A Tutorial for Reinforcement Learning

Abhijit Gosavi
Department of Engineering Management and Systems Engineering
Missouri University of Science and Technology
210 Engineering Management, Rolla, MO 65409
Email:gosavia@mst.edu

September 30, 2019

If you find this tutorial or the codes in C and MATLAB (weblink provided below) useful, please do cite my book (for which this material was prepared), now in its second edition:

A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, Springer, New York, NY, Second edition, 2014.

Book website: http://simoptim.com
Codes: http://simoptim.com/bookcodes.html

Contents

1 Introduction

The tutorial is written for those who would like an introduction to reinforcement learning (RL). The aim is to provide an intuitive presentation of the ideas rather than concentrate on the deeper mathematics underlying the topic.

RL is generally used to solve the so-called Markov decision problem (MDP). In other words, the problem that you are attempting to solve with RL should be an MDP or its variant. The theory of RL relies on dynamic programming (DP) and artificial intelligence (AI). We will begin with a quick description of MDPs. We will discuss what we mean by "complex" and "large-scale" MDPs. Then we will explain why RL is needed to solve complex and large-scale MDPs. The semi-Markov decision problem (SMDP) will also be covered.

The tutorial is meant to serve as an *introduction* to these topics and is based mostly on the book: "Simulation-based optimization: Parametric Optimization techniques and reinforcement learning" [4]. The book discusses this topic in greater detail in the context of simulators. There are at least two other textbooks that I would recommend you to read: (i) Neuro-dynamic programming [2] (lots of details on convergence analysis) and (ii) Reinforcement Learning: An Introduction [11] (lots of details on underlying AI concepts). A more recent tutorial on this topic is [8]. This tutorial has 2 sections:

- Section 2 discusses MDPs and SMDPs.
- Section 3 discusses RL.

By the end of this tutorial, you should be able to

- Identify problem structures that can be set up as MDPs / SMDPs.
- Use some RL algorithms.

We will not discuss how to use function approximation, but will provide some general advice towards the end.

2 MDPs and SMDPs

The framework of the MDP has the following elements: (1) state of the system, (2) actions, (3) transition probabilities, (4) transition rewards, (5) a policy, and (6) a performance metric. We assume that the system is modeled by a so-called abstract stochastic process called the Markov chain. When we observe the system, we observe its Markov chain, which is defined by the states. We explain these ideas in more detail below.

State: The "state" of a system is a parameter or a set of parameters that can be used to describe a system. For example the geographical coordinates of a robot can be used to describe its "state." A system whose state changes with time is called a *dynamic* system. Then it is not hard to see why a moving robot produces a dynamic system.

Another example of a dynamic system is the queue that forms in a supermarket in front of the counter. Imagine that the state of the queuing system is defined by the number of people in the queue. Then, it should be clear that the state fluctuates with time, and then this is dynamic system.

It is to be understood that the transition from one state to another in an MDP is usually a random affair. Consider a queue in which there is one server and one waiting line. In this queue, the state x, defined by the number of people in the queue, transitions to $(x+1)$ with some probability and to $(x-1)$ with the remaining probability. The former type of transition occurs when a new customer arrives, while the latter event occurs when one customer departs from the system because of service completion.

Actions: Now, usually, the motion of the robot can be controlled, and in fact we are interested in controlling it in an optimal manner. Assume that the robot can move in discrete steps, and that after every step the robot takes, it can go North, go South, go East, or go West. These four options are called *actions* or *controls* allowed for the robot.

For the queuing system discussed above, an action could be as follows: when the number of customers in a line exceeds some prefixed number, (say 10), the remaining customers are diverted to a new counter that is opened. Hence, two actions for this system can be described as: (1) Open a new counter (2) Do not open a new counter.

Transition Probability: Assume that action a is selected in state i. Let the next state be j. Let $p(i,a,j)$ denote the probability of going from state i to state j under the influence of action a in one step. This quantity is also called the transition probability. If an MDP has 3 states and 2 actions, there are 9 transition probabilities per action.

Immediate Rewards: Usually, the system receives an immediate reward (which could be positive or negative) when it transitions from one state to another. This is denoted by $r(i,a,j)$.

Policy: The policy defines the action to be chosen in every state visited by the system. Note that in some states, no actions are to be chosen. States in which decisions are to be made, i.e., actions are to be chosen, are called *decision-making* states. In this tutorial, by states, we will mean decision-making states.

