Controlling a Toy Car around a Grid [Version 1]

Project Code: TCV1

CS60077-Reinforcement Learning

Pranav Mehrotra, 20CS10085

Overview:

- Objective: Train a Car(agent) to traverse a square grid along a specified track.
- Task:
    - Design an Environment Class that models the given specifications.
    - Design a visualization tool for rendering how the algorithm controls the robot.
    - Use two different methods Policy Evaluation & Iteration and Monte Carlo to train the agent
    - Create a report comparing both methods and give insights

States:

- A state has the following components:
    - X-coordinate
    - Y-coordinate
    - Direction of the car(Up, Right, Down and Left)
- All this information is combined into one state number, so as to facilitate the use of only 1-D array to represent the state space.
- Each cell on the track has a cell number, the starting cell(bottom-left) is the 0th cell and similarly next cell(Upside neighbour of starting cell) on the track is 1st cell and so on. (See the image below for visualisation)

3	4	5	6
2	—	—	7
1	—	—	8
0	11	10	9

- The Car can have four possible directions in each cell: Up, Right, Down and Left

  Directions are numbered(in the code) serially, 0 to 3 in the same order.
- So, **total number of states**:

  $$Number\ of\ Boundary\ Cells\ X\ 4\ =\ (4\ *\ (grid\ size\ -\ 1)\ )\ *\ 4$$
- States are numbered(in the code) serially, starting from 0 to number of states
- From each state, we can get:
  - Cell number: State // 4
  - Direction: State % 4

Actions:

- There are 3 possible actions: **Left, Forward and Right**.
- The actions are also numbered(in the code) serially, 0 to 2 in the same order.

Rewards:

- Reward Structure is same for both Policy Iteration and Monte Carlo methods.
- On every rotation(left or right), the car gets a penalty of **-1**
- On Valid forward movements(moves to next cell), the car gets a reward of **+5**
- On InValid forward movements(hitting boundary walls), the care gets a penalty of **-3**
- On reaching the goal, from the penultimate goal cell(Highest Cell number, Cell to the right hand side of goal cell) the car receives a reward of **+10**

Policy Iteration:

Policy iteration is a method used in reinforcement learning for finding the optimal policy in a Markov Decision Process (MDP). Here's a brief explanation of the algorithm:

1. **Initialization:** Begin with a random policy and an arbitrary value function.

2. **Policy Evaluation:** In this step, we calculate the state-value function 'V' for the existing policy. This is done by iteratively applying the Bellman Expectation Equation until reaching a steady state where the value function does not change significantly.

3. **Policy Improvement:** After policy evaluation, we have the value function according to the current policy. Using this, we can update our policy. For each state, choose the action which maximizes the expected reward based on the current value function.

4. **Policy Stability:** If the policy does not change during the improvement step (i.e., it's already optimal), the algorithm stops and the current policy is returned as the optimal policy. Otherwise, go back to step 2 and repeat the process with the improved policy(upto a maximum number of iterations).

5. **Iterations:** Steps 2, 3, and 4 are repeated until the policy is stable; that is, the policy does not change from one iteration to another.

The key advantage of policy iteration is that it generally takes fewer iterations to converge to the optimal policy than other methods like value iteration. However, each iteration requires more computation because it involves policy evaluation for all states.

Hyper-parameters:

- **Grid Size = 8X8:** Size of the NXN grid
- **Gamma = 0.9:** Reward Discount factor

- **Convergence Factor = 1e-3:** Convergence factor for the Policy Iteration
- **Max Evaluation Iterations = 500:** Maximum number of iterations for the Policy Evaluation
- **Max Policy Improvement Iterations = 100:** Maximum number of iterations for the Policy Improvement

Monte-Carlo:

Monte-Carlo algorithm is a computational method used to find approximate solutions to complex mathematical problems. It is based on the use of random sampling to explore a problem space and estimate the solution. This approach contrasts with DP algorithms, where the whole environment is known prior to the training of the agent.

