Reinforcement Learning: Elementary Solution Methods

Yangang Chen

Department of Applied Mathematics, University of Waterloo, 20496836

Abstract

I investigate three basic solution techniques for reinforcement learning: dynamic programming, Monte Carlo and temporal-difference learning. I review these three solution techniques, which are essentially Generalized Policy Iteration with different policy evaluation approaches. In particular, I provide theoretical discussions on dynamic programming in terms of numerical linear algebra. I implement these algorithms for three concrete examples: Gridworld, Tic-tac-toe and robotic control of pendulum swing-up with limited torque. The results suggest that dynamic programming and temporal-difference learning are efficient learning methods.

The report is mainly empirical evaluation (Option B), with additional theoretical discussions. All the source codes for this report are developed from scratch by myself, and are available on my GitHub at `https://github.com/yangangchen/CS698-Course-Project`.

1 Introduction

Reinforcement learning, together with supervised and unsupervised learnings, are the three most fundamental categories of machine learning. The objective of reinforcement learning is to let an agent that interacts with its environment learn to make optimal decisions and maximize its long-term rewards. Unlike the typical supervised (or unsupervised) learning, which relies on a fully-labeled dataset (or does not use labels at all), reinforcement learning is able to exploit the labels that are only partially accessible. Such feature makes reinforcement learning both accurate and flexible. Reinforcement learning is widely applied in robotic controls, natural language processing, games, economics, etc. In particular, due to some recent innovations, such as the success of AlphaGo, reinforcement learning is deemed as the key to the realization of Artificial Intelligence.

The primary objective of this report is to compute the solution of reinforcement learning problems. In the literature, these solution techniques are usually summarized into three categories: dynamic programming methods, Monte Carlo methods and temporal-difference learning [5, 3]. These three families of solution methods have their own strengths and weaknesses. Dynamic programming methods give exact solutions for the optimal policy and the value function in an efficient manner. However, it requires a complete knowledge on the dynamics of the system. Monte Carlo methods are model-free and conceptually simple, but suffer from slow learning rates. Temporal-difference methods (i.e. Sarsa and Q-learning) are not only model-free, but also efficient in terms of learning rates. Indeed, temporal-difference methods are among the most popular reinforcement learning techniques nowadays.

2 Reinforcement Learning

In this section, I briefly review the reinforcement learning problem, including the mathematical model and the elementary solution framework called Generalized Policy Iteration. Interested readers are referred to Chapter 1-7 of Sutton and Barto's *Reinforcement Learning* [5].

2.1 Mathematical model

The problem of reinforcement learning can be mathematically formulated as follows [5]: At the t-th timestep, an agent, in a state s_t, decides to take an action a_t that depends on s_t. Due to the interaction with the environment, the state evolves from s_t to s_{t+1} under certain dynamics, governed by the transition probability $p(s_{t+1}|s_t, a_t)$. Meanwhile the agent gains an instant reward $r(s_t, a_t, s_{t+1})$ from the environment. The objective of the agent is to maximize its expected long-term reward, called value function:

$$V^\pi(s) \equiv \mathbb{E}_{\substack{\pi(s,a)\\ p(s'|s,a)}} \left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| s_0 = s \right\}. \tag{2.1}$$

Here $\pi(s,a) = Prob(a|s)$, called policy, is the probability distribution of taking an action a under its current state s[1]. The parameter γ, called discounted factor, prioritizes future reward over the instant reward. In other words, the objective of reinforcement learning is to find an optimal policy π^* that maximizes the value function as follows:

$$\pi^* = \underset{\pi}{\text{argmax}}\ V^\pi(s). \tag{2.2}$$

An alternative of the value function is the action-value function $Q^\pi(s,a)$:

$$Q^\pi(s,a) \equiv \mathbb{E}_{\substack{\pi(s,a)\\ p(s'|s,a)}} \left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| s_0 = s, a_0 = a \right\}. \tag{2.3}$$

