on-policy reinforcement learning algorithm, since the policy to be optimized and used for trajectory generation are the same (i.e., the ϵ-greedy policy derived from the action-value function q_{π_k} is both the behavior and target policies.)
Remark: the GPI algorithm in Section 5.4 of [1] in fact does not evaluate q_{π_k}. (Why?)

4.4 Off-policy MC prediction and control

All reinforcement learning tasks involve the exploration-exploitation trade-off. Exploration is to visit as many states or state-action pairs as possible for evaluation, and exploitation is to optimize a policy to the optimal one, which may have no exploration capability (in the extreme case, an optimal policy is a deterministic policy). The on-policy MC control with ϵ-greedy policy will have less and less exploration capability as GPI iterations go. Another reason for off-policy control is to let the agent learn from and improve upon demonstrating trajectories generated by humans or other expert agents. Learning policy from data generated from other policies lead to more data efficiency.

Can we have a behavior policy to generate data to optimize the target policy? The behavior policy $b(a|s)$ used for trajectory generation does not have to be the same as the target policy $\pi(a|s)$ to be optimized. Trajectories

$$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T. \tag{52}$$

will be sampled by following policy b. The evaluation of the policy π using such trajectories will lead to a biased estimation of value functions q_π, since the MC estimation $\sum_{i=1}^{m} G_t^{(i)}$, with $G_t^{(i)}$ being the i's return starting from state s after taking action a, converges to $\mathbb{E}_b[G_t|S_t = s, A_t = a]$, rather than $\mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.

Importance Sampling (IS) can correct the bias when computing the mean of a random variable $X \sim P(X)$, sampled from another distribution Q, using the following identity,

$$\mathbb{E}_P[X] = \sum_x P(x)x = \sum_x Q(x)\frac{P(x)}{Q(x)}x = \mathbb{E}_Q\left[\frac{P(X)}{Q(X)}X\right]. \tag{53}$$

The ratio $P(x)/Q(x)$ is the weight of each sampled value x. The Monte Carlo version is then

$$\mathbb{E}_P[X] \approx \sum_{x^{(i)}\sim P} x^{(i)} = \sum_{x^{(i)}\sim Q} \frac{P(x)}{Q(x)}x^{(i)} \approx \mathbb{E}_Q\left[\frac{P(X)}{Q(X)}X\right]. \tag{54}$$

Importance sampling is useful when the target distribution P is difficult to sample from but has a closed-form density function.

Applying IS to the MC evaluation of value function of a policy π using trajectories sampled from b, the weight $P(x)/Q(x)$ would be the ratio of probability of a trajectory starting from S_t and then following the policy π

$$\Pr(S_t, A_t, R_{t+1}, S_{t+1}, \ldots, S_{T-1}, A_{T-1}, R_T) = \prod_{k=t}^{T-1} \pi(A_t|S_t)\Pr(S_{t+1}, R_{t+1}|S_t, A_t), \tag{55}$$

and that of following the policy b

$$\Pr(S_t, A_t, R_{t+1}, S_{t+1}, \ldots, S_{T-1}, A_{T-1}, R_T) = \prod_{k=t}^{T-1} b(A_t|S_t)\Pr(S_{t+1}, R_{t+1}|S_t, A_t), \tag{56}$$

where S_T is the terminal state. The the ratio becomes

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{57}$$

The requirement for b is that when $\pi(a|s) \neq 0$, $b(a|s)$ must also be non-zero: with non-zero probability, a can be taken in state s under π, and $b(a|s) = 0$ leads to infinity in the ratio.