Performance Metric: Associated with any given policy, there exists a so-called performance metric — with which the performance of the policy is judged. Our goal is to select the policy that has the best performance metric. We will first consider the metric called the average reward (or revenues) of a policy. We will later discuss the metric called discounted reward (or revenues). We will assume that the system is run for a long time and that we are interested in a metric measured over what is called the infinite time horizon.

Time of transition: We will assume for the MDP that the time of transition is unity (1), which means it is the *same* for every transition. Hence clearly 1 here does not have to mean 1 hour or minute or second. It is some fixed quantity fixed by the analyst. For the SMDP, this quantity is *not* fixed as we will see later.

Let us assume that a policy named π is to be followed. Then $\pi(i)$ will denote the action selected by this policy for state i. Every time there is a jump in the Markov chain (of the system under the policy in consideration), we say a transition (or jump) has occurred. *It is* *important to understand that during a transition we may go from a state to itself!*

Let x_s denote the state of the system before the sth transition. Note that in a so-called infinite horizon problem, s will go from 1 to infinity. Then, the following quantity, in which $x_1 = i$, is called the average reward of the policy π if one starts at state i.

$$\rho_i = \lim_{k\to\infty} \frac{\mathsf{E}\left[\sum_{s=1}^{k} r(x_s, \pi(x_s), x_{s+1}) | x_1 = i\right]}{k} \tag{1}$$

This average reward essentially denotes the sum of the total immediate rewards earned divided by the number of jumps (transitions), calculated over a very long time horizon (that is k assumes a large value.) In the above, the starting state is i and $\pi(x_s)$ denotes the action in state x_s. Also note that $\mathsf{E}[.]$ denotes the average value of the quantity inside the square brackets.

It is not hard to show that the limit in (1) is such that its value is the *same* for any value of x_1, if the underlying Markov chains in the system satisfy certain conditions (related to the regularity of the Markov chains); in many real-world problems such conditions are often satisfied. Then

$$\rho_i = \rho \text{ for any value of } i.$$

The objective of the average-reward MDP is to find the policy that maximizes the performance metric (average reward) of the policy.

Another popular performance metric, commonly used in the literature, is discounted reward. The following quantity is called the discounted reward of a policy π. Again, let x_s denote the state of the system before the sth jump (transition). The total discounted reward of the policy π, when one starts at state i, is given as:

$$\psi_i = \lim_{k\to\infty} \mathsf{E}\left[\sum_{s=1}^{k} \gamma^{s-1} r(x_s, \pi(x_s), x_{s+1}) \middle| x_1 = i\right], \tag{2}$$

where γ denotes the discount factor; this factor, γ, is less than 1 but greater than 0. Eqn. (2) has a simple interpretation:

$$\psi_i = \mathsf{E}[r(x_1, \pi(x_1), x_2) + \gamma r(x_2, \pi(x_2), x_3) + \gamma^2 r(x_3, \pi(x_3), x_4) + \cdots]$$

The discounted reward essentially measures the present value of the sum of the rewards earned in the future over an infinite time horizon, where γ is used to discount money's value. We should point out that:

$$\gamma = \left(\frac{1}{1+\mu}\right)^1, \tag{3}$$

where μ is the rate of interest; the rate is expressed as a fraction here and not in percent. When $\mu > 0$, we have that $0 < \gamma < 1$. It is worthwhile pointing out that in the MDP, we have $1/(1+\mu)$ raised to the power 1 because in the MDP, we assume a fixed rate of discounting and that the time duration of each transition is fixed at 1. This mechanism thus captures within the MDP framework the notion of time value of money.

The objective of the discounted-reward MDP is to find the policy that maximizes the performance metric (discounted reward) of the policy starting from every state.

Note that for average reward problems, the immediate reward in our algorithms can be earned during the entire duration of the transition. However, for the discounted reward problems, we will assume that the immediate reward is earned immediately after the transition starts.

The MDP can be solved with the classical method of dynamic programming (DP). However, DP needs all the transition probabilities (the $p(i,a,j)$ terms) and the transition rewards (the $r(i,a,j)$ terms) of the MDP.

