Performances of different Reinforcement Learning Methods in the Mountain-Car GYM Environment

Ahmed Ahmed [1] **Jianwei An** [1] **Seyedamiryousef Hosseini** [1]

Abstract

We used two approximate solution methods of reinforcement learning to tactile the Mountain Car Task provided by OpenAI Gym. The main objective of the Mountain Car environment is to get the agent (car) to reach the top of the mountain (flag marked as the goal as shown in Figure 1). Two main learning algorithms used for training the agent are episodic semi-gradient sarsa and Sarsa(λ) with accumulating traces. For each learning algorithm, we investigated its learning performances under different hyper-parameters' settings and plotted the learning curves in each case. Since the Mountain Car Task has a two-dimensional continuous state space, in which there are infinite different states, state generalization methods should be adopted. We were able to demonstrate the effectiveness of Tile coding.

1. Introduction

The mountain car problem appeared first in Andrew Moore's PhD Thesis (1990). The Mountain Car MDP is a deterministic MDP in which a car is stochastically put in the bottom of a sinusoidal valley, with the only actions available being accelerations in either direction. The MDP's purpose is to strategically accelerate the car to the goal state on the right hill's summit. By simply accelerating forward, the car will not be able to reach the summit of the slope. To win, the car must gain momentum by swinging back and forth until it reaches the finish line at a sufficient speed.

The car serves as a reinforcement learning agent for us. It interacts with the environment by performing actions. The environment is straightforward and simple, and we only need to know two things about the car at any given time: its location and speed. This problem, like its state space, is two-dimensional.

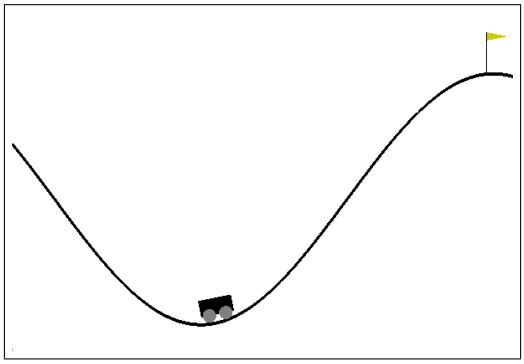

Figure 1. Mountain-Car-v0 Environment - OpenAi Gym

At any one time, the car can only take one of three actions: accelerate ahead, reverse, or do nothing. The environment will return to a new state whenever the agent executes an action. Let's say the car starts in the middle, with a position of 0 and a velocity of 0. That is how it started. The agent may then opt to accelerate forward, and will inform the environment of this decision. The environment will return a new state, such as position 1, velocity 1, or something similar. Our model's purpose is to figure out what activities should be taken in each state to assist it attain the flag.

Whenever the car approaches the flag, it will be rewarded. But, we shouldn't reward the model simply when it reaches the flag, as this would make learning harder for it. We can also give the agent a bonus at each state.

Our car will try to assess "how good" each action is in order to learn what actions to take at each step. The value of a state can be used to quantify how good an action is. A state's value is equal to the total of all its future projected rewards. The model believes that the higher the value of a state, the greater the reward it will earn. The purpose of reinforcement learning models is to approximate this value function as closely as possible. As a result, the car can consider all of its options (forward, backward, or not accelerate) in every given state. It can then calculate the future states' predicted values and choose the optimal action. However, selecting an action is more complicated than simply selecting the action with the highest predicted value. A reinforcement learn-

[1]Department of Computer Science, University of Victoria, Victoria, Canada. Correspondence to: Ahmed Ahmed <ahmedahmed@uvic.ca>, Jianwei An <jianweian@uvic.ca>, Seyedamiryousef Hosseini <yousofhosseini9877@Uvic.ca>.