Let m sampled trajectories be

$$\begin{gathered} S_0^{(1)}, A_0^{(1)}, R_{t+1}^{(1)}, S_{t+1}^{(1)}, \ldots, S_{T-1}^{(1)}, A_{T-1}^{(1)}, R_T^{(1)} \\ \vdots \\ S_0^{(m)}, A_0^{(m)}, R_{t+1}^{(m)}, S_{t+1}^{(m)}, \ldots, S_{T-1}^{(m)}, A_{T-1}^{(m)}, R_T^{(m)}. \end{gathered} \tag{58}$$

Assume that the pair (s, a) appears in the set of time steps $\mathcal{T}(s, a)$, with trajectory indices $i = 1, \ldots, m$ taken into account. The MC estimation of the expected return starting from (s, a) and following the *target* policy π is

$$q_\pi(s, a) \approx \frac{\sum_{t\in\mathcal{T}(s,a)} \rho_{t:T(t)-1}G_t}{|\mathcal{T}(s, a)|}, \tag{59}$$

where $T(t)$ is the ending time step of the trajectory that contains the time step t, since the trajectories can be of different lengths. Eq. (59) is called *ordinary* importance sampling. Note

that $\rho_{t:T(t)-1}$ can be quite small as it is a product of $T-t$ floating numbers. The *weighted* importance sampling addresses the scale of the ratio $\rho_{t:T(t)-1}$ and estimate the expected return using

$$q_\pi(s,a) \approx \frac{\sum_{t\in\mathcal{T}(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t\in\mathcal{T}(s,a)} \rho_{t:T(t)-1}}. \tag{60}$$

The weighetd IS has less variance while the ordinary IS can have infinite variance. See Section 5.5 of [1] for the description of an off-policy MC control algorithm. In practice, multiplying probabilities over multiple steps can lead to numerical issues and IS is more commonly seen in TD, where a single step is taken for value function estimation. See the next section.

The off-policy MC control algorithm is just to use MC estimation with IS to evaluate the current policy π, with trajectories generated from the behavior policy b. While π will eventually converge to the optimal policy π^*, b can remain the same policy that has a large exploration capability (e.g., b selects random actions with equal probabilities).

4.5 Incremental implementation

During prediction or control, the MC estimation of the value functions v_π or q_π involves storing sampled returns G_t in an array of indefinite (not necessarily *infinite*) length for each state s or state-action pair (s,a). Taking the average of the samples returns for each s or (s,a) every time π is updated can be a waste of time.

We derive the following incremental implementation of the average of an indefinite number of returns using constant space and time. Take estimating $q_\pi(s,a)$ as an example, with m sampled returns $\{G_{t(1)},\dots,G_{t(m)}\}$,

$$q_\pi^{(m)}(s,a) \approx \frac{1}{m}\sum_{i=1}^{m} G_{t(i)}. \tag{61}$$

Suppose now we have a new encounter of (s,a) at time t in some episode (i.e., $(S_t,A_t)=(s,a)$) and obtain the $(m+1)$-th sample return $G_{t(m+1)}$ for (s,a),

$$\begin{aligned}
q_\pi^{(m+1)}(s,a) &\approx \frac{1}{m+1}\sum_{i=1}^{m+1} G_{t(i)} && (62)\\
&= \frac{1}{m+1}\left(\sum_{i=1}^{m} G_{t(i)} + G_{t(m+1)}\right) && (63)\\
&= \frac{m}{m+1} q_\pi^{(m)}(s,a) + \frac{G_{t(m+1)}}{m+1} && (64)\\
&= \left(1-\frac{1}{m+1}\right) q_\pi^{(m)}(s,a) + \frac{G_{t(m+1)}}{m+1} && (65)\\
&= q_\pi^{(m)}(s,a) + \frac{1}{m+1}[G_{t(m+1)} - q_\pi^{(m)}(s,a)] && (66)
\end{aligned}$$

The update from $q_\pi^{(m)}(s,a)$ to $q_\pi^{(m+1)}(s,a)$ looks like a gradient ascent for the value function $q_\pi(s,a)$, using the gradient $G_{t(m+1)} - q_\pi^{(m)}(s,a)$ and learning rate $\frac{1}{m+1}$. Indeed, $G_{t(m+1)} - q_\pi^{(m)}(s,a)$ can be regarded as the gradient of the error $\frac{1}{2}[G_{t(m+1)} - q_\pi^{(m)}(s,a)]^2$ with respect to $q_\pi^{(m)}(s,a)$. We can think of $G_{t(m+1)}$ as the new target value for $q_\pi^{(m)}(s,a)$. We will see other estimation of the target value in the next section.

