Programming Project 6: Reinforcement Learning with Value Iteration, Q-Learning, and SARSA

Ricca D. Callis RCALLIS1@JH.EDU
Whiting School of Engineering
Engineering for Professionals, Data Science
Johns Hopkins University

Abstract

This project provided students enrolled in an Introduction to Machine Learning course (605.649.83.SU20), at Johns Hopkins University, the opportunity to implement three reinforcement learning algorithms: Value Iteration, Q-Learning, and SARSA. This project utilized these three reinforcement learning techniques to solve the Racetrack problem, a standard control problem. The algorithms were compared based on the score they obtained in solving the problem on various racetracks by varying parameters such as convergence tolerance, number of iterations, etc. This project found that performance of all algorithms was comparable.
Keywords: reinforcement learning, value iteration, Q-learning, SARSA

1 Problem Statement & Hypothesis

1.1 Introduction to Reinforcement Learning

Reinforcement learning is a machine learning technique which maps situations to actions in a way which maximizes a numerical reward (e.g., Sutton & Barton, 2020; Mitchell, 1997). The learner (or, agent) is not told which actions to take, but instead uses trial-and-error to discover which actions yield the most reward. Most often, actions affect the immediate reward and also all subsequent rewards.

The two key components of reinforcement learning, trial-and-error and delayed reward, create a challenge for the learning, which then has to find a balance between exploration and exploitation. Specifically, the agent has to exploit what it has already experienced in order to obtain a reward, but it also has to explore in order to make better actions in the future.

Beyond the agent and the agent and the environment, there are also four main subelements of a reinforcement learning system: a policy, a reward signal, a value function, and a model of the environment (e.g., Sutton & Barton, 2020; Mitchell, 1997). A policy defines the learning agent's way of behaving at a given time. More specifically, it maps perceived states of the environment to actions to be taken when in those states. A reward signal is the goal. At each step (or episode), the environment sends to the learning agent a reward (i.e., a single number). The agent's goal is to maximize the total reward it receives over the duration of the problem. Thus, the reward signal is the means by which the policy is altered. The value, on the other hand, is the total amount of reward an agent can expect to accumulate over time from that state. A model of the environment is used for planning, or ways of determining a course of action. Reinforcement learning may include model-based methods or model-free methods.

Reinforcement learners use Markov Decision Processes (MDP) to solve problems. Specifically, MDPs introduce 3 aspects: sensation, action, and goal (e.g., Sutton & Barton, 2020; Mitchell, 1997). As mentioned, a learning agent must be able to sense the state of its environment and must be able to take actions to achieve it's goal. In MDP, the optimal decision depends only on the current state of the agent. At each time step, after an action is taken, a reward is given.

1.2 The Racetrack Problem

The Racetrack Problem is a standard control problem (e.g., Sutton & Barton, 2020). Given a pre-defined racetrack (environment), a racecar (the agent) must navigate the course, without driving off-track, in the shortest amount of time possible. The racetrack itself is represented as a Cartesian grid, allowing the racer's position to be indicated and tracked by (x, y) coordinates. Each racetrack is marked by a starting line, a finish line, clearly demarked racetracks, and "walls" (or, areas which are indicated as off-track).

At each time step, the state of the racer is given by the racer's location (x_t, y_t) and velocity (V_{xt}, V_{yt}). The agent (racecar) can only control the acceleration (a_x, a_y) at any given time step, which then influences its state. Acceleration values can be assigned as -1, 0, 1. At each time step (i.e., episode), velocity is updated using the acceleration, followed by update of location.

1.3 This Project

This project implemented Value Iteration, Q-Learning, and SARSA to solve the Racetrack Problem. Three pre-determined racetracks, each with increasing size and difficulty (L-track, O-track, and R-track) were used as environments. The racecar (agent) had a maximum velocity of $(V_{xt}, V_{yt}) \in (\pm 5, \pm 5)$**.** Each attempt to accelerate had a 20% fail rate, thereby making this system non-deterministic. To ensure racecars stay on course, each racetrack also had two variants: harsh, simple. Harsh-variant racetracks yielded a strong penalty for racecars which hit a wall or dr0ve off-track. Specifically, if a racecar droves off-track, the car was re-positioned at the starting line and the velocity was reset to 0. Simple-variant racetracks yielded a smaller penalty. Here, off-track racecars were penalized by resetting position to the previous location, just prior to the accident, and the velocity was reset to 0.