In a Monte Carlo algorithm, many random inputs are generated, and their outputs are computed. The final result is the average of these outputs. The key advantage of the Monte Carlo algorithm is its simplicity and versatility. However, it also has some limitations. The accuracy of the results depends on the number of random samples. Therefore, a large number of samples may be needed to get a highly accurate solution, which can be computationally expensive.
Some pointers regarding code:

- The **Q-table** is a table that stores the expected value of taking each action in each state. The Q-table is initialized with all zeros, but it is updated over time as the agent learns from experience.
- The **policy** is a mapping from states to actions. The policy tells the agent which action to take in each state. The policy is updated over time to reflect the agent's learning.
- The **epsilon-greedy exploration** strategy means that the agent randomly chooses an action with probability epsilon and otherwise chooses the action that has the highest Q-value for the current state. This helps the agent to explore the environment and avoid getting stuck in local minima.

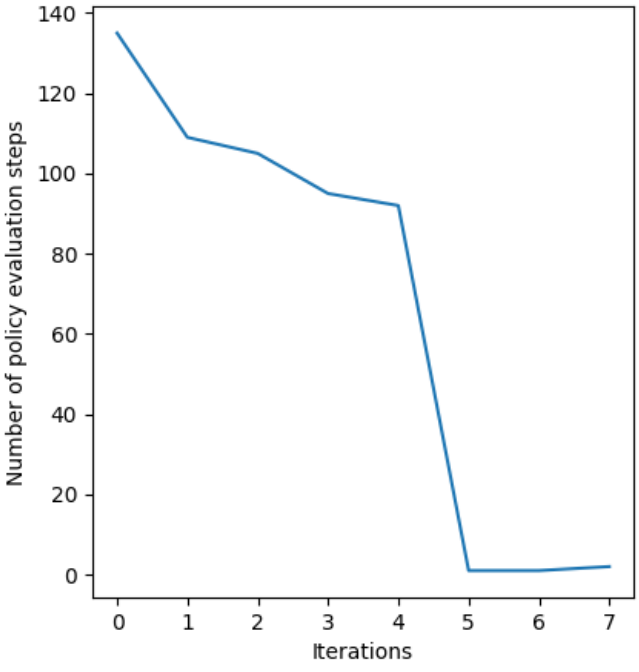
- The **discount factor** is used to weigh the importance of future rewards. A higher discount factor means that the agent will also value future rewards.

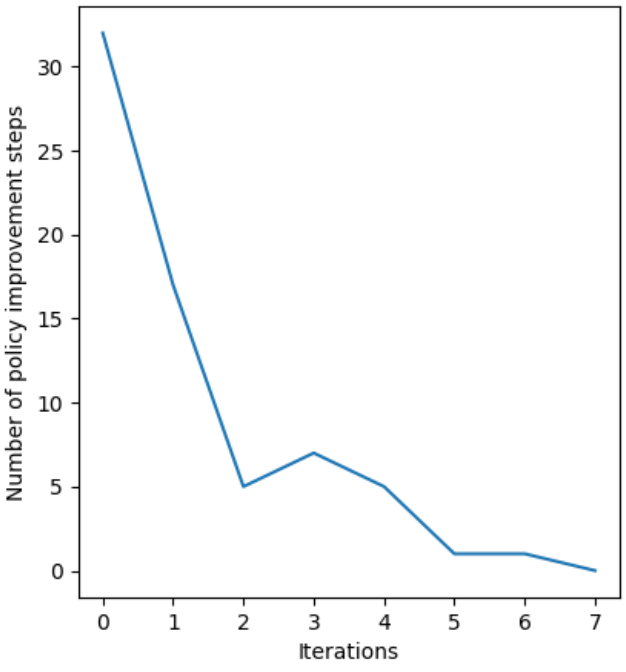
Hyper-parameters:

- **Grid Size = 8X8:** Size of the NXN grid
- **Gamma = 0.9:** Reward Discount factor
- **Epsilon = 0.7:** Exploration factor
- **Xi = 0.99:** Exploration Dampening factor, decreases the Epsilon value after every epoch, so as to reduce exploration factor and help in convergence.