The action-value function and the value function are related by

$$V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a). \tag{2.4}$$

2.2 Generalized Policy Iteration (GPI)

Once the mathematical formulation (2.2) is established, the primary task is to find its solution. The fundamental idea of solving the reinforcement learning problem (2.2) is Generalized Policy Iteration (GPI) [5]. GPI arises from the essential fact that the policy π and the value function $V^\pi(s)$ are coupled to each other. On the one hand, by fixing the policy π, we can compute the value function $V^\pi(s)$. This process is called *policy evaluation*. On the other hands, by fixing the value function $V^\pi(s)$, we can update the policy π towards optimality. This process is called *policy improvement*. GPI iterates between policy evaluation and policy improvement, which eventually converges towards the optimal policy and the optimal value function.

One key component of GPI is the policy improvement. A straightforward choice for the optimal policy is greedy policy. However, the greedy policy may lead to a local optimum rather than the global optimum. In order to find the global optimum, we may want to "explore" the action space that is not visited by the greedy action. Motivated by this, we use ϵ-greedy policy:

$$\begin{aligned} a^* &\equiv \underset{a\in\mathcal{A}(s)}{\text{argmax}}\ Q^\pi(s,a), \\ \pi^*(s,a) &= \begin{cases} 1-\epsilon, & a = a^*, \\ \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{otherwise}, \end{cases} \end{aligned} \tag{2.5}$$

where $\mathcal{A}(s)$ is the action space at the state s and $|\mathcal{A}(s)|$ is its volume.

The other key component of GPI is the policy evaluation. Choosing different algorithms for policy evaluation gives rise to the three elementary solution methods: dynamic programming, Monte Carlo and temporal-difference learning. I will discuss these three methods in the next section.

3 Elementary Solution Methods

In this section, we review the three categories of elementary solution techniques: dynamic programming, Monte Carlo and temporal-difference learning methods. All the detailed algorithms in this section can be found in Chapter 4-7 of Sutton and Barto's *Reinforcement Learning* [5].

[1]The policy is described in terms of probability distribution rather than a function $a = \pi(s)$, since this allows the agent to choose among more than one actions under a given state.

3.1 Monte Carlo methods

The basic idea of the Monte Carlo methods is to directly use the definition of the value function (2.1) or the action-value function (2.3), and estimate the expectation values of the long-term reward $\mathbb{E}\{\sum_{t=0}^{\infty} \gamma^t r_{t+1}\}$ by the empirical averages of the dataset or the experiments.

Monte Carlo methods are conceptually simple, but they have severe limitations. The evaluation of the long-term reward for a given (s, a) requires a complete episode[2], and in principle, a full Monte Carlo simulation requires averaging over an infinite number of episodes to converge to the (action-)value function. Unfortunately, in many practical applications, an episode can be extremely long, and the total number of the episodes that can be experimented is usually limited.

3.2 Temporal-difference (TD) methods

The good news is that we can reformulate the definitions (2.1) and (2.3), such that the policy evaluation can be performed instantly at each timestep (rather than till the end of the episode). This gives rise to temporal-difference (TD) methods. The idea is to reformulate the action-value function (2.3) as

$$Q^\pi(s,a) = \mathbb{E}_{p(s'|s,a)}\left\{r(s,a,s') + \gamma V^\pi(s')\right\}, \tag{3.1}$$

which is a direct result of the definitions (2.1) and (2.3). If we apply (2.4), and the fact that the transition probability $s \to s'$ through all "hidden" actions is $Prob(s'|s) = \sum_a Prob(a|s) Prob(s'|a,s) = \sum_a \pi(s,a) p(s'|s,a)$, then we have

$$V^\pi(s) = \mathbb{E}_{Prob(s'|s)}\left\{r(s,s') + \gamma V^\pi(s')\right\}. \tag{3.2}$$