1.4 Expectations

Overall. It is hypothesized that all 3 algorithms (Value Iteration, Q-Learning, and SARSA) will be able to solve all 3 racetracks (L-track, O-track, R-track) because they all meet the Markov criteria that all information needed is encoded in the current state. However, since these algorithms take varying times to train, some may have difficulty finding an optimal policy for all of tracks.

Track Variants. It is hypothesized that harsh-variant tracks will take longer and will be more difficult to train, because they reset the racecar to the start of the racetrack. This could potentially give large policy values to states which are close to the finish line.

Furthermore, since there is a 20% fail rate, harsh-variants may take more iterations to converge to correct values, even when otherwise good decisions had been made.

2 Description of Algorithm

2.1 Datasets

The datasets used for this project were four ASCII representations of racetracks. The first line in the data file contained the number of rows and columns in the racetrack, as a comma delimited pair. The remaining data in the file depicted a uniquely-shaped map, represented as a Cartesian grid. Each cell of the grid, or (x,y) coordinate, indicated a specific location on the map, each containing one of four characters: 'S', 'F', '.', and '#'. All 'S' characters represented one of the starting positions. All 'F' characters represented one of the finish-line positions. All '.' Characters represented an open racetrack, where if an action is applied, there'd be a 20% probability that the action would fail and there would be no change in velocity of the agent. Finally, all '#' characters represented a wall, or off-track position of the course, where the agent would be penalized for going. The racetracks were either of L, O, or R shapes. The performance of all the algorithms were done on all racetracks, both simple and harsh-variants (as described above).

2.2 Notation

The following notations will be used to describe the algorithms:

- S: The set of states in the environment (racetrack)
  - where, s or s_t is the present state and s' or s_{t+1} is the next state
- A: The set of actions that can be executed by the agent (racecar)
- Policy $\pi(s)$: A mapping from state s to action a.
- $Q(s, a)$: A mapping which dictates the expected reward for choosing action a from state s.
- γ: The discount rate, or the amount by which a future reward is discounted. A reward in t iterations gets discounted by a factor of γ^t
- V^*: Expected value or utility of being in state s
- r: Reward
- $P(s_{t+1}|s_t, a_t)$: The probability of going into state s_{t+1} from state s by choosing action a

2.3 The Bellman Equation

The Bellman Equation is used to calculate the utility of being in a state as the immediate reward for that state plus the expected discounted utility of the next state, weighted by the likelihood of ending up in the state after executing action a_t from s_t. The Bellman Equation is given by:

$$V^*(s_t) = \underset{a_t}{\operatorname{argmax}}\, Q^*(s_t, a_t)$$

$$= \underset{a_t}{\operatorname{argmax}}\, E\left(r_{t+1} \sum_{t=1}^{\infty} \gamma^{i-1} r_t + i + 1\right)$$

$$= E\left(r_{t+1} + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \operatorname*{argmax}_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \right)$$

2.4 Value Iteration

The Value Iteration algorithm determines an optimal policy by calculating the utility of each state and then uses the state utilities to select an optimal action in each state. Thus, Value Iteration attempts to find $Q^*(s, a)$ by iteratively updating each entry in Q as the sum of the current value and the discounted value from executing each action. More specifically, at each iteration, each state-action pair updates the estimate of its value by identifying future states and their respective values and then discounting backward to add the reward of the future state to its value.

The Value Iteration algorithm uses the Bellman Equation for each state. Thus, for each *n* state, we have *n* simultaneous equations. To calculate, the utility of all states is initialized to 0. Then, the utility value of each state is updated according to the Bellman Equation in every iteration. Updates continue until reaching equilibrium. The algorithm converges to values close to the optimal $Q^*(s, a)$, meaning these optimal utility values are used to create action (i.e., move the racecar) in each state, until the racecar reaches the finish line. The pseudocode for the Value Iteration algorithm is found in Figure 1.

Function VALUE-ITERATION (mdp, ϵ)
  Inputs: mdp, an MDP with states S, actions A(s), transition model $P(s'|s, a)$, rewards R(s), discount γ, ϵ the maximum error allowed in the utility of any state.

