CS747 Foundations of Intelligent and Learning Agents

Mountain Car

Assignment 3

Param Rathour | 190070049

Autumn Semester 2021-22

Contents

0 Introduction

In this assignment, I have implemented the Mountain Car. Task 1 was to be solved in less than 160 steps by discretising the states and then using tabular sarsa whereas Linear Function Approximation was to be used in Task 2 to solve the problem in less than 130 steps.

1 Code Introduction

I have discretised the space with step size 0.05 for position and 0.01 for velocity. The `weights` is a 3D vector with arguments as state number, tile number (= 0 for Task 1) and number of actions (= 3). The state number is a combined number given to the state. It is same as the index of 2D state matrix having position and velocity when taken as a 1D array row-wise.

2 Task 1

2.1 Initialisation

$\varepsilon = 0.01$, learning rate $\alpha = 0.1$.

2.2 Code

The `get_table_features` was implemented by choosing the closest discretised values for both position and velocity. It returns the state number of discritised state

```
    def get_table_features(self, obs):
        x = int(round((obs[0] - self.lower_bounds[0])/self.step_T1[0]))
        y = int(round((obs[1] - self.lower_bounds[1])/self.step_T1[1]))
        return np.array([x * self.states_num_T1[1] + y])
```

The reason why the returned quantity is array will be explained in Task 2.
For this task the weight vector is same as the estimate $\hat{Q}(s,a)$ initialised as zeros. `choose_action` chooses action epsilon greedily and the greedy choice is the action that gives maximum $\hat{Q}(s,a)$.
The Sarsa Update is the same as given in slides for this task

3 Task 2

3.1 Initialisation

$\varepsilon = 0.001$, learning rate $\alpha = 0.1$.

3.2 Code

The `get_better_features` was implemented by choosing the closest discretised values for both position and velocity. It returns an array of state number of discritised states which are "similar" (as we can already guess the similar states by intuition). The states are defined as similar if the difference in position is less than tile width. All these state are in different tiles.

```
    def get_better_features(self, obs):
        state = []
        for i in range(self.tile_num):
            x = int(round((obs[0] + i * self.tile_width - self.lower_bounds[0])/self.step_T1[0]))
            y = int(round((obs[1] + i * self.tile_width / 10 - self.lower_bounds[1])/self.step_T1[1]))
            s = x * self.states_num_T1[1] + y
            if s < self.states_num:
                state.append(s)

        return np.array(state)
```

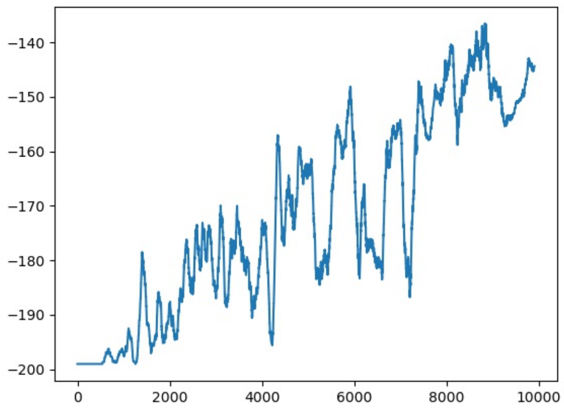
For this task we have to calucalte the estimate $\hat{Q}(s,a)$ from the tiles. `choose_action` chooses action epsilon greedily and the greedy choice is the action that gives maximum $\hat{Q}(s,a)$.

```
    def action_value(self, state, action, weights):
        Q_value = 0
        for i, s in enumerate(state):
            Q_value += weights[s][i][action]
        return Q_value
```

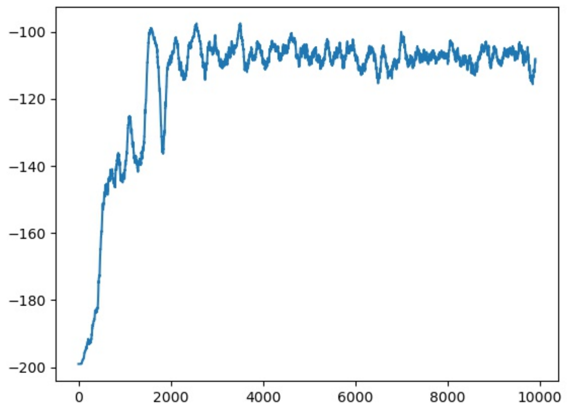
```
    def choose_action(self, state, weights, epsilon):
        if np.random.random() < epsilon:
            return np.random.choice(self.actions)
        else:
            Q_values = []
            for action in self.actions:
                Q_values.append(self.action_value(state, action, weights))
            return np.argmax(Q_values)
```

The Sarsa Update is the same as given in slides but it is applied to all the tiles via neighbouring states.

```
    def sarsa_update(self, state, action, reward, new_state, new_action, learning_rate, weights):
        action_value_old = self.action_value(state, action, weights)
        action_value_new = self.action_value(new_state, new_action, weights)
        target = reward + self.discount * action_value_new
        for i, s in enumerate(state):
            weights[s][i][action] += (learning_rate * (target - action_value_old))
        return weights
```



(a) Task 1



(b) Task 2

Figure 1: Comparison

4 Observations

The plots for task 1 and task 2 are given in Figure 1a and Figure 1b respectively,

- The first thing I observed was learning is much faster with Tiling (9000 vs 2000). Also task 2 has much higher reward (-140 vs -100). This shows the neccesaity of exploiting similar states.
- The algorithms fail to get good results when higher ε values are used. This is sensible as higher ε imply higher exploration but we have only 200 steps to climb the hill and thus the learning updates are less.
- The learning rate has to be tuned carefully as well. A lesser α means more time taken to learn and with higher α the algorithm might not converge to optimal value.

5 Conclusion

It was a pleasure doing this assignment. I learnt a lot about Policy Control and Python overall. Thanks.