Similarly, one can show that

$$Q^\pi(s,a) = \mathbb{E}_{Prob(s',a'|s,a)}\left\{r(s,a,s') + \gamma Q^\pi(s',a')\right\}. \tag{3.3}$$

[2]In reinforcement learning literatures, an episode refers to a complete sequence (or path) of events from a starting state to a terminal state, while a timestep refers to each instance of event inside the sequence

The meaning of the reformulation (3.2) is that the current value function $V^\pi(s)$ is equal to the current expected return from two sources: (1) the immediate return $r(s, s')$ for being at the states s, s' (2) the discounted future expected return $V^\pi(s')$ as s is transited to s'. Reformulation (3.3) can be interpreted in the similar vein.

The advantages of the reformulations (3.2) and (3.3) are two-fold. One is that the exact (action-)value functions can be interpreted as expectation values. Hence, they can be approximated by empirical averages. They do not rely on the explicit knowledge of the dynamics of the system, encoded in the underlying transition probabilities. This advantage is similar to Monte Carlo methods. The other advantage is that the estimation of the long-term reward $\mathbb{E}\{r(s,s') + \gamma V^\pi(s')\}$ and $\mathbb{E}\{r(s,a,s') + \gamma Q^\pi(s',a')\}$ only requires the information from the current timestep and the succeeding timestep. TD methods use this advantage and update the (action-)value functions instantly at every timestep.

3.3 Dynamic programming methods

Both Monte Carlo and TD methods use empirical average to "approximate" the (action-)value function. However, can we compute the (action-)value function exactly? This requires using the precise values of the underlying transition probabilities. Dynamic programming methods are such type of methods. They are based on another reformulation of the definition (2.1) and (2.3), called Bellman equations. The key towards the Bellman equations is to expand Equation (3.1) as

$$Q^\pi(s,a) = \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma V^\pi(s')\right]. \tag{3.4}$$

Furthermore, by (2.4), we have

$$\begin{aligned} V^\pi(s) = & \sum_a \pi(s,a) \sum_{s'} p(s'|s,a) \\ & \left[r(s,a,s') + \gamma V^\pi(s')\right], \forall s. \end{aligned} \tag{3.5}$$

Equation (3.5) is the famous Bellman equations.

Remarkably, given the policy $\pi(s,a)$, the Bellman equations become a linear system of equations with respect to the value function $V^{\pi}(s)$. Hence, solving the Bellman equations turns out to be manageable and efficient.

Typical dynamic programming methods are (exact) policy iteration and value iteration. Policy iteration is GPI, where at each policy evaluation, the system of linear Bellman equations is solved in the exact sense. More specifically, given a policy $\pi(s,a)$, we construct the linear system (3.5) with respect to $V^{\pi}(s)$, solve the linear system for $V^{\pi}(s)$, and then update the action-value function $Q^{\pi}(s,a)$ by (3.4).

In the literature, a common approach for solving the linear Bellman equations is to use the following iterative scheme:

$$V^{\pi}_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma V^{\pi}_k(s') \right], \tag{3.6}$$

where k is the iteration index. From the numerical linear algebra point of view, this turns out to be a Jacobi iteration. Unfortunately, if γ is close to 1, then the number of Jacobi iterations for convergence usually grows quickly as the size of the linear system N increases, which is very slow. Instead, my approach is to use a better linear solver, such as the direct Gaussian elimination. As a side remark, I can also choose other fast linear solvers, such as Krylov subspace methods (i.e. biCGSTAB, GMRES), or multigrid methods, which are much faster solvers than the Jacobi iteration. The key point is that the policy evaluation here is to solve the linear system (3.5), which turns out to be a linear system with a sparse diagonally-dominant M-matrix (I will come back to this statement soon).

The other typical dynamic programming method is value iteration. Value iteration is GPI where the solution of the system of linear Bellman equations is approximated by one step Jacobi iteration (3.6). Similar to the linear Jacobi iteration, value iteration is usually slow, so I do not consider value iteration.