RL Course Project Report, University of Victoria, Spring, 2022.

ing model's policy is the reasoning that determines which action to take. It is impossible to always choose the most valuable activity. This is because if our value function is incorrect, we will significantly impede our model's learning. Exploration (trying out new things) and exploitation must be balanced (taking the action with highest value). There are two techniques to determine the value function. The tabular discretization method and gradient descent methods are both used to accomplish this. The purpose of tabular discretization is to learn all potential states' values in a table format. All conceivable states will be listed in the table. Tabular discretization is simple to learn and effective. However, because this is a straightforward problem, the state space can be discretized with ease. Keeping the state space in memory may be unfeasible for more difficult tasks. Gradient descent methods are used to solve this problem. We learn a function to map the state space to the value approximation in gradient descent approaches. This is easily accomplished with a linear model by taking the dot product of the state and the weights of your model. In the gym, there are two different versions of the mountain car domain: one with discrete activities and one with continuous actions.

2. Related techniques used to solve the Mountain Car problem

To be able to cope with the continuous state space of the problem, Q-learning and comparable techniques for mapping discrete states to discrete actions need to be extended. State space discretization or function approximation are two approaches that are frequently used.

Two continuous state variables are pushed into discrete states in the discretization approach by bucketing each continuous variable into several discrete states. This method works with appropriately tuned parameters, but it has the drawback of not allowing information from one state to be utilised to evaluate another. Continuous variables are mapped into sets of buckets offset from one another in tile coding, which can be used to improve discretization. Because the information is spread when the offset grids are summed, each step of training has a larger impact on the value function approximation.

Another technique to solve the mountain car is to use function approximation. The agent can approximate the value function at each state by selecting a set of basis functions ahead of time or generating them as the car drives. Function approximation, as opposed to the step-wise form of the value function obtained by discretization, can more accurately estimate the true smooth function of the mountain car domain.

An interesting aspect of the problem involves the delay of actual reward. The agent isn't able to learn about the goal until a successful completion. Given a naive approach for each trial the car can only backup the reward of the goal slightly. This is a problem for naive discretization because each discrete state will only be backed up once, taking a larger number of episodes to learn the problem. This problem can be alleviated via the mechanism of eligibility traces, which will automatically backup the reward given to states before, dramatically increasing the speed of learning. Eligibility traces can be viewed as a bridge from temporal difference learning methods to Monte Carlo methods.

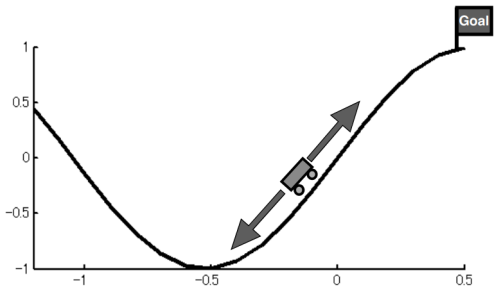

Figure 2. (Sugiyama, 2015)

3. The Environment of Mountain Car Task

In the simplified Mountain Car Task we want to investigate, although the position x_t and velocity v_t which would represent the state of the car at time t are all the continuous variables, there are only 3 possible actions could be chosen at each time. Specifically, $A_t \in \{0, 1, 2\}$ where A_t is the deterministic action the agent could choose at time t. $A_t = 0$ means that the car would fully accelerate to the left, $A_t = 1$ means that the car would not accelerate and $A_t = 2$ means that the car would accelerate to the right. The car moves according to the following simplified physics.

$$x_{t+1} = \textit{bound}[x_t + v_{t+1}]$$
$$v_{t+1} = \textit{bound}[v_t + 0.001(A_t - 1) - 0.0025\cos(3x_t)]$$

where *bound* operation enforces $-1.2 \leq x_t \leq 0.6$ and $-0.07 \leq v_t \leq 0.07$ for all t. In addition, at any time t when x_t reached the left bound, v_t was reset to zero. When it reached the right bound, the goal was reached and the episode was terminated. Each episode started from a random position $x_0 \in [-0.6, 0.4)$ and zero velocity.

Fortunately, the OpenAI Gym has already provided us with the environment of this simplified Mountain Car Task. It's worth noting that the maximum length of the episode is 200 in this special environment. That is to say, the minimum total return of each episode is -200. And if in some episodes

whose total return G are larger than -200, the car would reach its goal in these episodes.