  Local variables: U, U', vectors of utilities for states in S, initially zero, δ the maximum change in utility of any state in an iteration

  Repeat
    U = U'
    $\delta = 0$
    for each state $s \in S$ **do**
      $U'(s) = R(s) + \gamma \max_{a \in A(S)} \sum_{s'} P(s'|s, a) U'(s')$
      if $[U'(s) - U(s)] > \delta$ **then**
        $\delta = [U'(s) - U(s)]$
      end if
    end for
  until $\delta < \epsilon(1 - \gamma)/\gamma$
  return U
  end function

Figure 1. Value Iteration Algorithm which uses the Bellman Equation to calculate utilities for each state.

2.5 Q-Learning

The Q-Learning algorithm learns an action-utility representation instead of learning utilities, and thus also attempts to learn $Q^*(s, a)$. Unlike Value Iteration, however, Q-Learning uses episodes, in which the agent makes a sequence of decisions from the beginning to the end, updating the policy along the way. This is in contrast to Value Iteration, which updates each state-action pair on each iteration. As a result, Q-Learning may be more efficient at learning a policy given a large state space. We note, however, that the policy only determines a, and the reward is estimated by the approximation of the optimal choice at state $Q^*(s, a)$. However, the agent does not necessarily make that optimal choice.

Q-Learning uses a ϵ-greedy action selection to explore a space. For a state s_t, the ϵ-greedy policy selects the optimal (as of time *t*) action with probability $1 - \epsilon$, while selecting a random action with probability ϵ. The pseudocode for the Q-Learning algorithm is found in Figure 2.

Function Q-LEARNING (percept)
  Inputs: percept, indicating the current state s' and reward signal r'

  Persistent: Q, a table of action values indexed by state and action initially 0; $N_{s,a}$ a table of frequencies for state action pairs; s, a, r, the previous state, action, and reward, initially null.

  if TERMINAL(s) **then**
    $Q[s, N\ one] \leftarrow r'$
  end if

  if s is not null **then**
    choose a via ϵ-greedy selection
    increment $N_{s,a}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha\left(r + \gamma\ arg\max_{a'} Q(s', a') - Q[s, a]\right)$
  end if
  $s \leftarrow s'$
  return a
  end function

Figure 2. Q-Learning Algorithm

2.6 SARSA

The final algorithm used for this project was SARSA (state-action-reward-state-action). Although very similar to Q-Learning, SARSA is an on-policy learning algorithm. Not only does it determine the immediate-most action *a*, but also the succeeding one *a'*. The pseudocode for the SARSA algorithm is found in Figure 3.

Function SARSA (percept)

Inputs: percept, indicating the current state s' and reward signal r'

Persistent: Q, a table of action values indexed by state and action initially 0; $N_{s,a}$ a table of frequencies for state action pairs; s, a, r, the previous state, action, and reward, initially null.

If TERMINAL(s) **then**
$Q[s, N\ one] \leftarrow r'$
end if

if s is not null **then**
choose a via ϵ-greedy selection
increment $N_{s,a}[s, a]$
$Q[s,a] \leftarrow Q[s,a] + \alpha\left(r + \gamma\ arg\max_{a'} Q(s', a') - Q[s,a]\right)$
end if
$s \leftarrow s'$
$a \leftarrow a'$
return a
end function

Figure 3. SARSA Algorithm

3 Experimental Approach

This project used 3 environments, represented as racetracks, for each experiment:

1. L-Shaped Track
2. O-Shaped Track
3. R-Shaped Track

All three algorithms (Value Iteration, Q-Learning, and SARSA) were applied to each environment and for two environmental variants (harsh and simple). Each algorithm performed a variable number of iterations, until performance converged to a solution with an appropriate number of steps (proportional to the racetrack's difficulty).

All algorithms used a discount factor, defined as the amount by which future reward is discounted backward to the present time step. Both Q-Learning and SARSA also used two additional hyperparameters, learning rate and ϵ. Here, learning rate was defined as the proportion of the temporal difference added back to the current sum of the expected reward in a given state. ϵ was defined as the likelihood of choosing a random action in each state. Additional comparisons were made by examining differences in the number of training iterations and the average number of steps to solve the problem.