The advantage of dynamic programming methods is that they are guaranteed to converge. Moreover, the convergence rate of policy iteration is typically fast (below 10 iterations). The guaranteed convergence is deeply related to the following theory in numerical linear algebra:

Lemma 1 (Diagonally dominant M-matrix)**.** Denote the linear system (3.5) under the fixed policy the policy $\pi(s,a)$ as

$$AV = b. \tag{3.7}$$

Then the matrix A is a diagonally dominant M-matrix.

Proof. Equation (3.5) can be rewritten as

$$\begin{aligned}&\left[1-\gamma \textstyle\sum_a \pi(s,a)p(s|s,a)\right] V^{\pi}(s)\\ &-\gamma \textstyle\sum_{s'\neq s}\sum_a \pi(s,a)p(s'|s,a)V^{\pi}(s')\\ &= \textstyle\sum_a \pi(s,a)\sum_{s'} p(s'|s,a)r(s,a,s').\end{aligned}$$

Then $A_{s,s} = 1-\gamma\sum_a \pi(s,a)p(s|s,a)$, and $A_{s,s'} = -\gamma\sum_a \pi(s,a)p(s'|s,a)$. For $0<\gamma<1$, we have (1) $A_{s,s}>0$; (2) $A_{s,s'}\le 0$ for $s'\neq s$; and (3) $A_{s,s}-\sum_{s'\neq s}A_{s,s'}>0$. Hence, A is a diagonally dominant M-matrix [4]. ☐

Theorem 1 (Convergence of policy iteration)**.** If A is an M-matrix under all admissible control $a\in\mathcal{A}$, then policy iteration for the nonlinear system

$$\max_a \left[A(a)V - b(a)\right] = 0 \tag{3.8}$$

is guaranteed to converge to its unique solution.

Proof. Please see [1]. ☐

A common statement in the literature is that dynamic programming requires a full model on the dynamics of the system, namely, a full knowledge on the transition probability $p(s'|s,a)$. However, in my opinion, this does not prevent us from using dynamic programming when the exact transition probability is unavailable. My solution is to estimate the transition probability from the empirical data or (Monte-Carlo) experiments. More specifically, the underlying transition probability $p(s'|s,a)$ is estimated by the empirical transition probability (normalized frequency). Once these parameters are estimated, then dynamic programming can be applied to find the optimal policy and the optimal value function.

4 Case Studies

This section includes case studies and the corresponding experimental results using the elementary solution methods. The three concrete examples are Gridworld, Tic-tac-toe and robotic control of pendulum swing-up with limited torque. All the experimental results are implemented by my own codes written from scratch. Source codes are available on my GitHub at https://github.com/yangangchen/CS698-Course-Project.

4.1 Gridworld

Gridworld is a toy model for reinforcement learning. In the Gridworld, a walker moves on a grid where some locations have rewards or penalties, and the movement of walker has costs. The walker's goal is to reach the reward point with minimal steps but meanwhile avoid the penalties. The Gridworld I study is from http://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html; see Figure 4.1. Gridworld is a good starting point, since the state space is small (88 states), the action space is small (4 actions), and the transition probability $p(s'|s,a)$ is fully accessible and is either 1 or 0.

To compute the optimal value function and the optimal policy, I implement the following methods: dynamic programming (policy iteration), on-policy Monte Carlo, on-policy TD method (Sarsa) and off-policy TD method (Q-learning).

