林柏均 107062240

March 28, 2021

Deep Reinforcement Learning

Homework 1

Problem 1

I defined 3 classes : Transition, Policy, Grid world. Below will explain what these classes' property in detail.

Transition

probability : probability of transition to other state (set as 0.25)

next_state : id of the next state by this transition

reward : the reward after this transition (set as -1)

Policy

id : id of the given id in grid world

transitions : store all possible transition property

initial_transitions() : function for initialize the transitions array

Gridworld

gridworld : : array for every grid, whose type is *Policy*

initial_gridworld() : initial every gird with a specific *Policy* property

policy_iteration() : this function applied policy iteration pseudocode

show_states() : if the iteration is over show the final convergent gird world

save_states() : save learning policy data

According to the final output, I found that after about 2000 iterations, the policy would converge and the value of state won't change anymore. And we also need to change the gamma value of the policy update equation.

gamma = 0.1 : we notice that the converge value is quite small about -1

gamma = 0.9 : we notice that the converge value is smaller than former about -7

Based on the observation above, when gamma's value becomes greater, the final convergent value would be more smaller, the next state's value would become more affect for the current state.

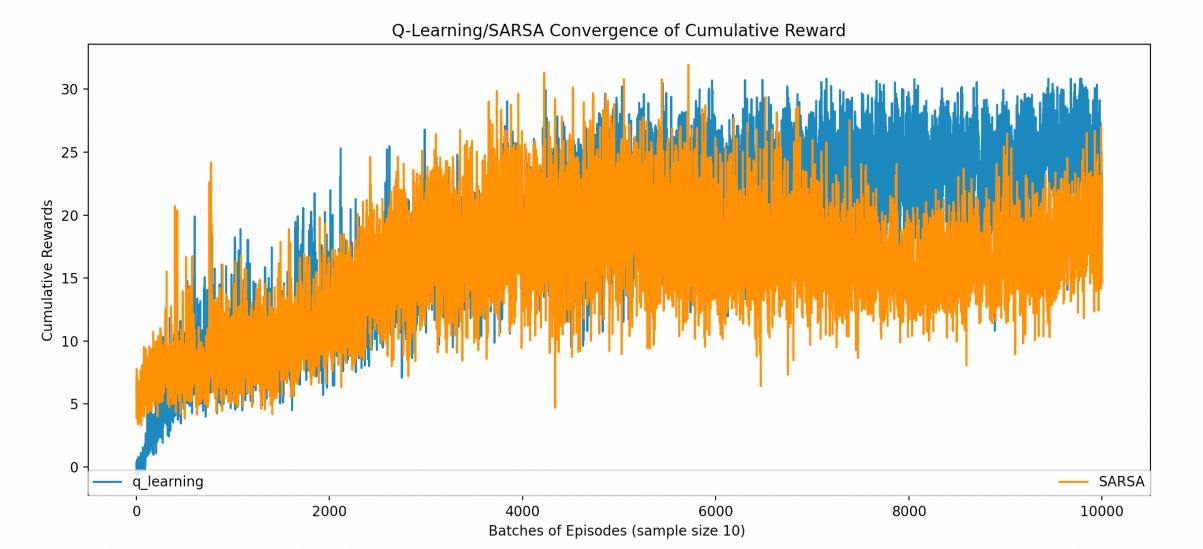
Problem 2

In this problem, I selected "CartPole-v1" from gym environment as my learning environment. I have to admit that this environment seem to be quite simple, yet I spent lots of time researching the parameters. Below I would explain how do I implement Q-Learning and SARSA in this environment.

Compared with Cliff Walking problem, the environment I used was more complicated, in Cliff Walking problem, we will create a 2-dimension q-table, however, in CartPole-v1, I created a 5-dimension q-table (car position, car velocity, Pole Angle, Pole Velocity and action). Below I briefly show the steps of my program :

1. Initialize environment "CartPole-v1"
2. Initialize q-table
3. Apply this environment with SARSA algorithm(100000 episodes)
4. Apply this environment with Q-Learning algorithm(100000 episodes)
5. When implementing 3 and 4, I also record the reward and make chart

Below I show the training result of cumulative reward :



When we observe the figure above, we will notice that the cumulative reward of q-learning was greater than SARSA, and I also rendered the environment and noticed that the performance of Q-Learning was more stable than SARSA. I think the reason is that Q-Learning always chose the maximum reward policy to be its next action, therefore, the training result was more amazing than the other one.

Problem 3

I applied SARSA to my tic-tac-toe problem, below I would explain three classes that I used in my task.

State

board : the tic-tac-toe board with shape (3, 3)
p1 : player1 (computer)
p2 : player 2 (computer or human)
isEnd : flag to indicate whether the game is over
boardHash : get unique has of current board as states key
playerSymbol : indicate which player is playing in this term

getHash(self) : return unique has of current board
winner(self) : check whether the game is over
availablePositions(self) : return available positions for player to use
updateState(self, position) : according to given position, update the board
giveReward(self) : according to the winner, feedReward to the winner,
win with 1, lose with 0
play(self) : alternatively play with each other and update the reward based on the result, after all iterations, I combined the learning result of p1 and p2, and stored them
play2(self) : this function can let computer competes with itself or with human
showBoard(self) : show the current board with certain format

Player

name : name of user

states : record all positions taken

lr : learning rate

exp_rate : for epsilon greedy search

decay_gamma : value influenced the update function

states_value : dictionary for {state : value}

getHash(self) : get unique has of current board as states key

chooseAction(self, positions, current_board, symbol) :

choose the position according to the given positions and current board.

addState(self) : add new state to the states

feedReward(self, reward) : back propagate and update states value

According to the above procedure, we can successfully trained the model with every great performance. I trained this game with 20000 iterations. The method I deal with the first play or second play : Since I trained the game with two computer players p1 and p2, so p1 would learn the strategy with first play and p2 on the other hand learn second hand. All I did was combined two learning result so that no meter I played first or second hand won't influence my performance.

Problem 4

I applied the same method in Problem 3, because I could not figure out how to run successfully with my code. However I still upload the my Monte Carlo Search Tree code to the zip file. (107062240_hw1_4_train_MTCS.py)