Reinforcement Learning Project

Team-19

RL-19

Problem Statement:

Implementation of **Connect-4** game using Q learning algorithm in Deep Reinforcement Learning.

Group Details:

1.SRIKAR SASHANK MUSHUNURI (2019A7PS0160H)

1.Reinforcement learning in Connect 4 Game

Link-http://www.warse.org/IJATCSE/static/pdf/file/ijatcse181022021.pdf

2.Grid-Wise Control for Multi-Agent Reinforcement Learning in Video Game AI

Link-https://scholar.google.co.in/scholar_url?url=http://proceedings.mlr.press/v97/han19a/han19a.pdf&hl=en&sa=X&ei=mPxfYfGmIIWM6rQPo_eDyAg&scisig=AAGBfm2q2YRH4x2oHqlGrk4vyY6okqlSfA&oi=scholarr

2.SHIVA HARSHITH GUNDLAPALLI (2019A7PS0030H)

1.Connect-based Subgoal Discovery for Options in Hierarchical Reinforcement Learning

Link-https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4344762

2.Q-Learning Algorithms: A Comprehensive Classification and Applications

Link-https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8836506

3.DURGAVARAPU SRI KRISHNA KARTHIK (2019A7PS0189H)

1.Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing

Link-https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8666545

2.Tuning Computer Gaming Agents using Q-Learning

Link- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6078182

4.GAGAN REDDY KONANI (2019A7PS0169H)

1.Reinforcement_Learning_for_Connect Four

Link- https://web.stanford.edu/class/aa228/reports/2019/final106

2.Using a Reinforcement Q-Learning-Based Deep Neural Network for Playing Video Games

Link-https://scholar.google.co.in/scholar_url?url=https://www.mdpi.com/2079-9292/8/10/1128/pdf&hl=en&sa=X&ei=A_xfYZOyJILQyQTB27FY&scisig=AAGBfm0nAtwUPLrVVrZIEg9uvfFI1ErC1Q&oi=scholarr

5.DONI AKHIL LOHITH (2019A7PS0026H)

1.Deep Reinforcement Learning for General Game Playing

Link- https://ojs.aaai.org/index.php/AAAI/article/view/5533/5389

2.Grid Path Planning with Deep Reinforcement Learning: Preliminary Results

Link-https://scholar.google.co.in/scholar_url?url=https://www.sciencedirect.com/science/article/pii/S1877050918300553/pdf%3Fmd5%3Dc81a3c0400e7d86dbeabca22f193b672%26pid%3D1-s2.0-S1877050918300553-main.pdf%26_valck%3D1&hl=en&sa=X&ei=V_tfYdzBA6XGywS7_I6ABQ&scisig=AAGBfm1uMDO4q-_fLKSb7s6YthHxig9ZJg&oi=scholarr

All of the research papers are embedded in the google drive link given below with each student having a separate folder.

https://drive.google.com/drive/folders/1_SzM86bpPZkkkT215CFiHgbrfJIWaSnt?usp=sharing

REPORT :

Implementation of the paper:

Paper Implemented:http://www.warse.org/IJATCSE/static/pdf/file/ijatcse181022021.pdf
The paper presents a reinforcement learning approach to the game Connect Four. We survey reinforcement learning technique Q-Learning to train an agent to play Connect Four using an optimal strategy for the game. We studied how varying exploration rate and rewards models affects performance by this algorithm. We decided to implement the game using the Q learning algorithm in python.

Action space:
Each player has at most seven possible actions that they can take at any given state. Therefore an action is defined as dropping a piece into one of the seven columns on the board. Number of actions possible:

$$Actions\ Possible = 7 - C$$

$$C = Number\ of\ full\ columns\ at\ current\ state$$

State Space:
A state in Connect Four is defined as the board with played pieces that a player sees. If the player starts the game, then the board will always have an even number of pieces. Alternatively if they are the second player, the board will always have an odd number of pieces.

Because the board size is 6 x7 , we get an upper bound of 3^{42} . The best lower bound on the number of possible positions has been calculated by a computer program to be around 1.6 x 10^{13}.

Algorithm used:
We have used the Q-learning algorithm, i.e, for each observation (s, a', r, s') at time t, we perform the following incremental update rule
$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_a Q(s',a') - Q(s,a))$

$$Q(s,a) = Q(s,a) + \alpha(\max(Q(s',a')) + gR - Q(s,a))$$

with a learning α and discount factor γ.