My dynamic programming code yields the same optimal value function and optimal policy as those on the website. In particular, my walker's optimal path from the starting point to the end point matches the website's. However, there is a difference between my algorithm and the website's. In my simulation, I use policy iteration, where for each policy evaluation, I solve the system of linear Bellman equations for the value function using direct Gaussian elimination. My approach is efficient, as policy iteration converges in 8 steps (see Table 4.1), and the computation is completed instantly within 2 seconds. As a comparison, the website uses value iteration. As one

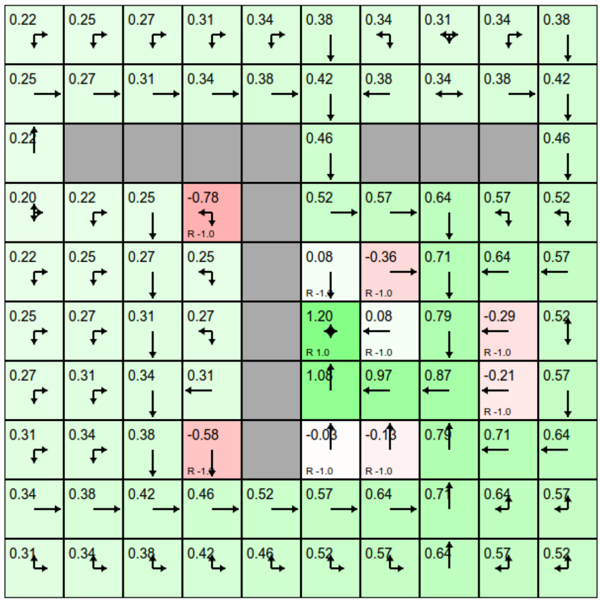
0.22 0.25 0.27 0.31 0.34 0.38 0.34 0.31 0.34 0.38
0.25 0.27 0.31 0.34 0.38 0.42 0.38 0.34 0.38 0.42
0.22 0.46 0.46
0.20 0.22 0.25 -0.78 R -1.0 0.52 0.57 0.64 0.57 0.52
0.22 0.25 0.27 0.25 0.08 R -1.0 -0.36 R -1.0 0.71 0.64 0.57
0.25 0.27 0.31 0.27 1.20 R 1.0 0.08 R -1.0 0.79 -0.29 R -1.0 0.52
0.27 0.31 0.34 0.31 1.08 0.97 0.87 -0.21 R -1.0 0.57
0.31 0.34 0.38 -0.58 R -1.0 -0.03 R -1.0 -0.13 R -1.0 0.79 0.71 0.64
0.34 0.38 0.42 0.46 0.52 0.57 0.64 0.71 0.64 0.57
0.31 0.34 0.38 0.42 0.46 0.52 0.57 0.64 0.57 0.52

Figure 4.1: Gridworld: the optimal value function and the optimal policy.

(1) Dynamic programming: Policy iteration

Step	0	1	2	3	4
$\|\Delta V\|$	72.5	66.8	4.32	23.0	1.31
Step	5	6	7	8	
$\|\Delta V\|$	0.21	1.01	0.20	0	

(2) On-policy Monte Carlo

Step	0	1000	2000	3000
$\|\Delta V\|$	14.2	3.64	1.24	1.18
Step	12500	50000	200000	800000
$\|\Delta V\|$	0.68	0.024	1.7x10^{-3}	1.2x10^{-4}

(3) On-policy temporal-difference: Sarsa ($\alpha = 0.2$)

Step	0	1000	2000	3000
$\|\Delta V\|$	30.9	1.79	0.16	0.014
Step	4000	5000	6000	7000
$\|\Delta V\|$	0.0016	2.5x10^{-4}	5.1x10^{-5}	1.1x10^{-5}

(4) Off-policy temporal-difference: Q-Learning ($\alpha = 0.2$)

Step	0	1000	2000	3000
$\|\Delta V\|$	30.8	1.69	0.15	0.013
Step	4000	5000	6000	7000
$\|\Delta V\|$	0.0013	1.6x10^{-4}	4.0x10^{-5}	9.4x10^{-6}

Table 4.1: Gridworld: Convergence of (1) policy iteration, (2) Monte Carlo, (3) Sarsa, and (4) Q-learning. In (1), $\|\Delta V\|$ is the change of value function at each iteration, while in (2)-(4), $\|\Delta V\|$ is the change of value function per 1000 iterations.

can experiment on the website, it takes 100+ steps to converge to the solution.