4. Two Linear methods

For each given specific position and velocity at time t, a *position-velocity* pair (x_t, v_t) could be formed to represent the state of the car at time t. Notice that (x_t, v_t) is in the two-dimensional continuous state space where there are infinite possible states, we cannot expect to find the optimal policy or the optimal value function for each state. Thus, we need to find some approximate solution methods in which the agent's experience with a limited subset of the state space could be generalized to produce a good approximation over a much larger subset.

In our project, two linear methods, in which the approximation function of each state-action pair is the linear function, are adopted to handle the Mountain Car Task. For both of them, tile coding would be utilized to construct the required representing vectors for the linear models. Before introducing these two linear methods in detail, we would first go to explain how the tile coding in our project would work.

4.1. Tile Coding

Our tile coding process mainly consists of two parts. First, we would use several grid-tilings to partition the two-dimensional position-velocity space, each of which would split the space in its own way. The same type of partitions could be reduced into one. As a simple example shown in Figure 3, each tiling must cover the whole position-velocity space so that every the possible state would sit in its unique tile in corresponding tiling. For a given state (x_t, v_t) at time t in the position-velocity space, each tiling could be considered as a binary vector whose length is the total number of its tiles. Because each specific state (x_t, v_t) can only sit in one tile in every tiling, each binary vector only have one position to be 1 and all the others are 0.

For example, in the Figure 3, the red point state s sits in the different tile in the two different grid-tilings. Both of these two tilings have 12 tiles. Then, two different binary vector $T_1(s)$ and $T_2(s)$ are generated according to the state's position in corresponding tilings. Because we only have 2 tilings in this simple example, the unique feature vector to represent the state s is $(T_1(s)^T, T_2(s)^T)^T$.

Besides mapping each state in the position-velocity space to its corresponding unique feature vector derived from tilings, in the second part of our tile coding process, we would combine each state's feature vector with different actions(i.e., $A_t \in \{0, 1, 2\}$) in order to map each state-action pair to a binary representing vector.

Because the agent can only choose three discrete actions in total, the mapping process is straightforward. Notice that all the state's feature vector would have the exactly same length. Thus, the representing vector of each state-action pair would be divided into three parts with the same length, which is also equal to the length of the state's feature vector.

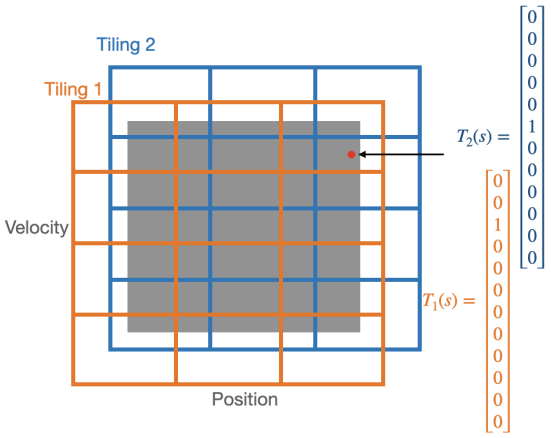

Figure 3. Two binary vector $T_1(s)$ and $T_2(s)$ are generated according to index of tile where the state s sits in different grid-tilings. In this simple tile coding whose total number of tilings is 2, the unique feature vector to represent state s is $(T_1(s)^T, T_2(s)^T)^T$

$$A_t = 0 : [T_1(s)^T, T_2(s)^T | 0,0,...,0 | 0,0,...,0]^T$$

$$A_t = 1 : [0,0,...,0 | T_1(s)^T, T_2(s)^T | 0,0,...,0]^T$$

$$A_t = 2 : [0,0,...,0 | 0,0,...,0 | T_1(s)^T, T_2(s)^T]^T$$

Figure 4. Different representing vectors for the state-action pairs where the specific state's feature vector is fixed but actions vary.

For example, in Figure 4, when the state's feature vector is fixed, different actions would lead to the different representing vectors. To be specific, if the agent at state s takes the action $A_t = 0$ at time t, the first $\frac{1}{3}$ part of this state-action pair's representing vector would be exactly the same as its state's feature vector and the rest $\frac{2}{3}$ part of the representing vector would be 0. Similarly, as what's illustrated in Figure 4, the middle $\frac{1}{3}$ is reserved for taking the action $A_t = 1$ and the final $\frac{1}{3}$ for taking the action $A_t = 2$ at state s.

