Appendix

B

REINFORCEMENT LEARNING (RL)

B.1 INTRODUCTION

Reinforcement learning (RL) is a learning paradigm pertaining to *learning to control a system to maximize a numerical performance measure that expresses a long-term objective*. It is of immense interest owing to the huge volume of applications it can help address. The application of reinforcement learning in learning to play games has proved to be very impressive. Job-shop scheduling (seeking a schedule of jobs fulfilling temporal and resource limitations) is one of the several operation-related problems that RL can effectively address. Other problems include inventory control, maintenance problems, targeted marketing, vehicle routing, fleet management, elevator control, etc. RL is capable of addressing a lot of information theory problems, for instance, optimal coding, packet routing, and optimization of channel allocation or sensor networks. Another significant category of problems emanates from finance. These include optimal portfolio management and option pricing. RL is employed to handle problems in control engineering: optimal control of chemical or mechanical systems in process control and manufacturing applications. The problem of controlling robots is part of the latter [31].

Reinforcement learning finds basis in the logical concept that if an action is followed by a satisfactory state of affairs, or by an improvement in the state of affairs (as established by a clearly defined measure), then the inclination to produce that action becomes stronger, that is, *reinforced*. Allowing actions to depend on state information, brings in the feedback aspect. Therefore, an RL learning system is one which results in an improvement in performance by interacting with its environment. Through interaction, the learning system receives feedback in the form of a scalar reward (or penalty)—a *reinforcement signal*, consistent with the response. No supervision is given to the learning system regarding what action is to be taken. Rather, trials lead to the discovery of the action which will give the most reward. The actions influence not just the *immediate reward* but also the situation that follows, and hence, all successive rewards. These two features—learning by trial-and-error and *cumulative reward*—are the two essential distinguishing traits of reinforcement learning. Even though the initial performance may be weak, through sufficient environmental

interaction it will ultimately be able to learn an effective strategy that maximizes the cumulative reward.

Suppose, we wish to build a machine that learns to play chess [6]. It is not possible for us to use a supervised learner as there is no best move; the appropriateness of a move is dependent on the moves that follow. A single move does not count but a *sequence* of moves is the best if the game is won after playing them (cumulative reward). There is a decision maker called the *agent* (the game-player, playing against an opponent) that is placed in an *environment* (the chess board). At any time, the environment is in a certain *state* (the state of the chess board), and the agent has a set of possible *actions* (legal moves of pieces on the chess board). Once an action is selected and performed, the state is altered. A *sequence of actions* is required for the solution to the task; the learning agent learns the best sequence of actions to solve a problem, where 'best' is quantified as a sequence possessing the maximum cumulative reward (winning the game). Feedback in the form of immediate reward following each action (move on the chess board) is used to select actions in order to ensure maximum cumulative reward.

Another example we take here is a robot (agent) placed in a maze (environment) [6]. The robot can move in one of the four compass directions without hitting the walls (set of possible actions). At any time, the robot is in a certain position in the maze, that is, one of the possible positions (environment is in a certain state that is one of the possible states). The robot performs a sequence of moves and when it reaches the exit, only then does it get a reward (cumulative reward). In this case, there is no opponent, but we can have preference for shorter trajectories, implying that in this case we play against time.

Reinforcement learning framework formulation to solve *sequential decision problems*, generally considers, the reinforcement learning problem as a straightforward framing of the problem of learning from interaction to attain a goal. The learner and the decision-maker is called *an agent*. Everything beyond that is known as the *environment*. There is continuous interaction amongst them, with the agent choosing the actions and the environment providing responses to them and presenting new situations (*states* of the environment) to the agent. Figure B.1 is a diagram of a generic agent, which perceives its environment via *sensors* and *acts* upon it using *effectors*. RL means learning to map states to actions so as to maximize a numerical *reward*. The agent is not

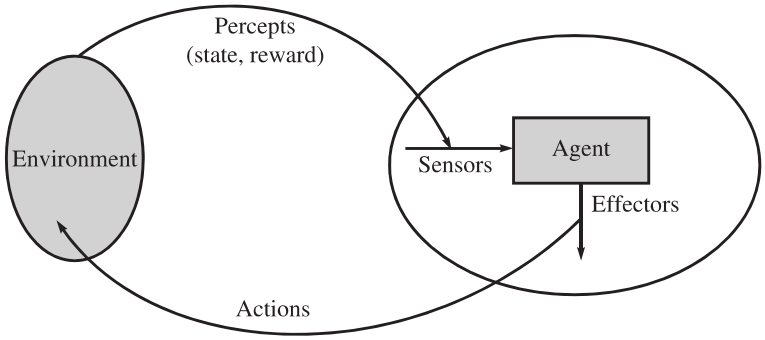


Figure B.1 A generic agent

informed regarding the actions to be taken; rather, it should discover the actions that give the maximum reward, by trying them out. To achieve huge rewards, an RL agent has to necessarily opt for actions that have been tried out earlier and found to be efficient in reward production. In order to find such actions, it has to test actions that have not been chosen before. The agent has to *exploit* the knowledge it already possesses by being *greedy* to maximize reward, but it has to also *explore* so as to choose better actions in the future. The problem is that exclusively exploring or exploiting will only lead to failure at the task. The agent has to attempt various actions and progressively opt for the ones that seem to be most efficient. Even if the agent's performance is not good enough initially, with adequate environmental interaction, it will ultimately learn an effective *policy* for reward maximization.

Reinforcement learning is not the same as supervised learning. It could be considered as "learning with a critic", which is different from "learning with a teacher". A *critic* is not the same as a teacher as it does not show us what is to be done. It only tells us how well we have done in the past; the critic gives no advance information. After several actions are taken and rewards received, it is desired to assess the individual actions performed earlier, and identify the moves that resulted in the winning of the reward, so that they can be recorded and recalled in the future. An RL program actually learns to generate a *value* for immediate states or actions; that is, how well they lead us to the goal. The moment an immediate reward mechanism of this kind is learned, the agent can perform the local actions to maximize it.

Until now we have assumed in the book that the instances that constitute a sample are *iid* (independently and identically drawn). This assumption is, however, not valid for applications where successive instances are dependent. Processes where there are a sequence of observations can't be modeled as simple probability distributions.

The problems which require a sequence of actions are ideally described in the *Markovian Decision Processes* (MDPs) framework of [6], wherein the sequence is characterized as being generated by a parametric random process. The states can be observed in an MDP model. At any time t we are aware of the state, and as the system progresses from one state to another, we obtain observation sequence, that is, a sequence of states and actions. In a *Hidden Markov Model* (HMM), the states cannot be observed.

In real-life applications, we usually come across two situations: observable states, and partially observable states. In certain applications, the agent is not aware of the exact state. It has the sensors that return an observation, which the agent then employs for an estimation of the state. Suppose there is a robot navigating in a room. It may be unaware of its precise location in the room, or what else exists in the room. The robot could have a camera for the purpose of recording sensory observations. This does not really convey to the robot its precise state, but provides some indication regarding its probable state. This setting resembles an MDP, except that once the action is taken, the new state is not known, but there is a sensor observation that is a stochastic function of earlier state and action. The solution to this *partially observable* MDP is somewhat similar to MDP: from sensor observation, the state (or rather the probability distribution for the states) can be inferred and then acted upon.

In our brief presentation in this appendix, we will assume observable states, and use MDP model to model the behavior of the agent.