My Monte Carlo and TD simulations converge to the same results as my dynamic programming simulation. To ensure this, for each episode of my Monte Carlo and TD simulation, I start with a randomly chosen (s, a), such that (almost) all the $Q(s, a)$ are visited and updated. Table 4.1 illustrates that the convergence rates of both Sarsa and Q-learning (7000 steps) are much faster than the Monte Carlo method (800000+ steps). On the website, however, its TD simulation converges to a different result from its dynamic programming simulation, since its algorithm always starts from a particular starting state, and thus only a portion of $Q(s, a)$ is visited and updated.

I also experiment non-zero ϵ (e.g. $\epsilon = 0.4$). In this case, a constant learning rate α may not lead to convergent TD methods. However, my numerical experiments show that by choosing a small enough α (e.g. $\alpha = 0.2$), the TD methods still converge.

4.2 Tic-tac-toe

Tic-tac-toe is a board game on a 3×3 board, where a player wins by first connecting three stones horizontally, or vertically, or diagonally.

I use both dynamic programming (policy iteration) and TD Q-learning methods to train the AI. Here are my implementation details:

(1) For both policy iteration and Q-learning, I use greedy policy rather than ϵ-greedy policy. The reason is that Tic-tac-toe has deterministic winning and tying strategy. In my experiments, greedy policy seems to be better at learning the deterministic strategy.

(2) For policy iteration, the key is to acquire the knowledge on the underlying transition probability $p(s'|s, a)$, In order to estimate them, I run 2000 random games between two "naive" AIs with uniform policies, record transition frequency of $(s, a) \to s'$, and use the empirical transition probability (normalized frequency) to estimate the underlying transition probability.

(3) For Q-learning, the training is done by letting two AIs play against each other for 3000 times. Each AI updates its own policy and action-value function

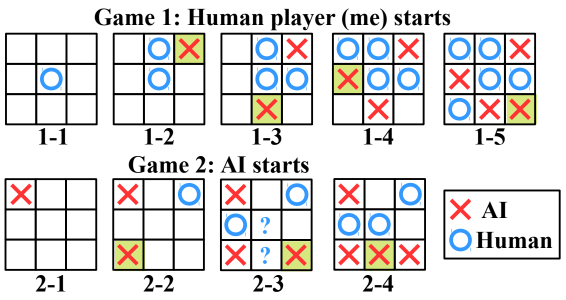

Figure 4.2: Tic-tac-toe: my AI versus human player (me). The AI is trained by the Q-Learning algorithm. The red crosses are AI's moves, while the blue circles are my moves. The green boxes are where the AI decides to place the stone with the probability 1.

simultaneously, such that they both improve their performances as more games are played. The difficulty of using this approach is that even if greedy policy is used, the training among the two AIs may not necessarily converge. My conjecture is that the two AIs are playing a zero-sum game, where each AI not only maximizes its own value function, but also minimizes the opponent's. Zero-sum game may not necessarily converge (similar to what we have seen in Generative Adversarial Networks). To ensure convergence, I decrease the training rate by $\alpha_{s,a} = 1/n_{s,a}$, where $n_{s,a}$ is the number of times that (s, a) is visited.

(4) I exploit the symmetry to reduce the size of the state space. Naively, there are $3^9 = 19683$ states on a size-9 board. Among them 8533 states are legal (e.g. It is illegal to have all 9 stones from Player 1). However, if I treat the states under rotational and reflectional transformations as equivalence, then the size of the state space is reduced to 1192. This turns out to dramatically increase the training efficiency.