Thus, each algorithm was compared using the following: racetrack, environment variant, discount factor, learning rate (Q-Learning and SARSA only), Number of Training Iterations, and Average Steps to Solve. In order to obtain reasonable statistics, 10 experiments per track were run and generated data was used to plot learning curves.

4 Experiment Results

4.1 Algorithm Results.

The results of the Algorithms can be found below in Table 1.

Discount Factor. Discount factor had no effect on Value Iteration.

Varying Discount Factor & Learning Rate. Varying discount factor and learning rate on Q-Learning and SARSA had no effect. The algorithms either converged to good policies after running for a couple hours or did not converge at all.

ϵ. Larger values of ϵ had the best effects.

Racetrack	Algorithm	Environment Variant	Discount Factor	Learning Rate	# Of Training Iterations	Average # of Steps to Solve
L-Track	Value Iteration	Harsh	.7	N/A	250	Did not converge
L-Track	Value Iteration	Simple	.8	N/A	100	61.7
L-Track	Q-Learning	Harsh	.5	.2	100,000	175.375
L-Track	Q-Learning	Simple	.5	.2	40,000	80.25
L-Track	SARSA	Harsh	.5	.2	125,000	152.75
L-Track	SARSA	Simple	.5	.2	4,000	133.225
O-Track	Value Iteration	Harsh	.7	N/A	1,000	544.925
O-Track	Value Iteration	Simple	.8	N/A	500	33.5
O-Track	Q-Learning	Harsh	.5	.2	175,000	915
O-Track	Q-Learning	Simple	.5	.2	75,000	75.22
O-Track	SARSA	Harsh	.5	.2	200,000	978
O-Track	SARSA	Simple	.5	.2	4,000	58.32
R-Track	Value Iteration	Harsh	.001	N/A	1,000	Did not converge
R-Track	Value Iteration	Simple	.001	N/A	500	Did not converge
R-Track	Q-Learning	Harsh	.3	.1	175,000	Did not converge
R-Track	Q-Learning	Simple	.3	.1	75,000	861
R-Track	SARSA	Harsh	.3	.2	200,000	Did not converge
R-Track	SARSA	Simple	.3	.1	75,000	980

Table 1. Experimental results comparing each algorithm's performance using hyperparameters discount factor, learning rate, number of training iterations, and average number of steps to solve on each racetrack and environment variant.

Racetrack Size. From the results in Table 1, we see that all algorithms behaved as expected. For small environments with simple-variants, all algorithms were able to solve the racetrack problem. However, as the environment size grew, it became more difficult

to solve. This is likely due to the fact that as the size of the racetrack grew, so too did the corresponding state space. As expected, it was observed that Value Iteration had difficulty converging on larger racetracks. Due to the fact that Value Iteration must visit every state-action pair, as environment size increases, it visits many irrelevant states and must also wait to finish before calculating reward.

Environment-Variant. Results indicate that the harsh-environment variant was difficult for all three algorithms, especially on the R-track. The only two which did converge on the R-track were both on the simple variant. Here, Q-Learning and SARSA took the same number of iterations, but SARSA look slightly longer to converge. The simple-variant L-track converged to good solutions for all three algorithms. The harsh-variant L-track converged to a good solution for Q-Learning and SARSA, but took many more training iterations compared to Value Iteration. The simple-variant O-track converged to good solutions for all three algorithms. Here, Value Iteration used the fewest training iterations and Q-Learning used the most. The harsh-variant O-track also converged to good solutions for all three algorithms. Here, Value Iteration used the fewest training iterations and SARSA used the most.

5 Behavior of Algorithms

Value Iteration. Figure 4 compares the average number of steps needed to solve each racetrack variant using Value Iteration. We see that, overall, Value Iteration had the most success on the L-track compared to all other tracks. Value Iteration did not converge on the R-track. And surprisingly, Value Iteration performed better for the simple-variant on the O-track.