More generally, let the return G_t starting from (S_t,A_t) have weight $W_t(S_t,A_t)$ ($W_t(S_t,A_t)=1$ in the on-policy learning and $W_t(S_t,A_t)=\rho_{t:T(t)-1}$ in the off-policy learning with importance sampling). The incremental update is to maintain the total weight of any (S_t,A_t) pair as $C(S_t,A_t) = \sum_{t\in\mathcal{T}(S_t,A_t)} W_t(S_t,A_t)$ whenever a new appearance of (S_t,A_t) is encountered in a sample trajectory. Then the update of $q_\pi(S_t,A_t)$ at time t when (S_t,A_t) is encountered is

$$q_\pi(S_t,A_t) \leftarrow q_\pi(S_t,A_t) + \frac{W_t(S_t,A_t)}{C(S_t,A_t)}[G_t - q_\pi(S_t,A_t)]. \tag{67}$$

Verify that that this recovers Eq. (66) with $W=1$. See the end of Section 5.6 of [1] for the details of the MC control algorithm.

5 Temporal difference

One outstanding disadvantage of Monte Carlo is that it works only on episodic MDPs that will terminate within finite steps, and the target return G_t needed for the estimation is only available when an episode ends. To obtain the target faster, and to make learning possible on episodes with infinite length, temporal difference (TD) creates another sort of estimation that is more aggressive than Monte Carlo. We will present a comparison of DP, MC, and TD at the end of this section.

5.1 TD prediction

To evaluate a policy π, MC prediction updates the action-value function using Eq. (67), which can be generalized to

$$q_\pi(S_t, A_t) \leftarrow q_\pi(S_t, A_t) + \alpha[G_t - q_\pi(S_t, A_t)]. \tag{68}$$

where α is the learning rate and can be selected to a proper positive number.

In TD prediction, G_t is replaced with a bootstrapped estimation $R_{t+1}+\gamma q_\pi(S_{t+1}, A_{t+1})$, leading to

$$q_\pi(S_t, A_t) \leftarrow q_\pi(S_t, A_t) + \alpha[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) - q_\pi(S_t, A_t)]. \tag{69}$$

$q_\pi(S_{t+1}, A_{t+1})$ is the estimated return when following π starting from (S_{t+1}, A_{t+1}) and (S_{t+1}, A_{t+1}) is what follows the action A_t in a sample episode. $q_\pi(S_{t+1}, A_{t+1})$ is a bootstrapped estimation since at the time of the update, it may not be available. In practice, $q_\pi(S_{t+1}, A_{t+1})$ is initialized to be 0, so that its first estimation would be R_{t+1}.

We can regard $R_{t+1}+\gamma q_\pi(S_{t+1}, A_{t+1})$ as a sample for estimating the expectation in the Bellman equation for the action-value function Eq. (13), reproduced below:

$$q_\pi(s, a) = \sum_{s'} \sum_{r} \Pr(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')]. \tag{70}$$

Both the environment dynamics $\sum_{s'} \sum_r \Pr(s', r|s, a)$ and the agent policy $\pi(a|s)$ is sampled to obtain (R_{t+1}), S_{t+1}, and A_{t+1}.

Both MC and TD have the advantage of not requiring the knowledge of the dynamics $\Pr(s', r|s, a)$ that is needed by DP. By comparing the targets G_t for MC and $R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})$ for TD, we can see that G_t requires an entire episode to be available, while $R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})$ can be obtained immediately on the transition to S_{t+1}. Therefore, TD can work on episodes with infinite length. The updated q_π function may be used immediately to improve the current policy π, as we will see in TD control algorithms.

There is actually a more subtle difference between TD and MC in terms what's being estimated by the two methods, as shown by the following example (Example 6.4 of [1]).