For Semi-Markov decision problems (SMDPs), an additional parameter of interest is the time spent in each transition. The time spent in transition from state i to state j under the influence of action a is denoted by $t(i,a,j)$. To solve SMDPs via DP, one also needs the transition times (the $t(i,a,j)$ terms). For SMDPs, the average reward that we seek to maximize is defined as:

$$\rho_i = \lim_{k\to\infty} \frac{\mathsf{E}\left[\sum_{s=1}^{k} r(x_s,\pi(x_s),x_{s+1})|x_1 = i\right]}{\mathsf{E}\left[\sum_{s=1}^{k} t(x_s,\pi(x_s),x_{s+1})|x_1 = i\right]}. \tag{4}$$

(Technically, lim should be replaced by lim inf everywhere in this tutorial, but we will not worry about such technicalities here.) It can be shown that the quantity has the same limit for any starting state (under certain conditions). A possible unit for average reward here is \$ per hour.

For discounted reward, we will, as stated above, assume the immediate reward is earned *immediately* after the transition starts and does not depend on the duration of the transition. Thus, the immediate reward is a lumpsum reward earned at the start of the transition (when the immediate reward is a function of the time interval, see instead the algorithm in [3]). Also, because of the variability in time, we will assume continuously compounded rate of interest. Then, we seek to maximize:

$$\psi_i = \lim_{k\to\infty} \mathsf{E}\left[r(x_1,\pi(x_1),x_2) + \sum_{s=2}^{k} r(x_s,\pi(x_s),x_{s+1}) \int_{\tau_s}^{\tau_{s+1}} e^{-\mu\tau} d\tau \middle| x_1 = i \right],$$

where $e^{-\mu\tau}$ denotes the discount factor over a period of length τ (under continuous compounding) and τ_s is the time of occurrence of the sth jump (transition). Note that since the immediate reward does not depend on the time τ, it can be taken out of the integral. Essentially, what we have above is the sum of discounted rewards with the discount factor in each transition appropriately calculated using the notion of continuous compounding.

Curses: For systems which have a large number of input random variables, it is often hard to derive the exact values of the associated transition probabilities. This is called the *curse of modeling*. For large-scale systems with millions of states, it is impractical to store these values. This is called the *curse of dimensionality.*

DP breaks down on problems which suffer from any one of these curses because it needs all these values.

Reinforcement Learning (RL) can generate near-optimal solutions to large and complex MDPs and SMDPs. In other words, RL is able to make inroads into problems which suffer from one or more of these two curses and cannot be solved by DP.

3 Reinforcement Learning

We will describe a basic RL algorithm that can be used for average reward SMDPs. Note that if $t(i,a,j) = 1$ for all values of i, j, and a, we have an MDP. Hence our presentation will be for an SMDP, but it can easily be translated into that of an MDP by setting $t(i,a,j) = 1$ in the steps.

It is also important to understand that the transition probabilities and rewards of the system are not needed if any one of the following is true:

1. we can play around in the real world system choosing actions and observing the rewards

2. if we have a simulator of the system.

The simulator of the system can usually be written on the basis of the knowledge of some other easily accessible parameters. For example, the queue can be simulated with the knowledge of the distribution functions of the inter-arrival time and the service time. Thus the transition probabilities of the system are usually **not** required for writing the simulation program.

Also, it is important to know that the RL algorithm that we will describe below requires the updating of certain quantities (called Q-factors) in its database whenever the system visits a new state.

When the simulator is written in C or in any special package such as ARENA, it is possible to update certain quantities that the algorithm needs whenever a new state is visited.

Usually, the updating that we will need has to be performed immediately after a new state is visited. In the simulator, or in real time, it IS possible to keep track of the state of the system so that when it changes, one can update the relevant quantities.

The key idea in RL is store a so-called Q-factor for each state-action pair in the system. Thus, $Q(i,a)$ will denote the Q-factor for state i and action a. The values of these Q-factors are initialized to suitable numbers in the beginning (e.g., zero or some small number to all the Q-factors). Then the system is simulated (or controlled in real time) using the algorithm. In each state visited, some action is selected and the system is allowed to transition to the next state. The immediate reward (and the transition time) that is generated in the transition is recorded as the feedback. The feedback is used to update the Q-factor for the action selected in the previous state. Roughly speaking if the feedback is good, the Q-factor of that particular action and the state in which the action was selected is increased (rewarded) using the Relaxed-SMART algorithm. If the feedback is poor, the Q-factor is punished by reducing its value.

Then the same reward-punishment policy is carried out in the next state. This is done for a large number of transitions. At the end of this phase, also called the learning phase, the action whose Q-factor has the highest value is declared to be the optimal action for that state. Thus the optimal policy is determined. Note that this strategy does not require the transition probabilities.

4 Average Reward RL

We begin with average reward RL. We will describe two algorithms: R-SMART (Relaxed Semi-Markov Average reward Technique) from Gosavi [6] and BAC (Bounded Actor Critic) from Lawhead and Gosavi [10].