The standard approach for solving MDPs is with the help of *dynamic programming*, which converts the problem of identifying a good agent into the problem of seeking a good *value function*. But, other than the simplest case when MDP has very few states and actions, dynamic programming is not feasible. The RL algorithms discussed here can be considered as a way of converting the infeasible dynamic programming techniques into feasible algorithms to make them applicable to large-scale problems. There are just two primary ideas that permit RL algorithms to attain this goal [31]:

- One primary idea is to employ *samples* that can represent the dynamics of the control system in a compact manner.
- Another key idea behind RL algorithms is to make use of powerful *function approximation* techniques (neural networks, for example) to efficiently represent value functions. Using neural networks helps control RL in terms of the realistic problems possessing large state- and action spaces. A neural network possesses the *generalization* property; experience with a restricted subset of state space is typically generalized to give rise to a good approximation over a much bigger subset.

The two ideas fit together properly. Samples may focus on a small subset of space they belong to, from which function approximation methods generalize to bigger spaces. *It is the understanding of the interplay between dynamic programming, samples, and function approximation, which is at the heart of the design, analysis, and application of RL algorithms.*

Recent advances relating reinforcement learning to dynamic programming are providing solid mathematical foundation; mathematical results that guarantee optimality in the limit for an important class of reinforcement learning systems are now available (the property that we lack in case of supervised learning, which is an empirical science—the asymptotic effectiveness of the learning systems has been validated only empirically).

In this appendix, we will concentrate on those RL algorithms that are built on the foundation of the powerful theory of dynamic programming. RL discussed here is also referred to as *neuro-dynamic programming* or *approximate dynamic programming*. The term neuro-dynamic programming is derived from the fact that, in several cases, RL algorithms are used with artificial neural networks.

There are several software packages which support the development and testing of RL algorithms. The most notable could be the RL-GLUE (http://glue.rl-community.org) and RL-LIBRARY (http://library.rl-community.org) packages.

The coverage of reinforcement learning in this appendix is to be regarded as an introduction to the subject; a springboard to advanced studies [30–32]. The inclusion of the topic has been motivated by the observation that reinforcement learning has the potential of solving many nonlinear control problems.

B.2 ELEMENTS OF REINFORCEMENT LEARNING

The learning decision maker is known as the *agent*. There is an interaction of the agent with the *environment*, which comprises everything beyond the agent. The agent is equipped with sensors that decide on its *state* in the environment and takes an *action* that modifies its state. When the agent takes an action, the environment offers a *reward* (Fig. B.2(a)). Time is discrete: $t = 0, 1, 2, \ldots$. When

the agent is in state s_t, takes an action a_t, the clock ticks, reward $r(s_t) \in \Re$ is received, and the agent moves to the next state s_{t+i}.

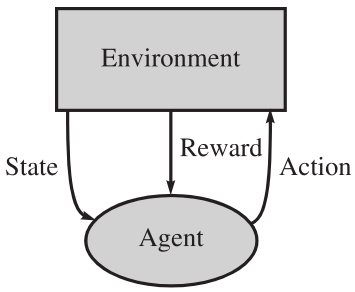
Beyond the agent and the environment, it is possible to find four main sub-elements of a reinforcement learning system—a *policy*, a *reward function*, a *value function* and *horizon* of decisions. A *policy* defines the learning agent's behavior at a given time. A *policy* is, roughly, a mapping of the environment from perceived states to actions that need to be implemented in those states. A *reward function* defines *immediate reward* for an action that is accountable for the present state of the environment. It maps environmental states to a scalar, a *reward*, which indicates the inherent desirability of the state. While a reward function implies whatever is good in the immediate sense, a *value function* lays down what is good in the long term. The *value* of a state is the *cumulative reward* that an agent can expect to receive in the future as an outcome of the sequence of its actions, beginning from that state. While rewards establish the immediate, intrinsic desirability of environment states, values imply the desirability of states in the long run, after considering the states that are expected to follow, and the rewards obtainable in those states. An agent's only aim is the maximization of the cumulative reward (value) attained in the long term.

The value function is dependent on the existence of a *finite* or an *infinite horizon* for making decisions. A finite horizon indicates a *fixed* time after which nothing really matters, as the game is kind of finished. In case of a finite horizon, the optimal action for a given state may be different at different times, that is optimal policy for a finite horizon is *nonstationary*.

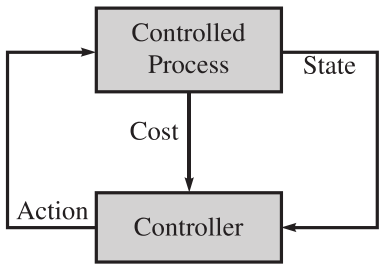
In the absence of a fixed time limit, on the contrary, the reason for any different behavior in the same state at different times does not arise. Therefore, the optimal action is dependent solely on the present state, and the policy is *stationary*. Policies for finite-horizon case are complex, whereas policies for the infinite-horizon are much simpler.

'Infinite horizon' does not imply that all state sequences are infinite; it merely means that no deadline is fixed. There will not be any infinite sequences if the environment comprises *terminal states* and if it is known with certainty that the agent will eventually reach one.

Our focus in this chapter is on reinforcement learning solutions to control problems (Fig. B.2(b)). The controller (agent) has a set of sensors to observe the state of the controlled process (environment); the learning task is to learn a control strategy (policy) for choosing control signals (actions) that achieve minimization of a performance measure (maximization of cumulative reward).



(a) Agent-environment interaction



(b) The basic reinforcement learning scenario for control problems

Figure B.2

In control problems, we minimize a performance measure; frequently referred to as *cost function*. The reinforcement learning control solution seeks to minimize the long-term accumulated cost the controller incurs over the task time. The general reinforcement learning solution seeks to maximize the long-term accumulated reward the agent receives over the task time. Since in control problems, reference of optimality is a cost function, we assign *cost* to the reward structure of the reinforcement learning process; *the reinforcement learning solution then seeks to minimize the long-term accumulated cost the agent incurs over the task time*. The value function of the reinforcement learning process is accordingly defined with respect to cost structure.

The stabilizing control problems are all infinite-horizon problems. Here also, we will limit our discussion to this class of control problems.

Some reinforcement learning systems have one more element—a *model* of the environment. This replicates the behavior of the environment. For instance, considering a state and action, the model may predict the subsequent next state and next cost.

RL systems were initially clearly model-free, trial-and-error learners. Over time, it became quite clear that RL techniques are similar to dynamic programming techniques, which make use of models. *Adaptive dynamic programming* has emerged as a solution method for reinforcement learning problems wherein the agent learns the models through trial-and-error interaction with the environment, and then uses these models in dynamic programming methods.

In control engineering, vector **x** is used to represent the *state* of a physical system: $\mathbf{x} = [x_1\ x_2 \cdots x_n]^T$, where x_j: $j = 1, \ldots, n$, are *state variables* of the system. State **x**, a vector of real numbers, is a point in the state space. In reinforcement learning (RL) framework, we will represent the state by '*s*'; thus *s* is a point in the *n*-dimensional state space. Similarly, the vector **u** is used for control. We will represent this by the action '*a*' in our RL framework.

If the environment is *deterministic*, then an agent's action *a* will transit the state of the environment from *s* to *s*′ deterministically; there is no probability involved. If the environment is *stochastic*, then transition of *s* to *s*′ under action *a* will be different each time action *a* is applied in state *s*. This is captured by a probabilistic model. If the environment is deterministic, but uncertain, then also transition of *s* to *s*′ under action *a* will not be unique each time action *a* is applied is state *s*. Since uncertainty in environments is the major issue leading to complexity of the control problem, we will be concerned with probabilistic models.

(1) A specification of the outcome probabilities for each admissible action in each possible state is known as the *transition model*.

$P(s, a, s')$: probability of reaching state s' if action a is applied in state s.