By these two main processes in the tile coding, we actually map each possible state-action pair to its unique representing vector. After that, all the preparations for our two linear methods are done. It's worth noting that we could adjust the number of the partitions on each tiling's horizontal side and vertical side to control the length of each state's feature

vector. Combining that with adjusting the number of total grid-tilings, we could control the density of the partitions on the whole velocity-position space. For convenience and in order to simplify the discussion, in our project, **the total number of grid-tilings in our tile coding is fixed to be 8**.

Although we don't go to investigate the optimal partitions on this two-dimensional state space for Two linear methods, we do want to see how the number of partitions on each tiling's both sides in tile coding to influence the learning process of the *semi-gradient one-step sarsa algorithm*. We would show the influence in the next section.

4.2. Semi-gradient Sarsa in the Mountain Car Task

In the first part of our experiments, our reinforcement learning agent would adopt the *episodic semi-gradient one-step sarsa*, which is an on-policy control method with function approximation, to learn the proper policy for achieving the Mountain Car Task. The sudo-code and detailed explanation of this algorithm have included in the section 10.1 of (Sutton & Barto, 2018).

Recall that in this special OpenAI mountain-car environment, the maximum steps in each episode is 200. In other words, if the car cannot achieve its goal at the 200th step in one episode, this episode would terminate and the next new episode would begin. Thus, the minimum total return is -200. When the total return $G > -200$, $|G|$ is the number of steps which the car takes from its start state to the goal. Furthermore, in all of our experiments, it's the rolling average return whose **rolling average width is 91** that would be treated as the standard to illustrate the performance of the learning curves.

4.2.1. Tile Coding with different tile sizes

In the beginning, we would see how the number of partitions on each tiling's both sides in tile coding to influence the learning process of the episodic semi-gradient sarsa algorithm. In this experiment, $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithms and the step size $\alpha = 0.1/8$. Each learning curve is averaged over 5 runs, each of which contains 3000 episodes.

In Figure 5, you can see that the tile sizes of each tiling is represented by a tuple(i.e., $(13, 11)$). The first(second) number minus 1 in this tuple is the number of partitions on the interval of the position(speed) space. That's to say, if the tile sizes is $(13, 11)$, each grid-tiling on the horizontal(vertical) side would be **equally** divided into $13(11)$ parts and each grid-tiling would contain $13 \times 11 = 143$ tiles in total.

The experimental results in Figure 5 illustrates that the tiling code with more tiles in each grid-tiling would make the agent reach the proper policy to achieve its goal later. That makes sense. More tiles in each tiling means the less generalization on the position-velocity space. It would cost more time for the agent to find the proper policy to achieve its goal when the total number of the different representing vectors of all the state-action pairs is larger. Then, in rest of our experiments, **the tile sizes of each grid-tiling is fixed to be** $(13, 11)$.

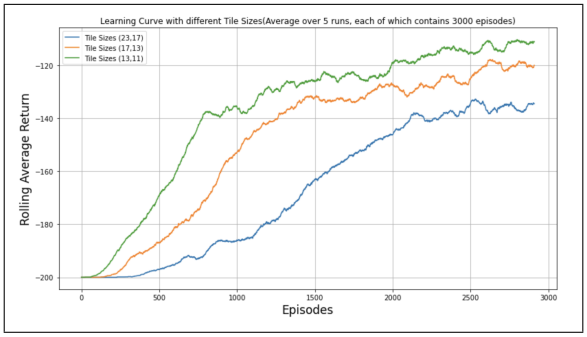

Figure 5. Semi-gradient one-step Sarsa with different Tile Sizes. Step size $\alpha = 0.1/8$, $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithms. Each learning curve is averaged over 5 runs, each of which contains 3000 episodes.