4.1 R-SMART

The original version of R-SMART appeared in [6]. It is presented below for the SMDP, but can be converted to an MDP format by setting $t(.,.,.) = 1$ everywhere in the algorithm's steps. Below, we present a modified version from [5] that has better convergence properties in practice.

Steps in R-SMART:

The steps in the Learning Phase are given below.

- Step 0 **(Inputs)**: Set the Q-factors to some arbitrary values (e.g., 0), that is:

  $$Q(i,a) \leftarrow 0 \text{ for all } i \text{ and } a.$$

  Set the iteration count, k, to 1. Let ρ^k denote the average reward in the kth iteration of the algorithm. Set ρ^1 to 0 or any other value. Let the first state be i. Let $\mathcal{A}(i)$ denote the set of actions allowed in state i. Let α^k and β^k denote two learning rates that we will need. How these values should be selected and updated is discussed in the next subsection. Set the total_reward and total_time to zero. Set $ITERMAX$, which will denote the number of iterations for which the algorithm is run, to a large number. Set η, a scaling constant needed in the algorithm, to a small positive value close to 1 but less than 1, e.g., 0.99.

- Step 1 **(Q-factor Update)**: Determine the action associated to the Q-factor that has the highest value in state i. (For instance, if there are two actions in a state i and their values are $Q(i,a) = 19$ and $Q(i,2) = 45$, then, clearly, action 2 has the greatest Q-factor.) This is called the **greedy** action. Select the greedy action with probability $(1 - \mathsf{p}(k))$. One common approach to defining $\mathsf{p}(k)$ is as follows:

  $$\mathsf{p}(k) = \frac{G_1}{G_2 + k}$$

  in which $G_2 > G_1$, and G_1 and G_2 are large positive constants, e.g., 1000 and 2000 respectively. With a probability of $\mathsf{p}(k)$, choose one of the other actions. (For the two-action case, you can generate a random number between 0 and 1. If the number is less than or equal to $(1 - \mathsf{p}^k)$, choose the greedy action; otherwise, choose the other action.) The non-greedy actions are called **exploratory** actions, and selecting an exploratory action is called exploration. Our probability of exploration will decay with k, the number of iterations.

Let the action selected be denoted by a. If a is a greedy action, set $\phi = 0$; otherwise, set $\phi = 1$. Simulate action a. Let the next state be denoted by j. Let $r(i, a, j)$ denote the immediate transition reward and $t(i, a, j)$ denote the immediate transition time. Then update $Q(i, a)$ as follows:

$$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) - \rho^k t(i,a,j) + \eta \max_{b \in \mathcal{A}(j)} Q(j,b) \right].$$

In the above, $\max_{b \in \mathcal{A}(j)} Q(j, b)$ equals the maximum numeric value of all the Q-factors of state j.

- Step 2 **(Average Reward Update)**: If $\phi = 1$, i.e., the action a was non-greedy, go to Step 3. Otherwise, update total_reward and total_time as follows.

$$\begin{aligned} \text{total\_reward} &\leftarrow \text{total\_reward} \; + r(i,a,j), \\ \text{total\_time} &\leftarrow \text{total\_time} \; + t(i,a,j). \end{aligned}$$

  Then update the average reward as:

$$\rho^{k+1} \leftarrow (1-\beta^k)\rho^k + \beta^k \left[\frac{\text{total\_reward}}{\text{total\_time}} \right].$$

- Step 3 **(Check for Termination)**: Increment k by 1. Set $i \leftarrow j$. If $k < ITERMAX$, return to Step 1. Otherwise, go to Step 4.

- Step 4 (**Outputs**): declare the action for which $Q(i, .)$ is maximum to be the optimal action for state i (do this for all values of i, i.e., for all states to generate a policy), and STOP.

In the above, the exploration probability $\mathsf{p}(k)$ gradually decays to 1, and in the limit, the algorithm selects the greedy actions. This decaying of the probability has to be gradual. If the decay is very quick (e.g., you use small values for G_1 and G_2), the algorithm will most likely converge to an incorrect solution. In other words, the algorithm should be allowed to *explore* sufficiently.

Further note: η has to satisfy $0 < \eta < 1$. The positive scalar η must be less than 1; it enables the algorithm to converge gracefully to the optimal solution. Its value should be as close to 1 as possible and should not be changed during the learning. An example could be $\eta = 0.99$. In practice, $\eta = 1$ may also generate convergent behavior, but there is no known mathematical proof of this.