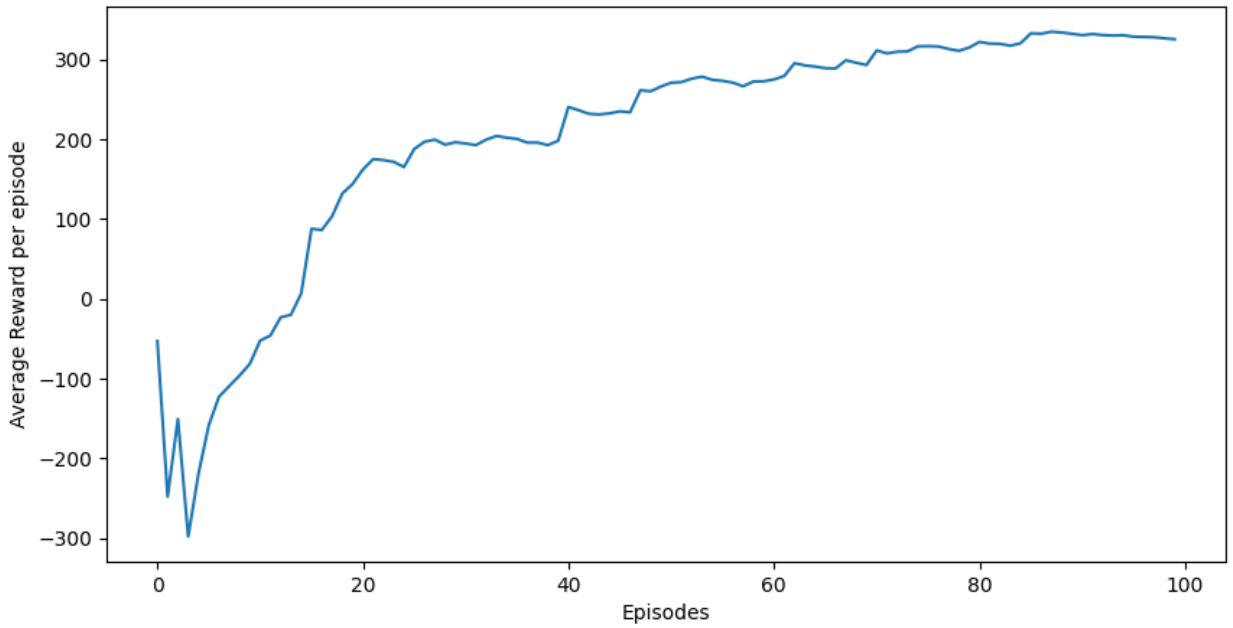
Plots and Analysis:

1. After experimenting with different Hyper-parameter values of Policy iteration, the optimal parameters were finalised, also it was found that almost always of the times the algorithm converges within 6-7 iterations of Policy Improvement, also the number of steps needed in Policy Evaluation before its convergence, also decreases over the iterations.
2. Keeping the Gamma = 1, drastically increases the number of iterations before convergence and also produces the wrong optimal policy.
3. Plots of Number of Policy evaluation steps and number of Policy improvements w.r.t. Iterations:





4. While experimenting with Monte Carlo hyper-parameters, it was found that the convergence of the algorithm is sensitive to the Hyperparameter values.
5. Plot of incremental average Reward per episode w.r.t. Episodes:



6. More plots of **Heat Map** of **Optimal State Values** and **Optimal Policy Grid** are attached in the Zip folder.
7. The plots are generated for both Policy Iteration and Monte Carlo, the code for generating the plots is also integrated in the codebase.

----------END----------