Reinforcement Learning

Jieyi Chen

December 13, 2020

1. Introduction

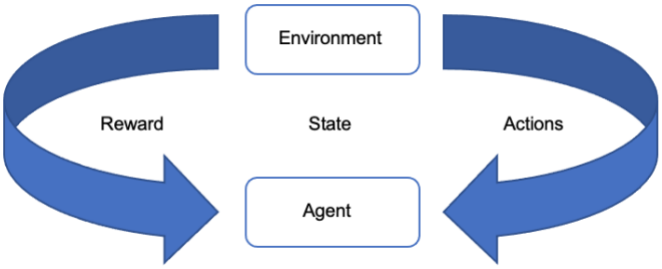
Reinforcement learning is a machine learning technique that teaches the software agent to find a sequence of best actions in an unknown environment with the goal of maximizing the total reward. Some popular applications are the control problems and games, but this technique can also apply to solve problems in numerous industries such as eCommerce, finance, robotics and manufacturing. The adaptability and no requirement for a large labeled dataset drive many attentions in this field because it opens up more opportunities to solve some "unsolvable" problems. In this project, we apply value iteration, Q-Learning and SARSA algorithms to teach an autonomous race car to travel from the starting position to the finish line with a minimum amount of time in a predefined racetrack without crashing into the walls. In addition, we will test two hypotheses: 1. The value iteration algorithm outperforms Q-Learning and SARSA in terms of time complexity because it is a model-based reinforcement learning algorithm. 2. Reset the race car to the original starting position of the racetrack cause more crashes than place it at the nearest position on the track to the place where it crashed because we hypothesize resetting the car to the starting position will require more trial and error in order to reach the final destination. To test these two hypotheses, we compare the number of crashes occurred when developing policies and the number of time trials needed to finish the race with the optimal policy for these three algorithms.

The rest of the paper is organized as follows. Section 2 describes the algorithms that are implemented. Section 3 discusses the experimental approach. Section 4 presents the results of the experiments. Section 5 discusses the results. Section 6 concludes.

2. Algorithm Implementation:

Reinforcement learning, a type of machine learning that allows the agent to explore in the environment by taking actions and learn based on the feedback/reward. The agent will be able to identify a series of actions that maximize the total rewards through trial and error.

2.1 Components of the reinforcement learning model:



Components & Definition of the Reinforcement Learning Model		
Components	**Definition**	**Example in this project**
Agent	An entity that takes actions and learns	Race car
Environment	The world that the agent interacts with and it provides feedback based on the agent's action	Racetrack
State	A representation that describes the environment where the agent is in at a particular point of time	<x position, y position, x velocity, y velocity> Maximum Velocity: 5 Minimum Velocity: -5
Actions	The move that the agent makes in a given state	< x acceleration, y acceleration> Available Actions: [[-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 0], [0, 1], [1, -1], [1, 0], [1, 1]]
Reward	The immediate feedback that the agent receives after it conducts an action in a given state	-1 for each move 0 for finish the game

Table 1: Components & Definition of the Reinforcement Learning Model

Policy (π): The decision-making process where the agent determines its action based on the current state. The function is $\pi(s) = a$.

Discount Factor: A discount factor γ that ranges from 0 to 1 and is used to calculate the value of the future reward at the current state

Value Function: A function that calculates the expected reward that the agent will receive if it starts in state S_0 and follows policy (π)

$$V_\pi(s) = E(R|s_0 = s) = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$$

Q -Value Function: A function that calculates the expected reward that the agent will receive if it starts in state S_0, takes action a and follows policy (π)

$$Q_\pi(s, a) = E(R|s, a, \pi) = E[\sum_{t=0}^{\infty} \gamma^t r_t | s, a, \pi]$$

There are two versions of the reinforcement learning problems: 1) Deterministic: all actions yield the expected behavior. 2) Non-deterministic: most but not all actions yield the expected behavior due to noise. For this project, we add a small amount of non-determinism where there is a 20% chance that each attempted action for each possible state will fail and the velocity will remain the same at the next timestep. There is an 80% probability that the race car accelerates as specified.

In order to minimize the number of explorations required to learn the policy, the agent needs to optimize a value function that represents the state of the problem. There are three ways to represent

the associated value function, assuming there is probability distribution over transitions. For this project, we use the discounted value function.

- Finite Horizon

$$V^{\pi}(s_t) = E\left[\sum_{i=0}^{k} r(s_{t+i}, a_{t+i})\right]$$

- Discounted

$$V^{\pi}(s_t) = E\left[\sum_{i=0}^{\infty} \gamma^i r(s_{t+i}, a_{t+i})\right]$$

- Average Reward

$$V^{\pi}(s_t) = \lim_{k \to \infty} E\left[\frac{1}{k}\sum_{i=0}^{k} r(s_{t+i}, a_{t+i})\right]$$

2.2 Markov Decision Process

Markov Decision Process was invented by Richard Bellman in the late 1950s and it is defined in the following components [1].