(2) There are *Markovian* transitions in control problems. That is, the probability of reaching state *s*′ from *s* depends only on *s* and not on the history of earlier states.
(3) In each state *s*, the agent gets a *reinforcement* $r(s)$, which measures the immediate cost of the action.
(4) A *Markov Decision Process* (MDP) is the specification of a sequential decision problem for a completely observable environment, with a Markovian transition model and cost for each state
(5) The RL framework is based on the Markov decision processes.

Off-line Learning and On-line Learning

Dynamic programming is a general-purpose technique used to identify optimal control strategies for nonlinear and stochastic dynamic systems. It addresses the problem of designing closed-loop policies off-line, assuming that a precise model of the stochastic dynamic system is available. The off-line process of design generally results in a computationally efficient technique which determines each action as a function of the observed system state.

There are two practical issues related to the use of dynamic programming:

(1) For many real-world problems, the number of possible states and admissible actions in each state are so large that the computational requirements of dynamic programming are overwhelming ('curse of dimensionality').
(2) Dynamic programming algorithms require accurate model of the dynamic system; this prior knowledge is not always available ('curse of modeling').

Over the past three decades, the focus of researchers has been to develop methods capable of finding high-quality approximate solutions to problems where exact solutions via classic dynamic programming are not attainable in practice due to high computational complexity and lack of accurate knowledge of system dynamics. In fact, *reinforcement learning* is a field that represents this stream of activities. All of the reinforcement learning can be viewed as attempts to achieve the same effect as dynamic programming, only with less computation and without assuming a perfect model of the dynamic system. By focusing on computational effort along behavioral patterns of interactions with the environment, and by using function approximation (neural network) for generalization of experience to states not reached through interactions, reinforcement learning can be used no-line for problems with large state spaces and with lack of accurate knowledge of system dynamics.

RL and use of dynamic programming to arrive at solutions for sequential decision problems are closely related as follows:

(i) the environment in both is characterized by a set of states, a set of admissible actions, and a cost function;
(ii) both aim to identify a decision policy that reduces the cumulative cost over time to a minimum.

However, there is a significant difference. While solving a sequential decision problem with the help of dynamic programming, the agent (apparently the designer of the control system) has a complete (albeit stochastic) model of the environmental behavior. With this information, the agent can calculate the optimal control policy pertaining to the model. In RL, the set of states, and the set of admissible actions are known *a priori*. However, the effects of actions on the environment and on the cost are unknown. Therefore, the agent cannot compute an optimal policy *a priori* (off-line). Rather, the agent should learn an optimal policy by experimenting in the environment. Therefore, RL system is an on-line system.

In an on-line learning system, the learner moves around in a real environment observing the outcomes. In such a situation, the main concern is generally the number of real-world actions that the agent should perform to converge to a computational agreeable policy (instead of the number of cycles, as in off-line learning). The reason is that for many practical problems, the costs in time and in dollars of performing actions dominate the computational costs.

An *adaptive dynamic programming* agent learns the transition model of the environment by interacting with the environment. It then plugs the transition model and the observed costs in the dynamic programming algorithms. Adaptive dynamic programming is, therefore, an on-line learning system.

The process of learning the model itself is not difficult when it is possible to fully observe the environment. In the most simple case, we can represent the transition model as a table of probabilities. We track the frequency of each action outcome, and make an estimation of the transition probability $P(s, a, s')$ from the frequency with which state s' is attained during the execution of the action a in state s.

Temporal Difference Learning: An idea that is central and novel to RL, is undoubtedly *Temporal Difference* (TD) learning. Temporal difference learning can be considered as a version of dynamic programming, with the only difference that TD techniques can learn on-line in real-time, from raw experience without a model of the environment's dynamics. TD techniques do not make an assumption of complete knowledge of the environment; they need only *experience*—sample sequences of states, actions and costs from actual environmental interaction. Learning from *actual* experience is outstanding as it needs no prior knowledge of the environment's dynamics, but still can attain optimal behavior.

The main benefit of dynamic programming is that, if a problem can be specified in terms of Markov decision process, then its analysis can be done and an optimal policy achieved *a priori*. The two primary drawbacks of dynamic programming are as follows:

(i) for various tasks, it is not easy to specify the dynamic model; and
(ii) as dynamic programming establishes a fixed control policy *a priori*, it fails to offer a mechanism for adapting the policy to compensate for disturbances and/or modeling errors (nonstationary dynamics).

Reinforcement learning has complimentary benefits as follows:

(i) it needs no prior dynamical model of any type, but learns by experience gathered directly from the environment; and
(ii) to a certain level, it can keep track of the dynamics of nonstationary systems.

The primary drawback of RL is that, typically, several trials (repeated experiences) are needed to learn an optimal control strategy, particularly if the system begins with a weak initial policy.

It appears that the respective drawbacks of these two approaches can be taken care of through their integration. That is, if a complete, possibly incorrect, model to the task is available *a priori*, model-based techniques (including dynamic programming) may be employed for the development of initial policy for an RL system. A reasonable initial policy can substantially improve the system's initial performance and reduce the time required to reach an acceptable level of performance. Alternatively, if an adaptive RL element is added to an otherwise model-based fixed controller, an inaccurate model can be compensated for.

In this appendix, we limit our discussion to *naive* reinforcement learning systems. Our focus is on temporal difference learning. We begin with an introduction to dynamic programming, and then using this platform, develop temporal difference methods of learning.

B.3 BASICS OF DYNAMIC PROGRAMMING

We first define a general formulation of the problem of learning sequential control strategies. To do so, we consider building a learning controller for stabilization of an inverted pendulum. Figure B.3 shows an inverted pendulum with its pivot mounted on a cart. The cart is driven by an electric motor. The motor drives a pair of wheels of the cart; the whole cart and the pendulum become the 'load' on the motor. The motor at time t exerts a torque $T(t)$ on the wheels. The linear force applied to the cart is $u(t)$; $T(t) = Ru(t)$, where R is the radius of the wheels.

The pendulum is obviously unstable. It can, however, be kept upright by applying a proper control force $u(t)$. This somewhat artificial system example represents a dynamic model of a space booster on take off—the booster is balanced on top of the rocket engine thrust vector.

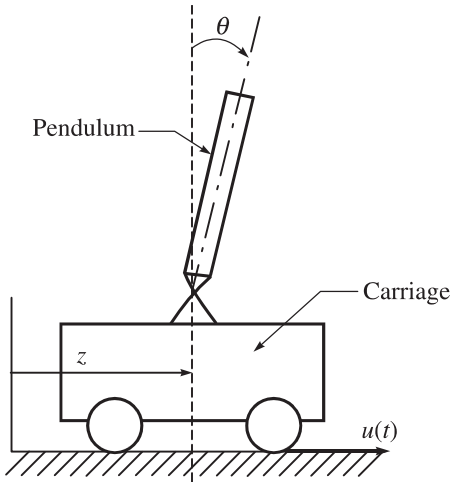


Figure B.3 Inverted pendulum system

In the *reinforcement learning control* setting, the controller, or *agent*, has a set of sensors to observe the *state* of its *environment* (the dynamic system: inverted pendulum mounted on a cart). For example, a controller may have sensors to measure angular position θ and velocity $\dot{\theta}$ of the pendulum, and horizontal position z and velocity $\dot{z}$ of the cart; and actions implemented by applying a force of u newtons to the cart. Its task is to learn control strategy, or *policy*, for choosing actions that achieve its goals.

A common way of obtaining approximate solutions for continuous state and action tasks is to quantize the state and action spaces, and apply finite-state dynamic programming (DP) methods. The methods we explore later in this appendix, make learning tractable on the realistic control problems with continuous state spaces (infinitely large set of quantized states).