The next phase is called the *frozen* phase because the Q-factors are *not* updated in it. This phase is performed to estimate the average reward of the policy declared by the frozen phase to be the optimal policy. (By optimal, of course, we only mean the best that RL can generate; it may not necessarily be optimal, but hopefully is close enough to the optimal in its performance.) Steps in the Frozen Phase are as follows.

- Step 0 **(Inputs)**: Use the Q-factors learned in the Learning Phase. Set iteration count k to 0. $ITERMAX$ will denote the number of iterations for which the frozen phase is run. It should be set to a large number. Also set the following two quantities to 0: total_reward and total_time.

- Step 1 **(Simulation)**: Select for state i the action which has the maximum Q-factor. Let that action be denoted by a. Simulate action b. Let the next state be denoted by j. Let $r(i,a,j)$ denote the immediate transition reward and $t(i,a,j)$ denote the immediate transition time. Then update total_reward and total_time as follows.

$$\text{total\_reward} \leftarrow \text{total\_reward} + r(i,a,j);$$

$$\text{total\_time} \leftarrow \text{total\_time} + t(i,a,j).$$

- Step 2 **(Check for Termination)**: Increment k by 1. Set $i \leftarrow j$. If $k < ITERMAX$, return to Step 1. Otherwise, go to Step 3.

- Step 3 **(Output)** Calculate the average reward of the policy learned in the learning phase as follows:

$$\rho = \frac{\text{total\_reward}}{\text{total\_time}}.$$

The value of ρ^k at the end of the learning phase can also be used as an estimate of the actual ρ while the learning is on. But typically a frozen phase is carried out to get a cleaner estimate of the actual average reward of the policy learned.

Selecting the appropriate learning rate or step size The learning rates (α^k and β^k) should be positive values typically less than 1. The learning rate should also be a function of k. The learning rates typically have to satisfy key conditions rules described in [4]. Common examples of step-sizes are:

$$\alpha^k = A/(B+k)$$

where for instance $A = 90$ and $B = 100$. Another rule is the log rule, which is $\alpha^k = log(k+1)/(k+1)$ for k starting at 1 (see [7]). The simplest rule $\alpha^k = 1/k$, where $A = 1$ and $B = 0$ in the above, may not lead to good behavior (see [7] for evidence).

In R-SMART, we have two step sizes, α^k and β^k, within the same algorithm. For R-SMART, we must make sure that β^k converges to 0 faster than α^k. One example that satisfies this condition is $\alpha^k = log(k)/k$ and $\beta^k = 90/(100+k)$. Ideally, as k tends to ∞, β^k/α^k should ideally tend to zero.

4.2 Bounded Actor Critic

These algorithms use two sets of values, rather than the one set of Q-values used in R-SMART: the P-factors that are also called actor values and the V-factors that are also called the critic values. (The critic values are really the value function of dynamic programming, but you don't need to worry about this at first.) The steps for the MDP can be obtained by setting $t(.,.,.) = 1$ everywhere, like in the case of R-SMART. We present the learning phase below. Also, note that a frozen phase following the learning phase is needed here as well to determine the best value of the average reward.

Steps in Bounded Actor Critic for Average Reward [10]:

- **Inputs**: Initialize all actor, $P(.,.)$, and critic, $V(.)$, values to zero. Set k, the number of iterations, to 0. Set the scalars, R, T, and ρ, to zero. Set η to a positive value very close to 1 but strictly less than 1. Let $k_{\max}$ denote the maximum number of iterations for which the algorithm is run.
- Loop until $k = k_{\max}$
  - Let i be the current state. Select action a with probability of
    $$q(i,a) = \frac{e^{P(i,a)}}{\sum_{b \in \mathcal{A}(i)} e^{P(i,b)}}. \tag{5}$$
    Let j be the next state. Let $r(i,a,j)$ denote the immediate reward in the state transition and $t(i,a,j)$ denote the time taken in the transition.
  - Actor's update:
    $$P(i,a) \leftarrow (1-\alpha)P(i,a) + \alpha\left[r(i,a,j) - \rho t(i,a,j) + \eta V(j) - V(i)\right]. \tag{6}$$
  - Critic's update:
    $$V(i) \leftarrow (1-\beta)V(i) + \beta\left[r(i,a,j) - \rho t(i,a,j) + \eta V(j)\right]. \tag{7}$$
  - Average Reward Update: Update R, T, and ρ as follows:
    $$R \leftarrow R + r(i,a,j); \quad T \leftarrow T + t(i,a,j); \quad \rho = (1-\delta)\rho + \delta[R/T]. \tag{8}$$
  - Set $k \leftarrow k+1$. If $k = k_{max}$, exit loop; otherwise, set $i \leftarrow j$ and continue within loop.
- **Outputs**: The policy delivered by the algorithm is computed as follows. Declare the action for which $P(i,.)$ is maximum to be the optimal action for state i (do this for all values of i).

