A Tutorial for Reinforcement Learning

Abhijit Gosavi
Department of Industrial Engineering
The State University of New York at Buffalo
317 Bell Hall Box 602050, Buffalo, NY 14260-2050
Telephone: (716) 645-2357, Email:agosavi@buffalo.edu

June 10, 2004
Contents

1 Introduction 3
2 MDPs and SMDPs 3
3 RL 5
4 Conclusions 8
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.

RL is generally used to solve the so-called Markov decision problems (MDPs). In other words, the problem that you are attempting to solve with RL must be an MDP or its variant. 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. The book, “Simulation-based optimization: Parametric Optimization techniques and reinforcement learning” [1], discusses this topic in greater detail. The book is available in our library at SUNY, Buffalo.

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

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

- Identify problems that can be set up as MDPs / SMDPs.
- Use a basic RL algorithm.

We will not discuss how to use function approximation, but 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.

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 consider only one performance metric in this tutorial. This metric is called the average reward of a policy.

Let us assume that a policy named $\pi$ is to be followed. Then $\pi(i)$ will denote the action selected by this policy for state $i$. Every time the state changes, we say a *jump* has occurred. Let $x_s$ denote the state of the system before the $s$th jump. Then, the following quantity, in which $x_1 = i$, is called the average reward of the policy $\pi$ starting at state $i$.

$$\rho_i = \lim_{k \to \infty} \frac{E \left[ \sum_{s=1}^{k} r(x_s, \pi(x_s), x_{s+1}) | x_1 = i \right]}{k}$$  \hspace{1cm} (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 $E[.]$ 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 satisfy certain conditions; in many real-world
problems such conditions are usually satisfied. Then

\[ \rho_i = \rho \] for any value of \( i \).

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

Another popular performance metric, commonly described in books, is discounted reward. 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 if the starting state is \( i \), under certain conditions, is defined as:

\[
\rho_i = \lim_{k \to \infty} \frac{E \left[ \sum_{s=1}^{k} r(x_s, \pi(x_s), x_{s+1}) | x_1 = i \right]}{E \left[ \sum_{s=1}^{k} t(x_s, \pi(x_s), x_{s+1}) | x_1 = i \right]}. \tag{2}
\]

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.

Curses: For systems which have a large number of governing 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. 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 RL

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 algorithm that we will describe is called Relaxed-SMART (Semi-Markov Average reward Technique) [2].

The basic 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 arbitrary numbers in the beginning. 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.

The steps in the Learning Phase are given below.

- Step 1: Set the $Q$-factors to some arbitrary values (e.g. 0), that is:

$$Q(i, a) \leftarrow 0 \text{ for all } i \text{ and } a.$$
• Step 2: Select an action \( a \) in state \( i \) with probability \( 1/|A(i)| \). For example, if two actions are allowed in state \( i \), choose action 1 with probability \( 1/2 \) and the other action with the same probability. (For this, you can generate a random number between 0 and 1. If the number is less than or equal to 0.5, choose action 1 and choose action 2 otherwise.) If \( a \) denotes the action associated with the maximum \( Q \)-factor of state \( i \), set \( \phi = 0 \) (this means that the action selected is greedy with respect to the \( Q \)-factor.) Otherwise, set \( \phi = 1 \). Simulate action \( a \).

• Step 3: Let the next state be denoted by \( j \). Let \( r(i, a, j) \) denote the transition reward and \( t(i, a, j) \) denote the transition time. Then update \( Q(i, a) \) as follows:

\[
Q(i, a) \leftarrow (1 - \alpha^k)Q(i, a) + \alpha^k \left[ r(i, a, j) - \rho^k t(i, a, j) + Q_{\text{max}} \right],
\]

where \( Q_{\text{max}} \) denotes the highest \( Q \)-factor (\( Q \)-factor with the maximum value) in state \( j \).

• Step 4: If \( \phi = 1 \), that is the action was non-greedy, go to Step 5. Otherwise, update total_reward and total_time as follows.

\[
\text{total_reward} \leftarrow \text{total_reward} + r(i, a, j),
\]
\[
\text{total_time} \leftarrow \text{total_time} + t(i, a, j).
\]

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 5: Increment \( k \) by 1. Set:

\[
i \leftarrow j.
\]

If \( k < \text{ITERMAX} \), return to Step 2. Otherwise, for each state \( i \) declare the action \( u_i \) for which \( Q(i, .) \) is maximum to be the optimal action, and STOP.

The next phase is called the frozen phase because the \( Q \)-factors are not updated here. 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 very similar to the optimal in its performance.)

Steps in the Frozen Phase are as follows.

• Step 1: Use the \( Q \)-factors learned in the Learning Phase. Set iteration count \( k \) to 0. \( \text{ITERMAX} \) 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 2: Select for state \( i \) the action which has the maximum \( Q \)-factor. Let that action be denoted by \( u \). Simulate action \( b \).
• Step 3: Let the next state be denoted by $j$. Let $r(i, u, j)$ denote the transition reward and $t(i, u, j)$ denote the transition time.

Then update total_reward and total_time as follows.

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

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

• Step 4: Increment $k$ by 1. Set:

$$i \leftarrow j.$$

If $k < \text{ITERMAX}$, return to Step 2. Otherwise, calculate the average reward of the policy learned in the learning phase as follows:

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

and STOP.

The value of $\rho_k$ in the learning phase can also be used as an estimate of the actual $\rho$ 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.

4 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. There is a great deal of controversy about whether function approximation is actually stable [1].

We can provide some advice to beginners in this regard. Do not attempt to use back-propagation-based neural networks in your first attempt at function approximation. 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 [1]), in each grid. If you prefer using linear regression, go ahead because that will work just as well.

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

8
References