Suppose that our stabilization problem demands that the pendulum must be kept within ± 12° from vertical, and the cart must be kept within ± 2.4m from the center of the track.

We define the following finite sets of possible states S and available actions A.

States →	1	2	3	4	5	6
Pend. angle(deg); θ	< – 6	–6 to –1	–1 to 0	0 to 1	1 to 6	> 6
Pend. velocity; $\dot{\theta}$	< – 50	–50 to 50	> 50			
Cart position(m); z	< – 0.8	–0.8 to 0.8	> 0.8			
Cart velocity; $\dot{z}$	< – 0.5	– 0.5 to 0.5	> 0.5			

Actions →	1	2	3	4	5	6	7
Apply force of u newtons	–10	–6	–2	0	2	6	10

Define: $x_1, = \theta$, $x_2, = \dot{\theta}$, $x_3 = z$, $x_4 = \dot{z}$. Vector $\mathbf{x} = [x_1\ x_2\ x_3\ x_4]^T$ defines a point in the state space; the distinct point corresponding to $\mathbf{x}$ is the distinct state s of the environment (pendulum on a cart). Therefore, there are $6 \times 3 \times 3 \times 3 = 162$ distinct states: $s^{(1)}, s^{(2)}, \ldots, s^{(162)}$, of our environment. The finite set of states, in our learning problem, is thus given as,

$$S : \{s^{(1)}, s^{(2)}, \ldots, s^{(162)}\}$$

The action set size is seven: $a^{(1)}, a^{(2)}, \ldots, a^{(7)}$. The finite set of available actions in our learning problem, is thus given as,

$$A : \{a^{(1)}, a^{(2)}, \ldots, a^{(7)}\}$$

We assume the knowledge of *state transition model*:

$P(s, a, s')$: probability of reaching state s' if action a is applied in state s; for all $s \in S$, and for all $a \in A$

Note that our model is stochastic; it captures the uncertainties involved in the environment.

In each state s, the agent receives a *reinforcement* $r(s)$, which measures the immediate *cost* of action. For the particular inverted pendulum example, a cost of '–1' may be assigned to *failure states* ($\theta > 12°$; $\theta > -12°$), and a cost of '0' may be assigned to every other state. Note that cost structure for a learning problem is an important design parameter. It controls the convergence speed of a learning algorithm. The functions $P(\cdot)$ and $r(\cdot)$ are part of the environment.

The specification of a sequential design problem for a fully observable (the agent knows where it is) environment with a Markovian decision model and cost for each state, is a *Markov Decision Process* (MDP). An MDP is defined by the tuple (S, A, P, r) where S is the set of possible states the environment can occupy; A is the set of admissible actions the agent may execute to change the state of the environment, P is the state transition probability, and r is the cost function. We assume that

$$S : \{s^{(1)}, s^{(2)}, \ldots, s^{(N)}\}; A : \{a^{(1)}, a^{(2)}, \ldots, a^{(M)}\}$$

where N represents the total number of distinct states of the environment, and M represents the total number of admissible actions in each state.

Let us now consider the structure of solution to the problem. Any fixed action sequence (open-loop structure) will not solve the problem because due to uncertainties in the behavior of the environment, the agent might end up in a failure state; i.e., the scheme lacks the robustness properties. Therefore,

a solution must specify what the agent should do far *any* state that the environment might reach. The resulting feedback loop is a source of a measure of internal/external disturbances. A solution of this kind is called a *policy*. We usually denote a policy by π.

A stationary policy π for an MDP is a mapping $\pi\colon S \to \Omega(A)$, where $\Omega(A)$ is the set of all probability distributions over A. $\pi(a, s)$ stands for the probability that policy π chooses action a in state s. Since each action $a^{(1)}, a^{(2)}, \ldots, a^{(M)}$ is a candidate for state s, policy $\pi(a, s)$ for s is a set of action-selection probabilities associated with $a^{(1)}, \ldots, a^{(\mathrm{M})}$; their sum equals one.

A stationary deterministic policy π is a policy that commits to a single action choice per state, that is, a mapping $\pi\colon S \to A$ from states to actions. In this case, $\pi(s)$ *indicates the action that the agent takes in state s*. For every MDP, there exists an *optimal deterministic policy*, which minimizes the *expected*, *total discounted cost* (to be defined shortly) from any initial state. It is therefore, sufficient to restrict the search for the optimal policy only within the space of deterministic policies.

The next question we must decide is how to calculate the *value of a state*. Recall that the value of a state is the *cumulative cost* an agent can expect to incur over the future as a result of sequence of its actions, starting from that state. A sequence of actions for a given task will force the environment through a sequence of states. Let us call it *environment trajectory* of a given task. In an infinite-horizon problem, the number of actions for a task is not fixed; therefore, number of distinct states in an environment trajectory is not fixed. A typical state sequence in a trajectory may be expressed as $\{s_0, s_1, s_2, \ldots\}$ where, each s_t; $t = 0, 1, 2, 3, \ldots$, could be any of the possible environment states $s^{(1)}, \ldots, s^{(N)}$.

Given the initial state s_t and the agent's policy π, the agent selects an action $\pi(s_t)$, and the result of this action is next state s_{t+1}. The state transition model, $P(s, a, s')$, gives a probability that the next state s_{t+1} will be $s' \in S$, given that the current state $s_t = s$ and the action $a_t = a$. Since each state $s^{(1)}, s^{(2)}, \ldots, s^{(N)}$ is a candidate to be the next state s', the environment simulator gives a set of probabilities: $P(s_t, a_t, s^{(1)}), \ldots, P(s_t, a_t, s^{N})$; their sum equals one. Thus, a given policy π generates not one state sequence (environment trajectory), but a whole range of possible state sequences, each with a specific probability determined by the transition model of the environment.

The quality of a policy is, therefore, measured by the *expected value* (cumulative cost) of a state, where the expectation is taken over all possible state sequences that could occur. For MDPs, we can define the 'value of a state under policy π' formally as,

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{t=0}^{\infty} \gamma^t r(s_t)\right\} \tag{B.1}$$

where $E_{\pi}\{\cdot\}$ denotes the expected value given that the agent follows policy π. This is a *discounted cost* value function; the *discount factor* γ is a number between 0 and 1 ($0 \leq \gamma < 1$).

Note that

$$\sum_{t=0}^{\infty} \gamma^t r(s_t) \leq \sum_{t=0}^{\infty} \gamma^t r_{\max} = r_{\max}/(1-\gamma)$$

Thus, the infinite sequence converges to a finite limit when costs are bounded and $\gamma < 1$.

The discount factor γ determines the relative value of delayed versus immediate costs. In particular, costs incurred t steps into the future are discounted exponentially by a factor of γ^t. Note that if we set $\gamma = 0$, only the immediate cost is considered. If we set γ closer to 1, future costs are given greater emphasis relative to the immediate cost. The meaning of γ substantially less than 1 is that future costs matter to us less than the costs paid at this present time. The discount factor is an important design parameter in reinforcement learning scheme.

The final step is to show how to choose between policies. An *optimal policy* is a policy that yields the lowest expected value. We use π^* to denote an optimal policy.

$$\pi^* = \arg\min_{\pi} E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right] \tag{B.2}$$

The 'arg min' notation denotes the values of π at which $E_{\pi}[\cdot]$ is minimized. $\pi^*(s)$ is, thus, a solution (obtained off-line) to the sequential decision problem. Given π^*, the agent decides what to do in real time by observing the current state s and executing the action $\pi^*(s)$. This is the simplest kind of agent, selecting fixed actions on the basis of the current state. A reinforcement learning agent, as we shall see shortly, is *adaptive*; it improves its policy on the basis of on-line, real-time interactions with the environment.