The action-selection scheme used here (see Equation (12)) is called the Boltzmann scheme and very importantly does not need the temperature, U, which is necessary in the classical actor-critic (shown below). Also, this version keeps the actor values bounded, unlike the classical actor-critic, which makes the actor unbounded and hence needs a temperature in the action-selection scheme. The classical version of the actor-update for average reward based on the update in [1] (which was for discounted reward) and is also presented in [9] would be as follows:

$$P(i,a) \leftarrow P(i,a) + \alpha\left[r(i,a,j) - \rho t(i,a,j) + \eta V(j) - V(i)\right]. \tag{9}$$

Note that there is a difference between the above update and the one in the algorithm above, Eqn. (13). The difference is that in the BAC algorithm, we use a multiplier of $(1-\alpha)$ to

$P(i,a)$, which along with η keeps the actor's values bounded. That multiplier is not used in the classical actor critic, i.e., Eqn. (9). The temperature, U, needed in the classical version employs the following Boltzmann scheme:

$$q(i,a) = \frac{e^{U \times P(i,a)}}{\sum_{b \in \mathcal{A}(i)} e^{U \times P(i,b)}}, \tag{10}$$

where U is set to satisfy $0 < U < 1$, such that even if $P(.,.)$ becomes large, the action-selection scheme works. Another approach to use the classical algorithm is to use an awkward projection of $P(.,.)$ to keep it artificially bounded; see Lawhead and Gosavi [10] for more details.

5 Discounted Reward

For discounted reward, the learning phase is sufficient. The steps we first describe are from the famous Q-Learning algorithm of Watkins [13]. They apply to the MDP; we will discuss the SMDP extension later. We will follow this with a discussion on the BAC (Bounded Actor Critic) for discounted reward.

5.1 Q-Learning

- Step 0 **(Inputs)**: Set the Q-factors to 0:

  $$Q(i,a) \leftarrow 0 \text{ for all } i \text{ and } a.$$

  Let $\mathcal{A}(i)$ denote the set of actions allowed in state i. Let α^k denote the main learning rate in the kth iteration. Set $k = 1$. $ITERMAX$ will denote the number of iterations for which the algorithm is run and should be set to a large number.

- Step 1 **(Q-factor Update)**: Select an action a in state i with probability $1/|\mathcal{A}(i)|$, where $|\mathcal{A}(i)|$ denotes the number of elements in the set $\mathcal{A}(i)$. Simulate action a. Let the next state be denoted by j. Let $r(i,a,j)$ denote the immediate transition reward. Update $Q(i,a)$ as follows:

  $$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) + \gamma \max_{b \in \mathcal{A}(j)} Q(j,b) \right],$$

  where you should compute α^k using one of the rules discussed above. Further, γ here denotes the discount factor.

- Step 2 **(Termination Check)**: Increment k by 1. Set $i \leftarrow j$ If $k < ITERMAX$, return to Step 1. Otherwise, go to Step 3.

- Step 3 **(Outputs)**: for each state i, declare the action for which $Q(i,.)$ is maximum to be the optimal action.

Note that in the algorithm above, there is no decay of exploration. This is because, in Q-Learning, the exploration need not decay. However, in practice, to get the algorithm to converge faster, decay of exploration if often employed. We do not recommended it in simulators, unless there is a need to obtain a solution quickly. In online applications (such as in robotics), decay may be needed.

For the SMDP extension, you can use the following update in Step 2 of the above algorithm (see [4] for more details):

$$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) + e^{-\mu t(i,a,j)} \max_{b \in \mathcal{A}(j)} Q(j,b) \right], \tag{11}$$

where the exponential term arises as follows:

$$\gamma^\tau = \left(\frac{1}{1+\mu} \right)^\tau \approx e^{-\mu\tau}.$$

in which τ denotes the time and the discounting mechanism was explained above in Equation (3). Note that in the above approximation to obtain the exponential term, we use the fact that μ is quite small.