4.2.2. Different Performances Under different parameters' settings

In this section, we come to compare the different performances of the *episodic semi-gradient one-step sarsa* under different parameters' setting. First, we would like to see how the different settings of step sizes would influence the learning curves. In this experiment, $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithms. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

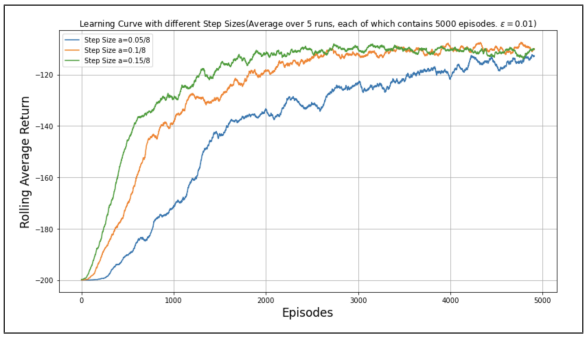

Figure 6. Semi-gradient one-step Sarsa with different Step Sizes α. $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithms. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

The learning curves in Figure 6 clearly show the different performances of the semi-gradient sarsa under different settings of the step sizes. When step size $\alpha \in [0.05/8, 0.15/8]$, the larger step size would make the agent obtain its proper policy earlier. However, it would still cost at least around

2000 episodes for the agent to obtain its proper policy. Then, in the second part of this section, the step size would be fixed to $\alpha = 0.15/8$ and we would vary the ϵ in utilized ϵ-greedy algorithms.

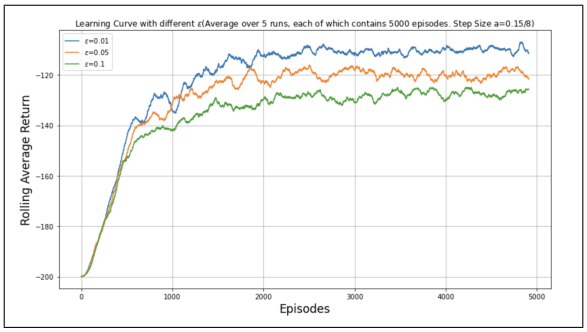

Figure 7. Semi-gradient one-step Sarsa with different ϵ utilized in its ϵ-greedy algorithm and the step size $\alpha = 0.15$. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

According to the results illustrated in Figure 7, it shows that the agent with larger ϵ would have smaller rolling average return after it obtains the proper policy. This still makes sense. After getting the stable policy, the agent with larger ϵ would have more time not to choose the proper policy. It would take more steps for the agent with larger ϵ to reach its goal.

After discussing the different performances of the episodic semi-gradient one-step sarsa under different parameters' settings, we would come to consider the next linear method—*Sarsa*(λ), which would use the eligibility trace in the learning process in the following section.

4.3. Sarsa(λ) in the Mountain Car Task

In the second parts of our experiments, our reinforcement learning agent would adopt the Sarsa(λ) in which the eligibility trace method would use the **accumulating traces**. The sudo-code and detailed explanation of the Sarsa(λ) with accumulating traces are all in the section 12.7 of (Sutton & Barto, 2018).

Similarly, we would compare the different performances of the Sarsa(λ) under different parameters' settings. In our first experiment with Sarsa(λ), we want to see how the different step sizes would influence the learning performances of the agent employing Sarsa(λ) where $\lambda = 0.5$, $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithm. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

Based on the results illustrated in Figure 8, it's clear to see that when the step size $\alpha \in [0.05/8, 0.15/8]$, the larger step size would help the agent to obtain the proper policy to achieve its goal earlier. Especially, one important phenomenon should not be ignored. When $\alpha = 0.15/8$, it would only cost around 1000 episodes for the agent employing Sarsa(λ) where $\lambda = 0.5$ to learn the proper policy.