In the following, we describe algorithms for finding optimal policies of the dynamic programming agent.

B.3.1 Finding Optimal Policies

The dynamic programming technique rests on a very simple idea known as the *principle of optimality* [32].

An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the previous decisions.

Consider a state sequence (environment trajectory) resulting from the execution of optimal policy π^*: $\{s_0, s_1, s_2, \ldots\}$ where each s_t: $t = 0, 1, 2, \ldots$, could be any of the possible environment states $s^{(1)}, s^{(2)}, \ldots, s^{(N)}$. The index t represents *stages of decisions* in the sequential decision problem.

The dynamic programming algorithm expresses a generalization of the principle of optimality. It states that *the optimal value of a state is the immediate cost for that state plus the expected discounted optimal value of the next state, assuming that the agent chooses the optimal action*. That is, the optimal value of a state is given by

$$V^*(s) = r(s) + \gamma \min_{a} \sum_{s'} P(s, a, s')V^*(s') \tag{B.3}$$

This is one form of the *Bellman optimality equation* for V^*. For finite MDPs, this equation has a unique solution.

The Bellman optimality equation is actually a system of N simultaneous *nonlinear* equations in N unknowns, where N is the number of possible environment states. If the dynamics of the environment ($P(s, a, s')$) and the immediate costs underlying the decision process ($r(s)$) are known, then, in principle, one can solve this system of equations for V^*using any one of the variety

of methods for solving systems of nonlinear equations. Once one has V^*, it is relatively easy to determine an optimal policy:

$$\pi^*(s) = \arg\min_a \sum_{s'} P(s, a, s')V^*(s') \tag{B.4}$$

Note that $V^*(s) = V^{\pi^*}(s)$:

$$V^*(s) = \min_\pi V^\pi(s) \text{ for all } s \in S \tag{B.5}$$

The solution of Bellman optimality equation (B.3) directly gives the values V^* of states with respect to optimal policy π^*. From this solution, one can obtain optimal policy using Eqn (B.4).

Equation (B.5) suggests an alternative route to finding optimal policy π^*. It uses *Bellman equation for* V^π, given below.

$$V^\pi(s) = r(s) + \gamma \sum_{s'} P(s, \pi(s), s')\, V^\pi(s') \tag{B.6}$$

Note that this equation is a system of N simultaneous *linear* equations in N unknowns, where N is the number of possible environment states (Eqns (B.6) are same as Eqns (B.3) with 'min' operator removed). We can solve these equations for $V^\pi(s)$ by standard linear algebra methods.

Given an initial policy π_0, one can solve (B.6) for $V^{\pi_0}(s)$. Once we have V^{π_0}, we can obtain improved policy π_1, using the strategy given by Eqn (B.4):

$$\pi_1(s) = \arg\min_a \sum_{s'} P(s, a, s')V^{\pi_0}(s') \tag{B.7}$$

The process is continued:

$$\pi_0 \to V^{\pi_0} \to \pi_1 \to V^{\pi_1} \to \pi_2 \to \cdots \to \pi^* \to V^*$$

It is certain that each policy will be a strict improvement over the previous one (unless it is already optimal). As a finite MDP has only a limited number of policies, this procedure will have to end up as an optimal policy π^* and optimal value function V^* in a limited number of iterations.

Thus, given a complete and accurate model of MDP in the form of knowledge of the state transition probabilities $P(s, a, s')$ and immediate costs $r(s)$ for all states $s \in S$ and all actions $a \in A$, it is possible—at least in principle—to solve the decision problem off-line. There is one problem: the Bellman Eqn (B.3) is nonlinear because of the 'min' operator; solution of nonlinear equations is problematic. The Bellman Eqn (B.6) is linear and therefore, can be solved relatively quickly. For large state spaces, time might be prohibitive even in this relatively simpler case.

In the following, we describe basic forms of two dynamic programming algorithms: *value iteration* and *policy iteration*—a step towards answering the computational complexity problems of solving Bellman equations.

B.3.2 Value Iteration

As employed for solving Markov decision problems, *value iteration* is a successive approximation process that solves the Bellman optimality Eqn (B.3), where the fundamental operation is 'backing

up' estimates of optimal state values. We can solve Eqn (B.3) with the help of a simple iterative algorithm:

$$V_{(l+1)}(s) \leftarrow r(s) + \gamma \min_{a} \sum_{s'} P(s, a, s') V_l(s') \tag{B.8}$$

The algorithm begins with arbitrary guess $V_0(s)$ for each $s \in S$. The sequence of $V_1(s)$, $V_2(s)$, …, is then obtained. The algorithm converges to the optimal values $V^*(s)$ as the number of iterations l approaches infinity (We use the index l for the stages of iteration algorithm, whereas we have used earlier the index t to denote the stages of decisions in the sequential decision problem). In practice, we stop once the value function changes by a small amount. Then a *greedy policy* (choosing the action with the lowest estimated cost) with respect to the optimal set of values is obtained as an optimal policy.

The computation (B.8) is done off-line, i.e., before the real system starts operating. An optimal policy, that is, an optimal choice of $a \in A$ for each $s \in S$, is computed either simultaneously with V^*, or in real time, using Eqn (B.4).

A sequential implementation of iteration algorithm (B.8) requires temporary storage locations so that all the iteration-$(l + 1)$ values are computed based on the iteration-l values. The optimal values V^* are then stored in a *lookup table*. In addition to a problem of the memory needed for large tables, there is another problem of time needed to accurately fill them. Suppose there are N states, and M is the largest number of admissible actions for any state, then each iteration comprising backing up the value of each state precisely once, needs about $M \times N^2$ operations. For the huge state sets, typically seen in control problems, it is tough to attempt to complete even one iteration, leave alone iterate the procedure till it ends up in V^* (curse of dimensionality).

The iteration of *synchronous* DP algorithm defined in (B.8) *backs up* the value of every state once to produce the new approximate value function. We call this kind of operation as *full backup*; it is based on all possible next states rather than on a sample next state. We think of the backups as being done in a *sweep* through the state space.

Asynchronous DP algorithms are not organized in terms of systematic sweep of the entire set of states in each iteration. These algorithms back up the values of the states in any random order, with the help of whatever values of other states are available. The values of certain states may be backed up many times before the values of others are backed up once. To converge accurately, however, an asynchronous algorithm will have to continue to back up the values of all the states.

Of course, evading sweeps need not imply that we can get away with less computation. It simply means that our algorithm does not require to be locked into any hopelessly long sweep before it can progress. We can attempt to leverage this flexibility by choosing the states to which backups are applied to improve the algorithm's progress rate. We can attempt to order the backups to let value information propagate efficiently from one state to another. Some states may not require the backing up of their values as often as other states. Some state orderings give rise to faster convergence as compared to others, depending on the problem.

B.3.3 Policy Iteration

A *policy iteration algorithm* operates by alternating between two steps (the algorithm begins with arbitrary initial policy π_0):

(i) *Policy evaluation step*

Given the current policy π_k, we perform policy evaluation step that computes $V^{\pi_k}(s)$ for all $s \in S$, as the solution of the linear system of equations (Bellman equation)

$$V^{\pi_k}(s) = r(s) + \gamma \sum_{s'} P(s, \pi_k(s), s') V^{\pi_k}(s') \tag{B.9}$$

in the N unknowns $V^{\pi_k}(s)$.