Note on the vanishing discount approach: You can actually use a discounted RL algorithm to solve the average reward problem via the vanishing discount approach. In this heuristic approach, one uses a discounted reward algorithm with γ very close to 1 (for MDPs) and μ very close to 0 (for SMDPs). This can work very well in practice.

5.2 Bounded Actor Critic

We present the MDP version, noting that the SMDP extension can be obtained as shown above for Q-Learning via Equation (11).

Steps in Bounded Actor Critic for Discounted Reward [10]:

- **Inputs:** Initialize all actor, $P(.,.)$, and critic, $V(.)$, values to zero. Set k, the number of iterations, to 0. Let $k_{\max}$ denote the maximum number of iterations for which the algorithm is run.

- Loop until $k = k_{\max}$

  - Let i be the current state. Select action a with probability of

  $$q(i,a) = \frac{e^{P(i,a)}}{\sum_{b \in \mathcal{A}(i)} e^{P(i,b)}}. \tag{12}$$

  Let j be the next state. Let $r(i,a,j)$ denote the immediate reward in the state transition.

  - Actor's update:

  $$P(i,a) \leftarrow (1-\alpha)P(i,a) + \alpha \left[r(i,a,j) + \gamma V(j) - V(i) \right]. \tag{13}$$

- Critic's update:
$$V(i) \leftarrow (1-\beta)V(i) + \beta\left[r(i,a,j) + \gamma V(j)\right]. \tag{14}$$
- Set $k \leftarrow k+1$. If $k = k_{max}$, exit loop; otherwise, set $i \leftarrow j$ and continue within loop.

- **Outputs:** The policy delivered by the algorithm is computed as follows. Declare the action for which $P(i,.)$ is maximum to be the optimal action for state i (do this for all values of i).

As noted above in the context of the average reward problem, the action-selection scheme used here (see Equation (12)) is called the Boltzmann scheme and very importantly does not need the temperature, U, which is necessary in the classical actor-critic. Also, this version keeps the actor's values bounded, unlike the classical actor-critic, which makes the actor's values unbounded and hence needs a temperature in the action-selection scheme. The classical version of the actor-update for the discounted reward problem was as follows [1]:

$$P(i,a) \leftarrow P(i,a) + \alpha\left[r(i,a,j) + \gamma V(j) - V(i)\right]. \tag{15}$$

Note that there is a difference between this update and the one in the algorithm above. The difference is that in the algorithm above, we use a multiplier of $(1-\alpha)$ to $P(i,a)$, which along with γ keeps the actor's value bounded. See Lawhead and Gosavi [10] for more details.

6 MDP Example

We end with a simple example from Gosavi [4]. Figure 1 shows a simple MDP; the legend in the figure explains the transition rewards and probabilities.

This means that the transition data is given as follows: $\mathbf{P}_a$ denotes the transition probability matrix for action a, while $\mathbf{R}_a$ denotes the transition reward matrix for action a.

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}; \mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix};$$

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix}; \mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

6.1 Average reward

We first consider the average reward problem on which you could use R-SMART. The optimal policy for this MDP is: action 2 in state 1 and action 1 in state 2. The average reward of the optimal policy is 10.56. When R-SMART was used on the above, with $\eta = 0.99$, we obtained the following results at the end of the learning phase: $\rho^{ITERMAX} = 10.06$;

$$Q(1,1) = 37.0754; Q(1,2) = 55.1019; Q(2,1) = 51.9332; Q(2,2) = 29.0445.$$

This implies that according to the algorithm, the best action is 2 for state 1 and is 1 for state 2, which coincides with the optimal policy.

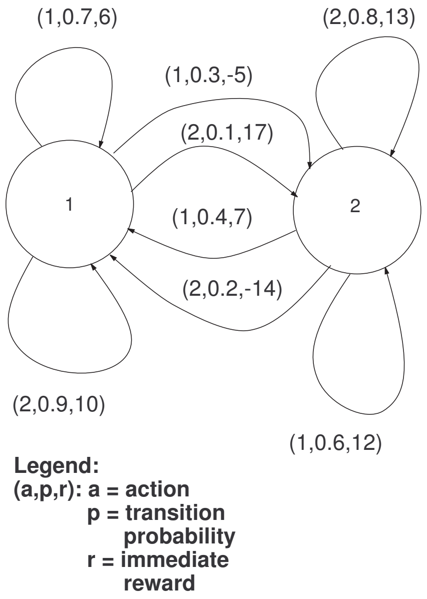

Figure 1: A two state MDP

6.2 Discounted reward

We consider the same problem as above but now with discounting, where $\gamma = 0.8$. The optimal policy is identical to that of the average reward case above. With Q-Learning, we obtained the following results:

$$Q(1,1) = 42.5892;\ Q(1,2) = 53.2902;\ Q(2,1) = 51.5412;\ Q(2,2) = 45.9043.$$

Here, the optimal values of the Q-factors are:

$$Q^*(1,1) = 44.84;\ Q^*(1,2) = 53.02;\ Q^*(2,1) = 51.87;\ Q^*(2,2) = 49.28.$$

As is clear, the algorithm generates the optimal policy, as well as Q-factors that approximate their optimal values. The optimal values of the Q-factors can be determined from Q-factor value iteration, which is discussed in the book.

7 Conclusions

The tutorial presented above showed you one way to solve an MDP provided you have the simulator of the system or if you can actually experiment in the real-world system. Transition probabilities of the state transitions were not needed in this approach; this is the most attractive feature of this approach.

We did *not* discuss what is to be done for large-scale problems. That is beyond the scope of this tutorial. What was discussed above is called the *lookup-table* approach in which each Q-factor is stored explicitly (separately). For large-scale problems, clearly it is not possible to store the Q-factors explicitly because there is too many of them. Instead one stores a few scalars, *called basis functions*, which on demand can generate the Q-factor for any state-action pair. Function approximation when done improperly can become unstable [4].

We can provide some advice to beginners in this regard. First, attempt to cluster states so that you obtain a manageable state space for which you can actually use a lookup table. In other words, divide the state space for each action into grids and use only one Q-factor for all the states in each grid. The total number of grids should typically be a manageable number such as $10,000$. If this does not work well, produce a smaller number of grids but use an incremental Widrow-Hoff algorithm, that is, a *neuron* (see [4]), in each grid. If you prefer using linear regression, go ahead because that will work just as well. If this works well, and you want to see additional improvement, do attempt to use *neural networks*, either neurons or those based on back-propagation [14]. It is with function approximation that you can scale your algorithm up to realistic problem sizes.

I wish you all the best for your new adventures with RL, but cannot promise any help with homework or term papers — sorry :-(

References

[1] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

[2] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena, MA, 1996.

[3] S. J. Bradtke and M. Duff. Reinforcement learning methods for continuous- time Markov decision problems. In *Advances in Neural Information Processing Systems* 7. MIT Press, Cambridge, MA, USA, 1995.

[4] A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, Springer, New York, NY, 2014. http://web.mst.edu/~gosavia/book.html

[5] A. Gosavi. Target-Sensitive Control of Markov and Semi-Markov Processes. *International Journal of Control, Automation, and Systems*, 9(5):1-11, 2011.

[6] A. Gosavi. Reinforcement Learning for Long-run Average Cost. *European Journal of Operational Research.* Vol 155, pp. 654-674, 2004. Can be found at: http://web.mst.edu/~gosavia/rsmart.pdf

[7] A. Gosavi. On Step Sizes, Stochastic Shortest Paths, and Survival Probabilities in Reinforcement Learning, *Conference Proceedings of the Winter Simulation Conference*, 2009. Available at: http://web.mst.edu/~gosavia/wsc_2008.pdf.

[8] A. Gosavi. Reinforcement Learning: A Tutorial Survey and Recent Advances. *INFORMS Journal on Computing.* Vol 21(2), pp. 178–192, 2009. Available at: http://web.mst.edu/~gosavia/joc.pdf

[9] K. Kulkarni, A. Gosavi, S. Murray, and K. Grantham. Semi-Markov adaptive critic heuristics with application to airline revenue management. *Journal of Control Theory and Applications*, 9(3):421–430, 2011.

[10] R. Lawhead and A. Gosavi. A bounded actor-critic reinforcement learning algorithm applied to airline revenue management. *Engineering Applications of Artificial Intelligence*, 82, 252-262, 2019.

[11] R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, MA, USA, 1998.

[12] MATLAB repository for Reinforcement Learning, created by A. Gosavi at Missouri University of Science and Technology, http://web.mst.edu/˜gosavia/mrrl_website.html.

[13] C. J. Watkins. *Learning from delayed rewards.* Ph.D. thesis, Kings College, Cambridge, England, May 1989.

[14] P. J. Werbos. *Beyond regression: New tools for prediction and analysis of behavioral sciences.*. Ph.D. thesis, Harvard University, USA, 1974.