For Q-learning algorithm, my experiments show that after 3000 episodes, *one of my AIs gains the ability of always winning or tying the games*. Figure 4.2 shows the Tic-tac-toe games between my AI and human player (me). The red crosses are AI's moves, while the blue circles are my moves. The green square elements are where the AI decides to place the stone with the probability 1. In Game 1, which I start, the AI successfully defends all my at-

(1) Dynamic programming: Policy iteration

Step	0	1	2	3	4
$\|\Delta V\|$	478	748	14.9	0.019	0

(2) Temporal-difference: Q-Learning

Step	0	100	200	300
$\|\Delta V\|$	93.2	31.3	30.2	26.2
Step	1000	2000	3000	4000
$\|\Delta V\|$	1.49	1.12	0.017	0.0092

Table 4.2: Tic-tac-toe: (1) Convergence of policy iteration, where $\|\Delta V\|$ is the change of value function at each iteration. (2) Convergence of Q-learning, where $\|\Delta V\|$ is the change of value function per 100 iterations.

tacks with certainty. That is, the greedy policy instructs the AI to place the stones at the green boxes. In Game 2, which the AI starts, I first place the stone at the top-right corner. After that, the AI implements a deterministic winning strategy, such that at Step 2-3, no matter where I place my stone, the AI will always win the game.

For policy iteration, the AIs can win or tie the games for most of the times, although the AIs may occasionally miss the moves that they must defend. Increasing the number of episodes for a better approximation of $p(s'|s,a)$ does not help. A possible explanation is that the AIs are trained when they play against each other naively, and their polices are not updated until the very end. As a result, they can play well against naive players, but not "smart" players.

Regarding the computational complexity, policy iteration is very efficient and takes less than 30 seconds. More specifically, the generation of 2000 games and statistics of $p(s'|s,a)$ is indeed cheap (around 10 seconds). Policy iteration converges in 4 iterations; see Table 4.2. Each policy iteration takes around 3 seconds. As a comparison, Q-learning takes 74 seconds in total, as each game requires an update of the policy and the value function. Table 4.2 shows that Q-learning also converges.

4.3 Robotic control of pendulum swing-up

In the previous cases, states and actions are discrete. Reinforcement learning can also be applied for continuous states and actions. I consider an example of robotic control of pendulum swing-up with limited torque from Reference [2]; see Figure 4.3.

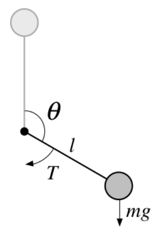

Figure 4.3: Robotic control of pendulum swing-up: the mathematical model [2].

For simplicity, we consider a deterministic model without noise. The state $s \equiv (\theta,\omega)^T$ is a two-variable vector consisting of pendulum angle $\theta \in [-\pi,\pi)$ and angular speed $\omega \in [-\infty,\infty)$. The action (or the control) is the external torque from the engine, $a \in [-a_{max}, a_{max}]$, where a_{max} is the maximum torque that can be generated by the engine. The policy is the greedy action, namely, $\pi(s,a) = \delta_{a,a^*}$. Instead of using the transition probability, here the dynamics of the pendulum, governed by Newton's law, is mathematically a differential equation: $\dot{s}(t) = f(s(t),a(t))$, where $f(s,a) \equiv \left(\omega, \frac{1}{ml^2}(-\mu\omega + \frac{g}{l}\sin\theta + a)\right)^T$. The reward is $r(s) = \cos\theta - c\frac{a^2}{2}$, where the first term is the reward by keeping the pendulum staying up, while the second term is the cost of applying external torque. The value function is[3]

$$V^a(s) = \int_t^\infty e^{-\gamma(\tau-t)} r(s(\tau),a(\tau))d\tau. \qquad (4.1)$$

The objective is to find the optimal action (or optimal policy) a^* that maximizes the value function $V^a(s)$. The continuous version of the Bellman equation, which is usually called Hamilton-Jacobi-Bellman (HJB) equation, is a partial differential equation:

$$\max_{a\in[-a_{max},a_{max}]} \left\{\nabla_s V \cdot f(s,a) + r(s,a) - \gamma V\right\} = 0. \qquad (4.2)$$

I consider implementing dynamic programming, i.e. policy iteration, on the HJB equation. To be more

[3]Since we are considering a deterministic model, there is no need to take the expectation.