To solve these equations, an iteration procedure similar to the one used in value iteration algorithm (given by (B.8)) may be used.

$$V_{l+1}^{\pi_k}(s) \leftarrow r(s) + \gamma \sum_{s'} P(s, \pi_k(s), s') V_l^{\pi_k}(s') \tag{B.10}$$

(ii) *Policy improvement step*

Once we have V^{π_k}, we can obtain improved policy π_{k+1} (refer to Eqn (B.7)) as follows:

$$\pi_{k+1}(s) = \arg\min_a \sum_{s'} P(s, a, s') V^{\pi_k}(s') \tag{B.11}$$

The two-step procedure is repeated with policy π_{k+1} used in place of π_k, unless we have $V^{\pi_{k+1}}(s) \approx V^{\pi_k}(s)$ for all s; in which case, the algorithm is terminated with optimal policy $\pi^* = \pi_k$.

Policy iteration algorithm can be viewed as an *actor-critic system* (Fig. B.4). In this interpretation, the policy evaluation step is viewed as the work of a *critic*, who evaluates the performance of the current policy π_k, i.e., generates an estimate of the value function V^{π_k} from states and reinforcement supplied by the environment as inputs. The policy improvement step is viewed as the work of an *actor*, who takes into account the latest evaluation of the critic, i.e., the estimate of the value function, and acts out the improved policy π_{k+1}.

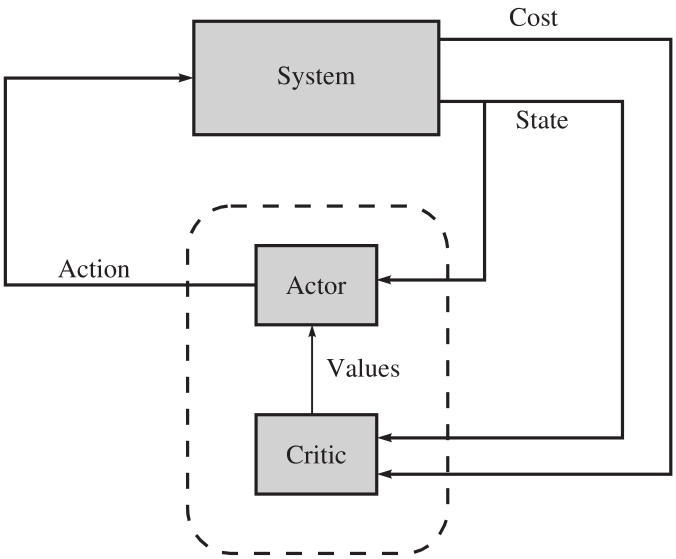


Figure B.4 The actor-critic architecture

The algorithm described till now needs updation of the values/policy for all states at once. But this is not strictly required. In fact, it is possible for us to pick any subset of states, on each iteration, and carry out updating to that subset. This algorithm is known as *asynchronous policy iteration*. Considering specific conditions on the initial policy and value function, asynchronous policy iteration is certain to converge to an optimal policy. The liberty to select any states to work on implies that it is possible to design highly effective heuristic algorithms—for instance, algorithms focussing on updation of the values of states which have a likelihood of being reached by a good policy.

B.4 TEMPORAL DIFFERENCE LEARNING

The novel aspect of learning that we address now is that it assumes the agent that does *not* have knowledge of $r(s)$ and $P(s, a, s')$, and therefore it cannot learn solely by simulating actions with environment model (off-line learning not possible). It has no choice but to interact with the environment and learn by observing consequences.

Figure B.5 gives a general setting of the agent-environment interaction process. Time advances by discrete unit length quanta; $t = 0, 1, 2, \ldots$. At each time step t, the agent senses the current state $s_t \in S$ of the environment, chooses an action $a_t \in A$, and performs it. The environment responds by giving the agent a cost $r_t = r(s_t)$, and by producing the succeeding state $s_{t+1} \in S$.

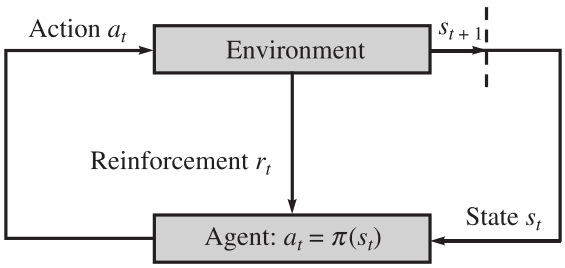


Figure B.5 The agent-environment interaction

The environment is stochastic in nature—each time the action a_t is applied in the state s_t, the succeeding state s_{t+1} could be any of the possible states in $S : s^{(1)}, s^{(2)}, \ldots, s^{(N)}$. For the stochastic environment, the agent, however, explores in the space of deterministic policies (a deterministic optimal policy is known to exist for Markov decision process). Therefore, for each observed environment state s_t, the agent's policy suggests a deterministic action $a_t = \pi(s_t)$.

The task of the agent is to learn a policy $\pi : S \rightarrow A$ that produces the lowest possible cumulative cost over time (*greedy policy*). To state this requirement more precisely, the agent's task is to learn a policy π that minimizes the value V^{π} given by (B.1).

Reinforcement learning methods specify how the agent updates its policy as a result of its experience. The agent could use alternative methods for gaining experience and using it for improvement of its policy. In the so called *Monte Carlo* method, the agent executes a set of *trials* in the environment using its current policy π. In each trial, the agent starts in state $s^{(i)}$ (any point $s^{(1)}, \ldots, s^{(N)}$ of state space) and experiences a sequence of state transitions until it reaches a terminal state.

In infinite-horizon discounted cost problems under consideration, terminal state corresponds to the *equilibrium state*. A learning episode (trial) is infinitely long, because the learning is continual. For the purpose of viewing the *infinite-horizon* problem in terms of *episodic learning*, we may define a stability region around the equilibrium point and say that the environment has terminated at a *success state* if the state continues to be in stability region for a prespecified time period. [In a real-time control, any uncertainty (internal or external) will pull the system out of stability region and a new learning episode begins.] Failure states (situations corresponding to 'the game is over and it is lost') if any, are also terminal states of the learning process.

In a learning episode, agent's precepts supply both the current state and the cost incurred in that state. Typical state sequences (environment trajectories) resulting from trials might look like this:

(1) $(s^{(1)})_{r(1)} \rightarrow (s^{(5)})_{r(5)} \rightarrow (s^{(9)})_{r(9)} \rightarrow (s^{(5)})_{r(5)} \rightarrow (s^{(9)})_{r(9)} \rightarrow (s^{(10)})_{r(10)} \rightarrow$
$\rightarrow (s^{(11)})_{r(11)} \rightarrow (s^{(\text{SUCCESS})})_{r(\text{SUCCESS})}$

(2) $(s^{(1)})_{r(1)} \rightarrow (s^{(5)})_{r(5)} \rightarrow (s^{(9)})_{r(9)} \rightarrow (s^{(10)})_{r(10)} \rightarrow (s^{(11)})_{r(11)} \rightarrow$
$\rightarrow (s^{(7)})_{r(7)} \rightarrow (s^{(11)})_{r(11)} \rightarrow (s^{(\text{SUCCESS})})_{r(\text{SUCCESS})}$

(3) $(s^{(1)})_{r(1)} \rightarrow (s^{(2)})_{r(2)} \rightarrow (s^{(3)})_{r(3)} \rightarrow (s^{(7)})_{r(7)} \rightarrow (s^{(\text{FAILURE})})_{r(\text{FAILURE})}$

Note that each state percept is subscripted with the cost incurred. The objective is to use the information about costs to learn the expected value $V^{\pi}(s)$ associated with each state. The value is defined to be the expected sum of (discounted) costs incurred if policy π is followed (refer to Eqn (B.2)).