Example 5.1. Let the eight sample trajectories from an unknown MDP by executing an policy π be

```
B, 0            B, 1
B, 1            B, 1
B, 1            B, 1
B, 1            A,0,B,0
```

Initialize the state-value function to be $v_\pi(A) = v_\pi(B) = 0$. According to first-visit MC prediction, the state-value function can be estimated as

$$v_\pi(B) = 3/4, \tag{71}$$
$$v_\pi(A) = 0. \tag{72}$$

But by TD estimation, one may estimate from the first seven episodes to obtain $v_\pi(B) = 6/7$, then process the last episode to obtain $v_\pi(A) = 6/7$ since B follows A immediately. Then $v_\pi(B)$ is updated to $3/4$.

The difference in $v_\pi(A)$ estimations according to MC and TD is due to the Markovian transition $S_t = A$ and $S_{t+1} = B$ and TD prediction uses the next state's value to estimate the current state's value. Implicitly, TD estimates that state B will always from state A according to the current policy and the unknown environment dynamics, according to the limited data.

5.2 TD control

We will discuss two control algorithms, SARSA and Q-learning.

5.2.1 SARSA: on-policy TD control

SARSA is a special case of GPI and it alternates between the evaluation and improvement steps. It uses TD prediction Eq. (69) to evaluate a policy π, and improves π by updating π to an ϵ-greedy policy according to the estimated q_π function.

Note that: 1) q_π must be used here since it is a model-free control problem (environment dynamics is not available); and 2) the ϵ-greedy policy does not have to be stored explicitly as it can be derived from the $q_\pi(s, a)$ function at any time. 3) SARSA gets its name from the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ in the sample episodes used in Eq. (69), and it is on-policy since A_{t+1} is selected according to the ϵ-greedy policy, which is at the same time the behavior and target policies. See Section 6.4 of [1] for more details of SARSA.

Conditions of convergence of SARSA to an optimal policy: 1) all state-action pairs are visited infinitely often during learning; and 2) the exploration rate $\epsilon \to 0$.

5.2.2 Q-learning: off-policy TD control

Q-learning is an off-policy learning algorithm. It learns the target policy that is greedy according to the q function

$$\pi(s) = \arg\max_a q(s, a). \tag{73}$$

Also, it has a behavior policy, which is ϵ-greedy according to the q function (see Eq. (46)). The evaluation and improvement step is combined into a single update to the q function:

$$q_\pi(S_t, A_t) \leftarrow q_\pi(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} q_\pi(S_{t+1}, a') - q_\pi(S_t, A_t)]. \tag{74}$$

We can see that the target $R_{t+1} + \gamma \max_{a'} q_\pi(S_{t+1}$ is obtained by executing the greedy target policy at the next state S_{t+1}, while A_t at any time is sampled according to the ϵ-greedy policy derived from the current q_π function. One way to interpret Q-learning is that, it applies the Bellman optimality equation for the action-value function Eq. (16) to the sample transitions $(S_t, A_t, R_{t+1}, S_{t+1})$. See Section 6.5 of [1] for more details of Q-learning.

6 A short summary

We have seen two tasks, prediction and control, and three families of algorithms, DP, MC, and TD. Let's summarize them in the follow table.

	DP	MC	TD
Prediction v_π	$v(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma v(S_{t+1})]$	$v(S_t) \leftarrow G_t$	$v(S_t) \leftarrow R_{t+1} + \gamma v(S_{t+1})$
Prediction q_π	$q(S_t, A_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma q(S_{t+1}, A_{t+1})]$	$q(S_t, A_t) \leftarrow G_t$	$q(S_t, A_t) \leftarrow R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$
Control q_*	$q_*(S_t, A_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma \max_a q(S_{t+1}, a)]$	$q(S_t, A_t) \leftarrow G_t$	$q_*(S_t, A_t) \leftarrow R_{t+1} + \gamma \max_a q(S_{t+1}, a)$

Table 1: Comparing different RL tasks and algorithms

7 Function approximation for RL

All the prediction and control algorithms we have discussed so far assume a table containing the value functions $v_\pi(s)$ or $q_\pi(s, a)$, and the table can be updated according to returns during policy evaluation and queried during policy improvement. These methods are called *tabular* methods. In real-world applications of RL, a state space can be rather large (e.g., the Go game has 10^{170} states), or can be continuous, with unaccountably many number of states (e.g., control of an autonomous driving vehicle involves multiple continuous readings from sensors). Building, updating, and accessing tables for value functions in such applications are not feasible. Another drawback of tabular methods is that the value functions do not generalized: considering two states that are similar and require similar actions, there is no such constraint enforced in tabular methods. For

example, in autonomous driving, if the vehicle is approaching a red light, it should slow down and then stop, and this behavior should be generalized to different locations where a red light is seen by the vehicle.