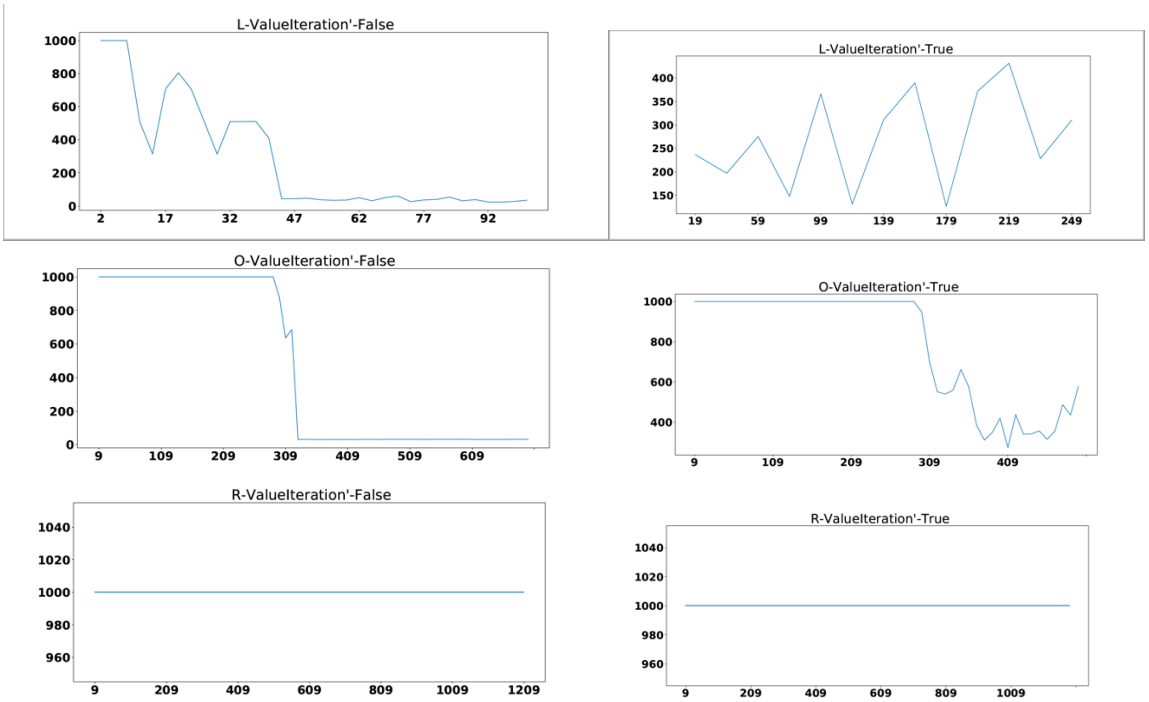

Figure 4. Value Iteration comparison across all racetracks.

Q-Learning. Figure 5 compares the average number of steps needed to solve each racetrack variant using the Q-Learning Algorithm. Overall, Q-Learning took the fewest number of training iterations on both environment variants, but took fewer steps on the O-track compared to the other two racetracks. Q-Learning was unable to converge on the harsh-R-track.