When a nonterminal state is visited, its value is estimated based on what happens after that visit. Thus, the value of a state is the expected total cost from that state onward, and each trial (episode) provides *samples* of the value for each state visited. For example, the first trial in the set of three given above, provides one sample of value for state $s^{(1)}$:

(i) $r(1) + \gamma r(5) + \gamma^2 r(9) + \gamma^3 r(5) + \gamma^4 r(9) + \gamma^5 r(10) + \gamma^6 r(11) + \gamma^7 r(\text{SUCCESS})$;

two samples of values for state $s^{(5)}$:

(i) $r(5) + \gamma r(9) + \gamma^2 r(5) + \gamma^3 r(9) + \gamma^4 r(10) + \gamma^5 r(11) + \gamma^6 r(\text{SUCCESS})$;

(ii) $r(5) + \gamma r(9) + \gamma^2 r(10) + \gamma^3 r(11) + \gamma^4 r(\text{SUCCESS})$;

two samples of values for state $s^{(9)}$:

(i) $r(9) + \gamma r(5) + \gamma^2 r(9) + \gamma^3 r(10) + \gamma^4 r(11) + \gamma^5 r(\text{SUCCESS})$;

(ii) $r(9) + \gamma r(10) + \gamma^2 r(11) + \gamma^3 r(\text{SUCCESS})$;

and so on.

Therefore, at the end of each episode, the algorithm computes the observed total cost for each state visited, and the estimated value is accordingly updated for that state simply by maintaining a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation of Eqn (B.2).

The Monte Carlo method differs from dynamic programming in the following two ways:

(i) First, it operates on *sample experience*, and thus can be used for direct learning without a model.
(ii) Second, it does not build its value estimates for a state on the basis of estimates of the possible successor states (refer to Eqn (B.6)); it must wait until the end of the trial to determine the update in value estimates of states. In dynamic programming methods, the value of each state equals its own cost plus the discounted expected value of its successor states.

The *Temporal Difference* (TD) learning techniques are a combination of sampling of Monte Carlo, and the value estimation scheme of dynamic programming. TD techniques update value estimates on the basis of the cost of one-step real-time transition and learned estimate of successor state, without waiting for the final outcome. Typically, when a transition occurs from state s to state s', we apply the flowing update to $V^{\pi}(s)$:

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \eta \left(r(s) + \gamma V^{\pi}(s') - V^{\pi}(s)\right) \tag{B.12}$$

where η is the learning parameter.

Because the update uses the difference in values between successive states, it is called the temporal-difference or TD equation. TD methods have an advantage over dynamic programming methods in that they do not require a model of the environment. Advantage of TD methods over Monte Carlo is that they are naturally implemented in an on-line fully incremental fashion. With Monte Carlo methods, one must wait until the end of a sequence, because only then is the value known, whereas with TD methods, one need only wait one time step.

Note that the update (B.12) is based on *one* state transition that just happens with a certain probability, whereas in (B.6), the value function is updated for all states simultaneously using all possible next states, weighted by their probabilities. This difference disappears when the effects of TD adjustments are averaged over a large number of transitions. The interaction with the environment can be repeated several times by restarting the experiment after success/failure state is reached. For one particular state, the next state and received reinforcement can be different each time the state is visited. As the frequency of each successor in the set of transitions is approximately proportional to its probability, TD can be considered as a crude but effective approximation to dynamic programming.

The TD Eqn (B.12) is, in fact, approximation of policy-evaluation step of policy iteration algorithm of dynamic programming (refer to previous section for a recall), where the agent's policy is fixed and the task is to learn the values of states. This, as we have seen, can be done without a model of the system. However, improving the policy using (B.11) still requires the model.

One of the most significant breakthroughs in RL was the development of model-free TD control algorithm, called Q-learning.

B.4.1 Q-learning

In addition to recognizing the inherent relationship between RL and dynamic programming, Watkins [30, 32] made a significant contribution to RL by suggesting a new algorithm named *Q-learning*. The importance of Q-learning is that when applied to a Markov decision process, it can be shown

to converge to the optimal policy, under appropriate conditions. Q-learning is the first RL algorithm seen to be convergent to the optimal policy for decision problems which involve cumulative cost.

The Q-learning technique learns an *action-value* representation rather than learning value function. We will make use of the notation $Q(s,a)$ *to denote the value of doing action a in state s*. Q-function is directly related to value function as follows:

$$V(s) = \min_a Q(s, a) \tag{B.13}$$

Q-functions may seem like just another way of storing value information, but they have a very imported property: *a TD agent that learns a Q-function does not need a model for either learning or action selection*. For this reason, Q-learning is called a *model-free* method.

The connections between Q-learning and dynamic programming are strong: Q-learning is motivated directly by value-iteration, and its convergence proof is based on a generalization of the convergence proof for value-iteration.

We can use the value-iteration algorithm (B.8) directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model:

$$Q_{l+1}(s, a) \leftarrow r(s) + \gamma \sum_{s'} P(s, a, s') \left[\min_{a'} Q_l(s', a') \right] \tag{B.14}$$

It converges to the optimal Q-values, $Q^*(s, a)$.

Once one has $Q^*(s,a)$ for all $s \in S$ and all $a \in A$, it is relatively easy to determine an optimal policy:

$$\pi^*(s) = \arg\min_a Q^*(s, a) \tag{B.15}$$

This does, however, require that a model is given [or is learned (adaptive dynamic programming)], because Eqn (B.14) uses $P(s, a, s')$.

The temporal-difference approach, on the other hand, requires no model. The update equation for TD Q-learning is (refer to Eqn (B.12))

$$Q(s, a) \leftarrow Q(s, a) + \eta \left(r(s) + \gamma \left[\min_{a'} Q(s', a') \right] - Q(s, a) \right) \tag{B.16}$$

which is calculated whenever action a is executed in state s leading to s'.

The Q-learning algorithm (B.16) takes a back-up of the Q-value for just a single state-action pair at each time step of control, where the state-action pair comprises the observed present state and the action actually executed. Particularly, make an assumption that at time step t in real-time control, the agent observes state s_t and has the estimated Q-values created by all the preceding stages of real-time Q-learning (the estimates stored in a *lookup table* with one entry for each state-action pair). We depict these estimates by $Q_t(s, a)$ for all admissible state-action pairs. The agent selects an action $a_t \in A$ using this information available in lookup table:

$$a_t = \arg\min_a Q_t(s_t, a)$$

After executing a_t, the agent receives the immediate cost $r_t = r(s_t)$ while the environment state changes to s_{t+1}. The Q-values in the lookup table are then updated as follows:
For the state-action pair (s_t, a_t):

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \eta_t\left[r_t + \gamma\left(\min_a Q_t(s_{t+1}, a)\right) - Q_t(s_t, a_t)\right] \tag{B.17a}$$

For other admissible state-action pairs, the Q-values remain unchanged:

$$Q_{t+1}(s, a) = Q_t(s, a)\ \forall\,(s, a) \neq (s_t, a_t) \tag{B.17b}$$

Watkins [30, 32] has shown the Q-learning system that

(1) decreases its learning parameter at an appropriate rate (e.g., $\eta_t = 1/t^{\beta}$, where $0.5 < \beta < 1$); and
(2) visits each state-action pair infinitely often, is guaranteed to converge to an optimal policy.

Convergence, thus requires that *the agent selects actions in such a fashion that it visits every possible state-action pair infinitely enough*. This means, if action a is an admissible action from state s, then over a period of time the agent will have to execute action a from state s again and again with nonzero frequency as the length of its action sequence approaches infinity.