Function approximation for RL is to use supervised learning models, such as linear regression and neural networks, to approximate the value functions:

$$v_\pi(s) \approx \hat{v}_\pi(s; \mathbf{w}) \tag{75}$$

$$q_\pi(s, a) \approx \hat{q}_\pi(s, a; \mathbf{w}) \tag{76}$$

The value functions are now functions with parameter $\mathbf{w}$. These parametric functions can be implemented using linear regression, neural network, or even kernel methods.

During policy evaluation, the value functions need to be updated so that the functions will predict its target value better than it would before the update. For example, in tabular DP, $v_\pi(s)$ will be shifted towards the expected future return $\mathbb{E}_\pi[r + \gamma v_\pi(s')]$, and in tabular TD, $v_\pi(S_t)$ will be shifted towards the sampled return $R_{t+1} + \gamma v_\pi(S_{t+1})$. Supervised learning can be regarded as changing parameter $\mathbf{w}$ so that the model can predict the output accurately based on the input, and this is exactly the idea that policy evaluation is trying to do. Here the input is a state s or a state-action pair (s, a), and the output is the real-valued target that the value functions supposed to take at s or (s, a). Multiple pairs of input-output can be regarded as training data of the value functions.

While there are many supervised learning models, only some of them are suitable in the reinforcement settings. First, data are generated as trajectories when the agent interacts with the environment, and the models need to be able to learn as data are generated. This is called *online learning*. Supervised models that allow stochastic gradient descent training are preferred. Second, non-stationary data are abundant in reinforcement learning. For example, the DP and TD methods use bootstrapping, relying on the existing estimation of the value functions $\hat{v}_\pi(s; \mathbf{w})$ or $\hat{q}_\pi(s, a; \mathbf{w})$ to construct the target values. As the parameter $\mathbf{w}$ is updated, the target values are also changed (more discussions on this in the DQN section). In the control task, the policy π will be updated too so that the distribution of visiting the states or state-action pairs will be changed as well, and what was seen frequently may not be seen in future iterations. The model should adapt to the new data while not entirely forget about the old training data.

7.1 Prediction with stochastic-gradient and semi-gradient methods

Given a policy π, the goal of prediction in RL is to find value functions that are as close to the true value functions as possible. In terms of function approximation for $v_\pi(s)$, a supervised model is trained to minimize the following weighted MSE loss (called Mean Squared Value Error in RL)

$$\sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}_\pi(s; \mathbf{w})]^2, \tag{77}$$

where $v_\pi(s)$ is the unknown true value function under the policy π, and $\mu(s)$ is the importance of the state s, depending on how often the agent will visit s under π and the environment dynamics.

As the agent learns from interactions, it won't visits all states and then fit $\hat{v}_\pi(s; \mathbf{w})$ to $v_\pi(s)$. Rather, the agent receives observation of a state S_t and constructs an estimation of $v_\pi(S_t)$ (e.g., the return G_t from a trajectory using MC). $\hat{v}_\pi(s; \mathbf{w})$ is then shifted to that estimation. The minimization of the MSE loss is handled by stochastic gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} [v_\pi(S_t) - \hat{v}_\pi(S_t; \mathbf{w})]^2 \tag{78}$$

$$= \mathbf{w} + \alpha [G_t - \hat{v}_\pi(S_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{v}_\pi(S_t; \mathbf{w}) \tag{79}$$

as the agent continues to visit different states, it obtains a MC sample of the states distributed according to $\mu(s), s \in \mathcal{S}$.