- S: a set of states for the environment
- A: a set of actions available to the agent
- T: a state transition function $\pi(s)$, which is the probability of transitioning to state s'given that the agent is currently in state s and applies action a. $T(s, a, s') = P(s'|s, a)$
- R: a reward function that indicates the reward receive when we take action a while we are in state s.
- γ: a discount factor that is in the range of 0 and 1

Using the following greedy policy, we can determine the optimal action based on the current state.

$$\pi(s) = \underset{a \in A}{\operatorname{argmax}}[r(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V^{\pi}(s')]$$

2.3 Value Iteration

Value Iteration is a dynamic programming algorithm that computes a sequence of value function V_t using $Q_t(s, a)$, which estimates the value of taking action a in state s and acting from that point forward according to the greedy policy derived in the current value function. The V_t and Q_t are updated iteratively based on the prior estimates until the maximum differences between two successive estimates drop below the Bellman Error Magnitude.

2.3.1 Algorithm:

1. Initialize the values for all the states to be 0
2. Repeat the following steps until the difference between two successive value functions V_t drop below the Bellman Error Magnitude: $\max_{s \in S} |V_t(s) - V_{t-1}(s)| < \epsilon$

a. For each state and action pair, calculate the $Q_t(s,a)$
    i. $Q_t(s,a) \coloneqq R(s,a) + \gamma \sum_{s' \epsilon S} T(s,a,s') V_{t-1}(s')$
b. Find the optimal action for each state
    i. $\pi_t(s) \coloneqq \underset{a \epsilon A}{\operatorname{argmax}} Q_t(s,a)$
    ii. $V_t(s) \coloneqq Q_t(s, \pi_t(s))$

3. Output the optimal policy: π_t

2.4 Q-Learning and SARSA

Q-Learning and SARSA are model free reinforcement learning algorithms where the states, actions, associated rewards and transitions to the resulting states are unknown. The agent learns based on temporal difference learning, which defines as using differences in prediction over successive time steps. With the partial knowledge, the greedy policy would take the action that will maximize the total reward whereas the ε- Greedy Policy would find the optimal balance between exploitation and exploration. The agent has a probability of ε to explore the unknown states by conducting random action and a probability of 1- ε to exploit by choosing action with the highest Q-value in the table.

$$\pi(s) = \begin{cases} \underset{a \epsilon A}{\operatorname{argmax}} Q(s,a) & with\ probability\ (1-\varepsilon) \\ random\ action & with\ probability\ \varepsilon \end{cases}$$

The difference between Q-Learning and SARSA is that Q-learning is off-policy and SARSA is on-policy. When the agent interacts with the environment, it implements behavior and target policy. Behavior policy is the strategy that the agent uses to determine its action at a given state and the target policy is the policy that the agent uses to update the Q values. The on-policy learner means that the behavior policy and the target policy are identical. The off-policy is defined as the situation when the behavior policy and the target policy are different.

2.4.1 Q-Learning Algorithm

1. Initialize all the values in the Q table to be 0
2. Repeat the following step for a fixed number of iterations:
    a. Randomly select an initial state from all the available starting positions to begin the game
    b. While not reaching the terminal state:
        i. Choose an action a with the highest Q value from the Q table given the current state
        ii. Take action a, observe reward r and next state s'
        iii. Update Q with:
        $$Q(s,a) \coloneqq Q(s,a) + \alpha[r + \underset{a \epsilon A}{\operatorname{argmax}} \gamma Q(s',a') - Q(s,a)]$$
        iv. Update the current state: s := s'
        v. check whether we reach the terminal state

2.4.2 SARSA Algorithm:

1. Initialize all the values in the Q table to be 0
2. Repeat the following step for a fixed number of iterations:

a. Randomly select an initial state from all the available starting positions to begin the game
b. While not reaching the terminal state:
    i. Choose an action a with the highest Q value from the Q table given the current state
    ii. Take action a, observe reward r and next state s'
    iii. Update Q with:
$$Q(s,a) := Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$
    iv. Update the current state: s := s'
    v. check whether we reach the terminal state

2.4.5 Bresenham's Algorithm:

Bresenham's Algorithm is a line drawing algorithm that finds the points required to draw a straight line between point A and point B. In our case, we want the agent to start from state s, conduct action a and reach state s'. This algorithm can help us to determine the path between states and we implement it using the following pseudocode code provided by Wikipedia [2].

```
plotLineLow(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    yi = 1
    if dy < 0
        yi = -1
        dy = -dy
    end if
    D = (2 * dy) - dx
    y = y0

    for x from x0 to x1
        plot(x, y)
        if D > 0
            y = y + yi
            D = D + (2 * (dy - dx))
        end if
        else
            D = D + 2*dy
```

```
plotLineHigh(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    xi = 1
    if dx < 0
        xi = -1
        dx = -dx
    end if
    D = (2 * dx) - dy
    x = x0

    for y from y0 to y1
        plot(x, y)
        if D > 0
            x = x + xi
            D = D + (2 * (dx -
        end if
        else
            D = D + 2*dx
```

```
plotLine(x0, y0, x1, y1)
    if abs(y1 - y0) < abs(x1 - x0)
        if x0 > x1
            plotLineLow(x1, y1, x0, y0)
        else
            plotLineLow(x0, y0, x1, y1)
        end if
    else
        if y0 > y1
            plotLineHigh(x1, y1, x0, y0)
        else
            plotLineHigh(x0, y0, x1, y1)
        end if
    end if
```