In the Q-learning algorithm shown in Eqns (B.17), the strategy for the agent in state s_t at time step t is to choose the action a that minimizes $Q_t(s_t, a)$, and thus, exploits the present approximation of Q^* by following a *greedy policy*. But, with this strategy, the agent stands the risk that it will overcommit to actions discovered with low Q-values during early stages, while failing to *explore* other actions that can have even lower values. In fact, the convergence condition needs each state-action transition to take place infinitely often. This will not take place if the agent follows the greedy policy all the time.

The Q-learning agent should necessarily follow the policy of *exploring* and *exploiting*: the former, exploration, makes sure that all admissible state-action pairs are visited enough to satisfy the Q-learning convergence condition, and the latter, exploitation, tries to minimize the cost by adopting a greedy policy.

Many exploration schemes have been employed in the RL literature. While the most basic one is to behave greedily most of the time, but every once a while, with small probability ε, instead choose a random action, uniformly, and independent of the action-value estimates. We call techniques that make use of this near-greedy action-selection rule *ε-greedy techniques*.

B.4.2 Generalization

Till now the assumption has been that the Q-values the agent learns are represented in a tabular form with a single entry for each state-action pair. This is a clear and instructive case, but it is certainly restricted to tasks possessing small numbers of states and actions. The issue is not merely the memory required for large tables, but the computational time required to experience all the state-action pairs to generate data to fill the tables in an accurate manner.

Very few decision and control problems in the real world fit into *lookup table representation* strategy for solution; the number of possible states and actions in the real world is often much too large to accommodate the computational and storage requirements. The problem is more severe *when state/action spaces include continuous variables*—to use a table, they should be discretized

to finite size, which may cause errors. The only way of learning anything about these tasks is to *generalize* from past experienced states, to states which have never been seen. Simply put, experience with a finite subset of state space can be meaningfully generalized to give rise to a good approximation over a bigger subset.

Luckily, generalization from examples has been studied at length, and we do not require to devise altogether novel techniques to use in Q-learning. To a great extent, we require to simply create a combination of Q-learning and off-the-shelf architectures for the purpose of inductive generalization—frequently referred to as *function approximation*, as it takes examples from desired Q-function and tries to generalize from them for the construction of an approximation of the function. Function approximation is done with the help of *supervised learning*. In principle, any of the techniques examined in this field can be employed in Q-learning.

In parametric methods, the tabular (exact) representation of the real-valued functions $Q(s,a)$ is replaced by a generic parametric function approximation $\hat{Q}(s,a;\mathbf{w})$ where $\mathbf{w}$ are the adjustable parameters of the approximator. Learning $Q(s,a)$ for all $s \in S$ and $a \in A$ amounts to learning parameters $\mathbf{w}$ of $\hat{Q}(s,a;\mathbf{w})$. The new version of Q-learning Eqn (B.16) is

$$w_j \leftarrow w_j + \eta\left(r(s) + \gamma\left[\min_{a'} Q(s',a';\mathbf{w})\right] - Q(s,a;\mathbf{w})\right)\frac{\partial Q(s,a,\mathbf{w})}{\partial w_j} \tag{B.18}$$

This update rule can be shown to converge to the closest possible approximation to the true function when the function approximator is *linear* in the parameters.

Unfortunately, all bets are off when *nonlinear* function approximators—such as neural networks—are used. For many tasks, Q-learning fails to converge once a nonlinear function approximator is introduced. Fortunately, however, the algorithm does converge for large number of applications. The theory of Q-learning with nonlinear function approximator still contains many open questions; at present, it remains an empirical science.

For Q-learning, it makes more sense to use an incremental learning algorithm that updates the parameters of function approximator after each trial. Alternatively, examples may be collected to form a training set and leaned in batch mode, but it slows down learning as no learning happens while a sufficiently large sample is being collected.

We give an example of *neural Q-learning*. Let $\hat{Q}_t(s,a;\mathbf{w})$ denote the approximation to $Q_t(s,a)$ for all admissible state-action pairs, computed by means of a neural network at time step t. The state s is input to the neural network with parameter vector $\mathbf{w}$ producing the output $\hat{Q}_t(s,a;\mathbf{w})$ $\forall\ a \in A$. We assume that the agent uses the training rule of (B.17) after initialization of $\hat{Q}(s,a;\mathbf{w})$ with arbitrary finite values of $\mathbf{w}$.

Treating the expression inside the square bracket in (B.17a) as the error signal involved in updating the current value of parameter vector $\mathbf{w}$, we may identify the target (desired) value of $\hat{Q}_t$ at time step t as,

$$Q_t^{\text{target}}(s_t, a_t; \mathbf{w}) = r_t + \gamma\left(\min_a Q_t(s_{t+1}, a; \mathbf{w})\right) \tag{B.19}$$

At each iteration of the algorithm, the weight vector $\mathbf{w}$ of the neural network is changed slightly in a way that brings the output $\hat{Q}_t(s_t, a_t; \mathbf{w})$ closer to the target $Q_t^{\text{target}}(s_t, a_t; \mathbf{w})$ for the current (s_t, a_t) pair. For other state-action pairs, Q-values remain unchanged (Eqn (B.17b)).

B.4.3 Sarsa-learning

The Q-learning algorithm is an *off-policy* TD method: the learned action-value function Q directly approximates Q^*, the optimal action-value function, independent of the policy being followed; optimal action for state s is then obtained from Q^*. The Q-learning is motivated by value iteration algorithm in dynamic programming.

The alternative approach, motivated by policy iteration algorithm in dynamic programming, is an *on-policy* TD method. The distinguishing feature of this method is that it attempts to evaluate and improve the same policy that it uses to make decisions.

Earlier is this section on TD learning, we considered transitions from state to state and learned the value of states (Eqn (B.12)) when following a policy π. The relationship between states and state-action pairs is symmetrical. Now we consider transition from state-action pair to state-action pair and learn the value of state-action pairs, following a policy π. In particular, for on-policy TD method, we must estimate $Q^\pi(s, a)$ for the current policy π and for all states $s \in S$ and actions $a \in A$. We can learn Q^π using essentially the same TD method used in Eqn (B.12) for learning V^π:

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \eta(r(s) + \gamma Q^\pi(s', a') - Q^\pi(s, a)) \tag{B.20}$$

where a' is the action executed in state s'.

This rule uses every element of the quintuple of events, (s, a, r, s', a'), that make up a transition from one state-action pair to the next. This quintuple (State-Action-Reinforcement-State-Action) gives rise to the name SARSA for this algorithm. Unlike Q-learning, here the agent's policy does matter. Once we have $Q^{\pi_k}(s, a)$, improved policy can be obtained as follows:

$$\pi_{k+1}(s) = \arg\min_a Q^{\pi_k}(s, a) \tag{B.21}$$

Since tabular (exact) representation is impractical for large state and action spaces, function approximation methods are used. Approximations in the policy-iteration framework can be introduced at the following two places:

(i) *The representation of the Q-function*: The tabular representation of the real-valued function $Q^\pi(s, a)$ is replaced by a generic parametric function approximation $\hat{Q}^\pi(s, a; \mathbf{w})$ when $\mathbf{w}$ are the adjustable parameters of the approximator.

(ii) *The representation of the policy*: The tabular representation of the policy $\pi(s)$ is replaced by a parametric representation $\hat{\pi}(s; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the adjustable parameters of the representation.

The difficulty involved in use of these approximate methods within policy iteration is that the off-the-shelf architectures and parameter adjustment methods cannot be applied blindly; they have to be fully integrated into the policy-iteration framework.

The introduction to the subject of reinforcement learning given in this appendix had the objective of motivating the readers for advanced studies [30–32]. Reinforcement learning has been successfully applied to many real-life decision problems. Vast amount of literature is available on the subject. Going through the available case studies, and experimenting with some, will provide a transition to the next improved state of learning.