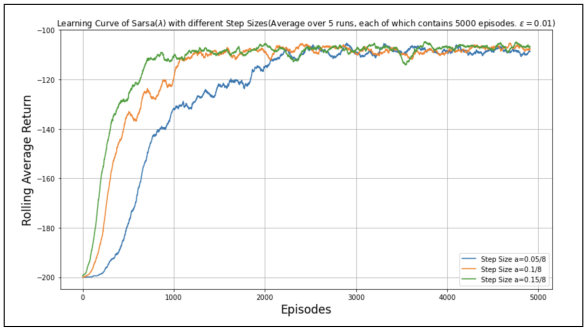

Figure 8. Sarsa(λ) with different Step sizes α. $\lambda = 0.5$, $\epsilon = 0.01$ for all the utilized ϵ-greedy algorithm. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

Recall that it's around 2000 episodes that the agent with episodic semi-gradient one-step sarsa should experience to obtain its proper policy under the similar parameters' settings($\alpha = 0.15/8, \epsilon = 0.01$). Compared with the semi-gradient one-step sarsa in similar parameters' settings, Sarsa(λ) with accumulating traces would reduce the learning time of the agent to get its proper policy to achieve its goal.

Notice that different from the semi-gradient sarsa, Sarsa(λ) has one more parameter λ, which could be adjusted to possibly influence the learning performances of the agent. In our final experiment with Sarsa(λ), we would see how the different settings of λ would influence the learning performance of our agent. In this test, Step size $\alpha = 0.15/8$, $\epsilon = 0.01$. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

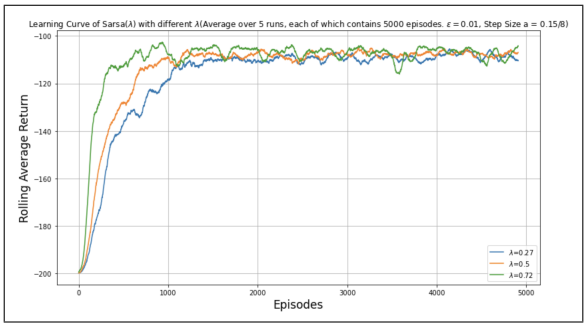

Figure 9. Sarsa(λ) with different λ. Step size $\alpha = 0.15/8$, $\epsilon = 0.01$ for all utilized ϵ-greedy algorithm. Each learning curve is averaged over 5 runs, each of which contains 5000 episodes.

The results are shown in Figure 9. When $\lambda \in [0.27, 0.72]$, the larger λ would make the agent learn the proper policy faster. At the meantime, however, the larger λ would cause more fluctuations on the part of learning curve where the agent has obtained the proper policy to achieve the goal.

One crucial phenomenon must be mentioned in varying λ to test different performances. When we set $\lambda = 0.96$ and all the other parameters are remained, the learning process would be interrupted because the ϵ-greedy algorithm cannot select an action based on the list whose elements are all '*nan*'. In other words, in our framework of Sarsa(λ) with accumulating traces, some settings of λ would make the value function approximation of some state-action pairs go to ∞(or $-\infty$) during the learning process so that Sarsa(λ) would lose it power. Thus, we should pay more attention to the valid range of λ when we want to employ the Sarsa(λ) with accumulating traces in the reinforcement learning.

5. Conclusion

In our project, with the help of tile coding, two reinforcement learning(RL) methods *episodic semi-gradient one-step sarsa* and *Sarsa*(λ), in which the approximated value functions are the linear functions, are employed to handle the simplified Mountain Car Task whose action space only contains 3 possible choices. For each RL method, we investigated its learning performances under different parameters' settings and gave the analyses for them.

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Moore, A. W. Efficient memory-based learning for robot control. Technical report, 1990.

Sanghi, N. *Deep reinforcement learning with python: With PYTORCH, tensorflow and Openai Gym*. Apress, 2021.

Singh, S.P., S. R. Reinforcement learning with replacing eligibility traces. *Machine Learning 22*, pp. 123–158, 1996. doi: 10.1023/A:1018012322525.

Sugiyama, M. *Statistical reinforcement learning: modern machine learning approaches*. CRC Press, 2015.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

xiong XU, Z., CAO, L., liang CHEN, X., xi LI, C., liang ZHANG, Y., and LAI, J. Deep reinforcement learning with sarsa and q-learning: A hybrid approach. *IEICE Transactions on Information and Systems*, E101.D(9): 2315–2322, 2018. doi: 10.1587/transinf.2017EDP7278.