Implementation:
We first created a class "Player()" which, according to its name, represents a player in the game. We then created the classes, "HumanPlayer()", "ComputerPlayer()", "RandomPlayer()"

and then finally “QLearningPlayer()” all of whom take the argument “Player()” which is the class we defined in the beginning.

- HumanPlayer(Player): A class that represents a human player in the game.
- ComputerPlayer(Player): A class that represents an AI player in the game.
- RandomPlayer(Player): A class that represents a computer that selects random moves based on the moves available in the game.
- QLearningPlayer(Player):A class that represents an AI player which uses the Q learning algorithm in the game.

The implementation of the Q learning algorithm is contained in the class “QLearningPlayer()”. We defined certain functions inside the class which are,

- __init__(): Initialize a Q learner with parameters epsilon, alpha and Gamma.

```
def __init__(self, coin_type, epsilon=0.2, alpha=0.3, gamma=0.9):

    Player.__init__(self, coin_type)
    self.q = {}
    self.epsilon = epsilon  # e-greedy chance of random exploration
    self.alpha = alpha  # learning rate
    self.gamma = gamma  # discount factor for future rewards
```

- getQ(): Return a probability for a given state and action where greater the probability, better is the move.

```
def getQ(self, state, action):

    # encourage exploration; "optimistic" 1.0 initial values
    if self.q.get((state, action)) is None:
        self.q[(state, action)] = 1.0
    return self.q.get((state, action))
```

- choose_action(): Return an action based on the best move recommendation by the current Q - Table with an epsilon chance of trying out a new move.

```
def choose_action(self, state, actions):

    current_state = state

    if random.random() < self.epsilon:  # explore!
        chosen_action = random.choice(actions)
        return chosen_action

    qs = [self.getQ(current_state, a) for a in actions]
    maxQ = max(qs)

    if qs.count(maxQ) > 1:
        # more than 1 best option; choose among them randomly
        best_options = [i for i in range(len(actions)) if qs[i] ==
        i = random.choice(best_options)
    else:
        i = qs.index(maxQ)

    return actions[i]
```

- learn(): Determine the reward based on its current chosen action and update the Q table using the reward received and the maximum future reward based on the resulting state due to chosen action.

```
def learn(self, board, actions, chosen_action, game_over, game_logic):

    reward = 0
    if (game_over):
        win_value = game_logic.get_winner()
        if win_value == 0:
            reward = 0.5
        elif win_value == self.coin_type:
            reward = 1
        else:
            reward = -2
    prev_state = board.get_prev_state()
    prev = self.getQ(prev_state, chosen_action)
    result_state = board.get_state()
    maxqnew = max([self.getQ(result_state, a) for a in actions])
    self.q[(prev_state, chosen_action)] = prev + self.alpha * ((reward + self.gamma * maxqnew) - prev)
```

Two Player Mode:

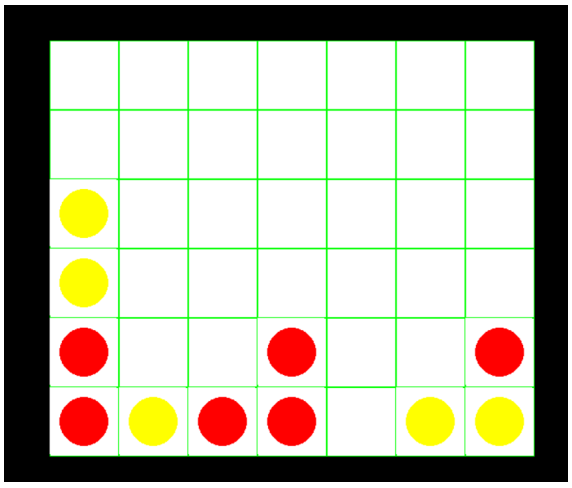
- This is a user interface where a connect-4 game between two human players is implemented.
- “HumanPlayer()” is used on both the sides

Vs Computer Mode:

- This is a user interface for the user to play the connect-4 game with the Computer
- Here both the classes “HumanPlayer()”, “ComputerPlayer()” play against each other.
- Also we implemented in such a way that, if we use train computers before using Vs Computer Mode, then the resultant “ComputerPlayer()” playing against the “HumanPlayer()” would be much more intelligent in making its moves.
- This mode acts like Proof that the computer uses its learned knowledge to make decisions regarding next move as part of Human Vs Computer connect-4 game.