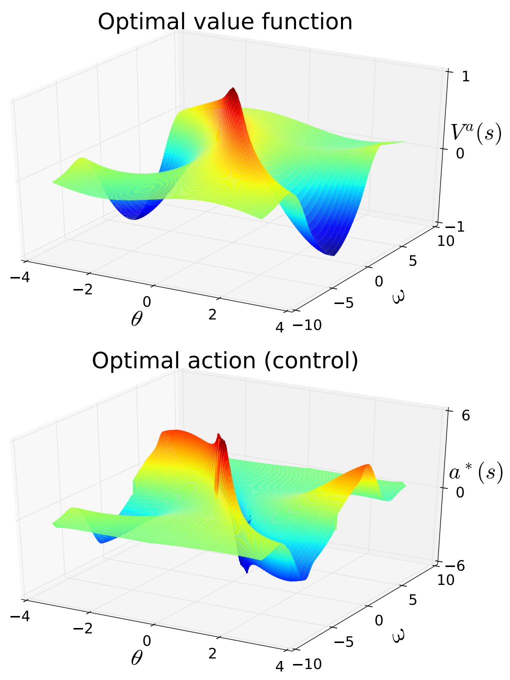

Figure 4.4: Robotic control of pendulum swing-up: the optimal value function and the optimal policy.

Step	0	1	2	3	4	5
$\|r\|$	6402	612	106	22.3	10.1	5.67
Step	6	7	8	9		10
$\|r\|$	2.27	0.46	0.051	1.1x10^{-3}		5.6x10^{-7}

Table 4.3: Robotic control of pendulum swing-up: Convergence of policy iteration, where $\|r\|$ is the norm of the residual.

precise, I first discretize the HJB equation using finite difference methods. The resulting discretized system can be symbolically written as

$$\max_{a\in[-a_{max},a_{max}]} \left\{A(a)V - b(a)\right\} = 0. \qquad (4.3)$$

Once I obtain the discretized equation, policy iteration becomes exactly the same as the algorithm for the discrete reinforcement learning problems. An important remark is that I use upwinding discretization on $\nabla_s V$ to make sure that the resulting matrix $A(a)$ is a diagonally-dominant M-matrix, which further makes sure that policy iteration converges. Conversely, central difference for $\nabla_s V$ does not yield a diagonally-dominant M-matrix.

Figure 4.4 shows the optimal value function and the optimal policy computed by my code. The optimal value function looks the same[4] as Figure 3 of Reference [2]. Policy iteration converges in 10 iterations.

5 Conclusion

The theories and experiments suggest that Monte Carlo methods are conceptually easy but practically slow; dynamic programming methods are the most efficient when the dynamics of the system is known or can be estimated well by the data; TD methods are model-free and also very efficient.

Due to the time constraint, I would leave the following topics for future explorations: (1) TD(λ) algorithms; (2) TD methods for continuous reinforcement learning problems [2]; (3) reinforcement learning with huge state-action spaces and high dimensions, which requires combining the elementary solution techniques with other machine learning techniques, such as neural networks.

References

[1] O. Bokanowski, S. Maroso, and H. Zidani, *Some convergence results for Howard's algorithm*, SIAM J. Numer. Anal., 47 (2009), pp. 3001–3026.

[2] K. Doya, *Reinforcement learning in continuous time and space*, Neural computation, 12 (2000), pp. 219–245.

[3] A. Gosavi, *Reinforcement learning: A tutorial survey and recent advances*, INFORMS Journal on Computing, 21 (2009), pp. 178–192.

[4] Y. Saad, *Iterative methods for sparse linear systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, second ed., 2003.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1, MIT press Cambridge, 1998.

[4]Note that the range of θ in Figure 3 of Reference [2] is $[-\pi, \frac{3}{2}\pi)$, different from the range in my Figure 4.4.