The above stochastic gradient method works only when the target does not depend on $\mathbf{w}$. If we adopt TD value prediction with function approximation, we will be minimizing this MSE loss

$$\sum_t [R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w}) - \hat{v}_\pi(S_t; \mathbf{w})]^2. \tag{80}$$

Here the target $R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w})$ is the TD estimation of the true value of $v_\pi(S_t)$. We can see that $\mathbf{w}$ appears in both terms in the error. Stochastic semi-gradient descent adopts the following

update to learn $\mathbf{w}$:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w}) - \hat{v}_\pi(S_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{v}_\pi(S_t; \mathbf{w}), \tag{81}$$

assuming that the TD target $R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w})$ is a constant with respect to $\mathbf{w}$. The stochastic semi-gradient descent does not converge to a local optimum as reliably as stochastic gradient descent, due to the immediate change of the target value once $\mathbf{w}$ is updated. More on this when discussing DQN.

7.1.1 Linear models

Linear models as function approximation are easy to analyzed. For example, the stochastic semi-gradient descent does converge to a local optimum robustly, when linear model is used for $\hat{v}_\pi(s; \mathbf{w})$. A linear model constructs an n dimension feature vector of each state s, so that a state is represented by the vector $\mathbf{x}(s) \in \mathbb{R}^n$ and $\hat{v}_\pi(s; \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$. Stochastic gradient descent with MC updates $\mathbf{w}$ by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}_\pi(S_t; \mathbf{w})] \mathbf{x}(S_t), \tag{82}$$

since $\nabla_{\mathbf{w}} \hat{v}_\pi(s; \mathbf{w}) = \mathbf{x}(s)$. Stochastic semi-gradient descent with TD updates $\mathbf{w}$ by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w}) - \hat{v}_\pi(S_t; \mathbf{w})] \mathbf{x}(S_t). \tag{83}$$

See Section 9.3 of [1] for the descriptions of the prediction algorithms with MC and TD.

As the goal is to minimize Eq. (77), the weighted average of the value errors over all states that have a non-zero probability to be visited, the above (semi-)gradient descent methods don't continue to optimize $\mathbf{w}$ all the way, so that $G_t = \hat{v}_\pi(S_t; \mathbf{w})$ or $R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w}) = \hat{v}_\pi(S_t; \mathbf{w})$, as that will lead to high errors on other states. Also, as data distribution continues to change, optimizing $\mathbf{w}$ to minimize an instantaneous error is not meaningful. This latter point is more pertaining to RL.

7.2 Control with function approximation and DQN

Similar to control with tabular methods, the action-value function $q_\pi(s, a)$ needs to be evaluated. With function approximation, the evaluation of $q_\pi(s, a)$ can be regarded as minimizing the following MSE loss function:

$$\sum_{s \in \mathcal{S}, a \in \mathcal{A}} \mu_\pi(s, a)[q_\pi(s, a) - \hat{q}_\pi(s, a; \mathbf{w})]^2, \tag{84}$$

with $\mu(s, a)$ being the joint probability of seeing state s and taking action a at s under the policy π and environment dynamics $\pi(s', r|s, a)$.

The stochastic gradient descent for on-policy MC control on the encounter of (S_t, A_t) in some sampled episode is

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{q}_\pi(S_t, A_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{q}(S_t, A_t; \mathbf{w}), \tag{85}$$

where G_t is the return obtained at the end of this episode, starting from (S_t, A_t).

With TD methods, we can find a function approximation version of SARSA (on-policy TD control), using stochastic semi-gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{q}_\pi(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}_\pi(S_t, A_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{q}_\pi(S_t, A_t; \mathbf{w}) \tag{86}$$

The off-policy Q-learning algorithm can work with function approximation too, with the update:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \max_a \hat{q}_\pi(S_{t+1}, a; \mathbf{w}) - \hat{q}_\pi(S_t, A_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{q}_\pi(S_t, A_t; \mathbf{w}). \tag{87}$$

When linear models are used to approximate $q_\pi(s, a)$, $\nabla \hat{q}_\pi(s, a; \mathbf{w}) = \mathbf{x}(s, a)$, where $\mathbf{x}(s, a)$ is a feature vector of the (s, a) pair.