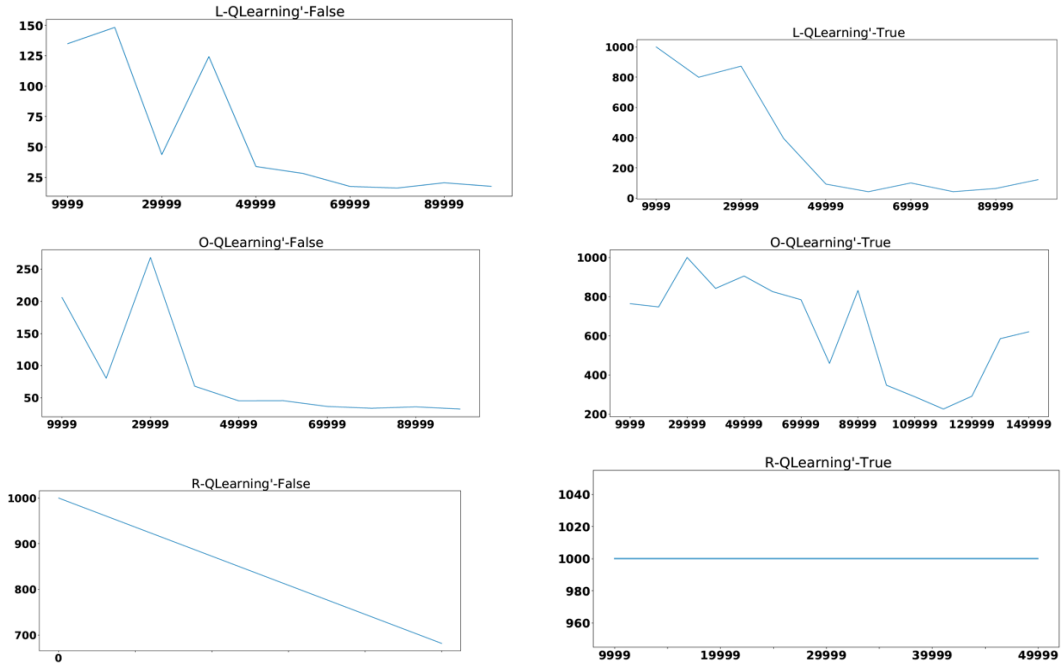

Figure 5. Q-Learning comparison across all racetracks.

SARSA. Figure 6 compares the average number of steps needed to solve each racetrack variant using the SARSA algorithm. Overall, SARSA performed best on the simple-variants of each racetrack. Interestingly, although SARSA took 4,000 iterations on both the simple-L-track and the simple-O-Track, SARSA was faster to converge on the Simple-O-track (taking 58.32 steps on average, compared to 152.75 steps on average).

6 Summary

This project provided students the opportunity to implement three different Reinforcement Learning algorithms: Value Iteration, Q-Learning, and SARSA. Here, we found that Value Iteration was a useful approach for simple, small state spaces but struggled with difficult problems and larger state spaces. Q-Learning and SARSA performed similar to one another, which makes sense because of their code similarities. Harsh-environment variants and non-deterministic systems made for longer and more difficult learning tasks.

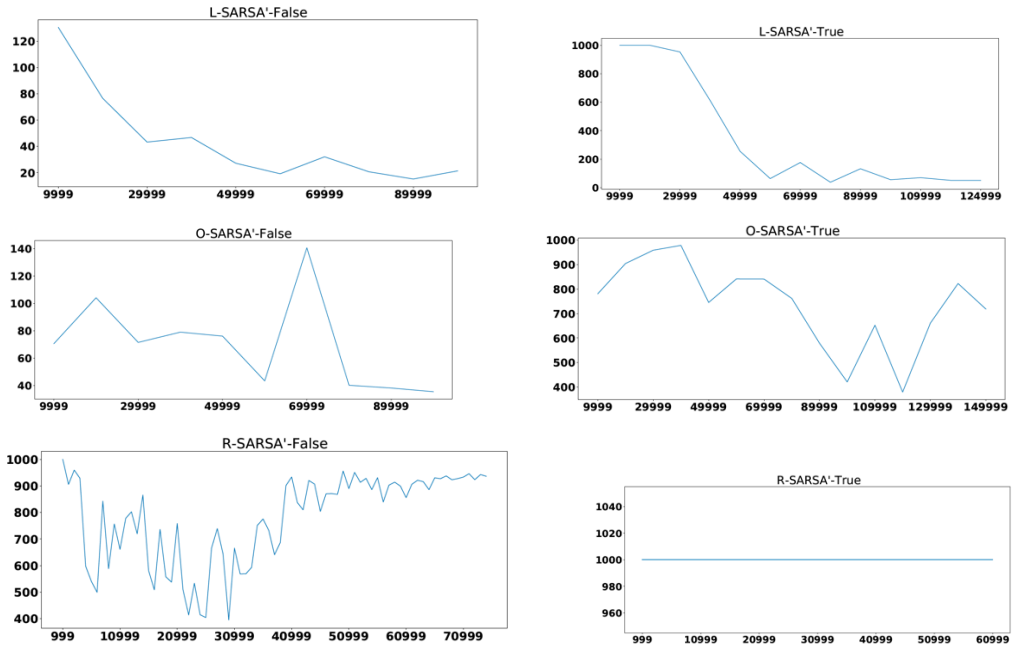

Figure 6. SARSA comparison across all racetracks.

References

Bellman, R. A. (1957). Markovian Decision Process. Indiana University Math Journal, 679-684.

Niranjan, M., & Rummery, G.A.(1994). On-line Q-Learning Using Connectionist Systems.

Melo, F.S. Convergence of Q-Learning: A Simple Proof.

Mitchell, T.M. (1997). Machine Learning. McGraw Hill.

Sutton, R.S., & Barto, A.G. (2020). Reinforcement Learning: An Introduction (2nd Ed). MIT Press, Cambridge, MA.