3. Experimental Approach:

3.1 Datasets:

The datasets used for this project are three provided tracks: L-track, O-track and R-track. The data describing the tracks are represented in ASCII. The first line in the file lists the size of the track. The rest of the file is a Cartesian grid of the specified dimensions with one character at each point. 'S' represents the starting line of the racetrack. 'F' indicates the finish line of the racetrack. '.' represents the open racetrack. '#' indicates the wall in the racetrack. No data preprocessing steps are necessary for this project. We eliminate the first line that indicates the dimension of the racetrack and save the grid in a numpy array. The agent can randomly select a square with a 'S' to start the game and ends a game by reaching a square with a 'F'.

3.2 Hyperparameter Tuning Process:

The hyperparameters for the value iteration algorithm are:

- Discount Rate: $\gamma \in [0, 1]$

- Bellman Error: $\varepsilon_{Bellman} = \{0.01\}$
- Crash Type: {1: Reset the car's position to the original starting position of the racetrack, 2: Reset the car's position to the nearest position on the racetrack to the place where it crashed}
- Number of training iterations

The hyperparameters for the Q-Learning and SARSA algorithm are:

- Discount Rate: $\gamma \in [0, 1]$
- ε-greedy: $\varepsilon_{Greedy} \in [0, 1]$
- Learning Rate: $\eta \in [0, 1]$
- Crash Type: {1: Reset the car's position to the original starting position of the racetrack, 2: Reset the car's position to the nearest position on the racetrack to the place where it crashed}
- Number of training iterations

4. Result

Number of crashes occurred when optimizing the policy Hyperparameter: γ = 0.9, $\varepsilon_{Bellman}$ = 0.1, Crash Type = 1, ε_{Greedy} = 0.1, η = 0.01			
	Value Iteration	Q-Learning	SARSA
L-track	5,288	38,577	37,697
O-track	7,597	219,615	265,306
R-track	9,962	1,819,673	1,961,843

Table 2: Number of crashes occurred when optimizing the policy for crash type 1

Number of crashes occurred when optimizing the policy Hyperparameter: γ = 0.9, $\varepsilon_{Bellman}$ = 0.1, Crash Type = 2, ε_{Greedy} = 0.1, η = 0.01			
	Value Iteration	Q-Learning	SARSA
L-track	9,897	8,819	9,081
O-track	10,123	4,207	7,555
R-track	9,619	33,127	39,644

Table 3: Number of crashes occurred when optimizing the policy for crash type 2

Number of time trials needed to finish the race with the optimal policy Hyperparameter: γ = 0.9, $\varepsilon_{Bellman}$ = 0.1, Crash Type = 1, ε_{Greedy} = 0.1, η = 0.01			
	Value Iteration	Q-Learning	SARSA
L-track	513	1,220	165
O-track	56	454	31
R-track	2,489	3,266	27,059

Table 4: Number of time trials needed to finish the race with the optimal policy for crash type 1

Number of time trials needed to finish the race with the optimal policy Hyperparameter: γ = 0.9, $\varepsilon_{Bellman}$ = 0.1, Crash Type = 2, ε_{Greedy} = 0.1, η = 0.01			
	Value Iteration	Q-Learning	SARSA
L-track	13	31	15
O-track	18	32	3
R-track	63	71	199

Table 5: Number of time trials needed to finish the race with the optimal policy for crash type 2

5. Discussion:

Table 2 and 3 show the number of crashes occurred when optimizing the policy in the value iteration, Q-Learning and SARSA algorithms. Table 4 and 5 show the number of trials needed to finish the race with the optimal policy. The value iteration algorithm outperforms the Q-Learning and SARSA in terms of time complexity because the number of trials needed to finish the race with the optimal policy are the lowest for the value iteration. The results for Q-Learning and SARSA are mixed depend on the type of the track. In addition, we show that resetting the race car to the original starting position of the racetrack cause more crashes than place it at the nearest position on the track to the place where it crashed. Therefore, we prove that all of our hypotheses are correct.

6. Conclusions:

This project helps us to gain a better understanding of three reinforcement learning algorithms, which includes value iteration, Q-Learning and SARSA. This paper can help data scientists to implement these algorithms from scratch and decide which one is more appropriate under different scenarios. Some future work on how to optimize the algorithm in terms of time complexity can be done. We can also tune more hyperparameters to achieve better result.

7. Reference:

[1] BELLMAN, RICHARD. “A Markovian Decision Process.” Journal of Mathematics and Mechanics, vol. 6, no. 5, 1957, pp. 679–684., www.jstor.org/stable/24900506. Accessed 13 Dec. 2020.

[2] “Bresenham's Line Algorithm.” Wikipedia, Wikimedia Foundation, 21 Nov. 2020, en.wikipedia.org/wiki/Bresenham's_line_algorithm.