Training $\hat{q}_\pi(s, a; \mathbf{w})$ is not as easy as in training a supervised learning model. We will discuss the challenges and solution next, with DQN.

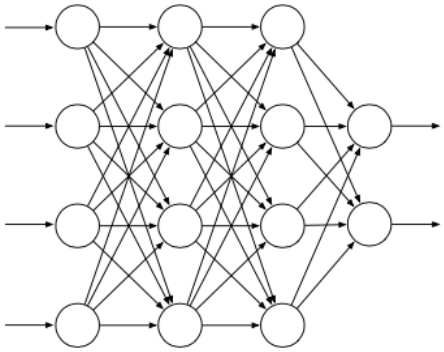

Figure 8: A MLP with input layer being the four raw features of the state and two output neurons for two actions $a = 0, 1$. An output neuron outputs $\hat{q}_\pi(s, a; \mathbf{w})$, given the input vector for the state s and the network parameter $\mathbf{w}$. DQN adopts a convolution neural network (CNN) that is more suitable for images.

7.2.1 Deep Q Network (DQN)

Using neural network as a function approximation of $q_\pi(s, a)$ has been seen as early as in the method called Neural Q Fitting in 2005. Basaed on recent progress in deep learning, in 2016, DeepMind published DQN that uses a deep neural network (convolution neural network), that can learn to play ATARI compute games from screenshots (images of raw pixels) only. There is no feature engineering for the feature vectors $\mathbf{x}(s, a)$. Rather, the deep network figures out a feature vector representation of the states by learning from the input images and rewards. DQN learned to outperform humans in playing some ATARI games.

Given all these advantages, DQN is however hard to train. Two major techniques led to the success of DQN, by making the Q-network training similar to supervised learning.

- **Experience replay**: in the stochastic semi-gradient descent for off-policy TD control, a sequence of (S_t, A_t) will be generated by agent-environment interactions and used to update the network parameter $\mathbf{w}$ in the order in which they are generated. Since the data in an episode are dependent, the distribution of such training data violates the I.I.D. assumption in supervised learning, including neural networks. For example, consecutive screenshots of an episode of game playing are highly similar, and parameter update using similar screenshots is biased and does not minimize the MSE Eq. (84).

  DQN uses a large buffer that can hold 1 million transitions encountered during training. As new screenshots are captured, older ones are thrown away if the buffer is full. Small mini-batches are randomly sampled from this large buffer to create a training set that is close to I.I.D.

- **Target network**: function approximation helps generalize experience from one state to some other states, but also make training non-stationary. In Eq. (87), once $\mathbf{w}$ is updated, the target $R_{t+1} + \gamma \max_a \hat{q}_\pi(S_{t+1}, a; \mathbf{w})$ is changed immediately, making the fitting inefficient. This is the problem of semi-gradient descent. DQN makes a copy of the Q-network, called "target Q-network" (denoted by $\mathbf{w}^{-1}$: the superscript -1 is not an "inverse" but to denote the lagging behind of the target Q-network). $\mathbf{w}^{-1}$ stays stationary over a period of training steps and is used to calculate the target $R_{t+1} + \gamma \max_a \hat{q}_\pi(S_{t+1}, a; \mathbf{w}^{-1})$, while the original Q-network $\mathbf{w}$ is being updated using this target value. From time to time, when $\mathbf{w}$ is supposed improved, the target network $\mathbf{w}^{-1}$ is updated to catch up with $\mathbf{w}$ and then remains the same for the next training period. The update of $\mathbf{w}$ is

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \max_a \hat{q}_\pi(S_{t+1}, a; \mathbf{w}^{-1}) - \hat{q}_\pi(S_t, A_t; \mathbf{w})] \nabla_{\mathbf{w}} \hat{q}_\pi(S_t, A_t; \mathbf{w}). \tag{88}$$

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.