Training Computer:

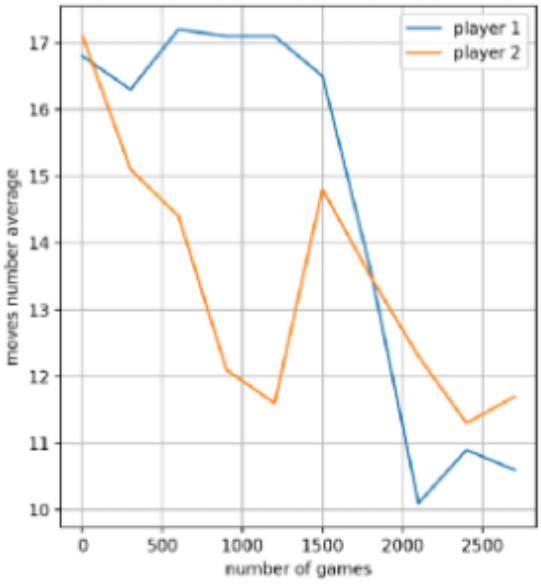
- This is where improvement in Learning is implemented.
- Here RandomPlayer() and QLearningPlayer() were used.
- RandomPlayer() as the name suggests picks random moves in the connect-4 game on one side.
- QLearningPlayer() responds to those moves while learning from previous moves and based on the rewards previously.
- This Learned knowledge with QLearningPlayer() can be witnessed when we play again in Vs Computer Mode.



Results of the implementation:

We have iterated many times to generate a graph with the number of iterations(/number of games the agent has played) on the X-axis and the win percentage on the Y-axis.

- The graph shows the winning rate of the Q agent against a random player at the beginning of the learning process. We can see that as the winning rate increases, the number of moves decreases which shows that the agent learns to win and win fast.



Proposed change/innovation

CHANGE-1 (policy improvement)

- The idea to improve the results that we have got in step 4 is to improve the policy by '**Policy Improvement theorem**'.
- We particularly have followed the **value iteration** policy improvement method instead of the policy iteration method .
- As discussed in the class , the improvement for a policy will be done first by initializing the policy then evaluating the state value from the bellman equation and then choosing a better policy which gives a better state value using bellman equality and so on .
- Then we reach a better policy in the long run. This is the policy iteration method.
- But we have followed the value iteration method which is a combination of policy improvement and truncated policy iteration. It chooses the best possible state at every iteration rather than after the evaluation of a state value.

- ❖ Here we first choose a random action (choose_action() method) based on checking the emptiness of all columns and iterate them through a for loop and choose a random one by **random.choice()** function.
- ❖ But we made an improvement in the policy to check out for the state values at every iteration and get the best among those(and the best is found from the Q-learning algorithm) and if we receive multiple best actions from that , then choose a random one among the best. If only one best action is received from the getQ() function , then we choose that as the next state or the action that results in that state.

CHANGE-2 (Monte Carlo Tree Search based algorithm)

More efficient and a better algorithm :

- Although the Monte Carlo Tree Search Based takes more time than Q-learning, learning is much more efficient or quicker than Q agents.
- **Q agent player learns to generalise by learning similarities between different states and the MCTS agent learns to take the most promising path**.
- The idea behind this algorithm is to create a game tree, but instead of exploring all the possible games, only the most promising routes are chosen.
- Every node in the tree stores 2 values: the number of times the node was visited, and the number of wins the player won when visited this node. The algorithm contains 4 steps which repeats multiple episodes.

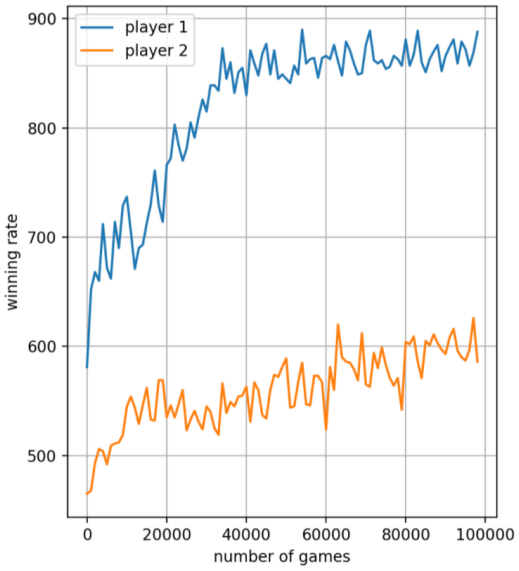


Fig1

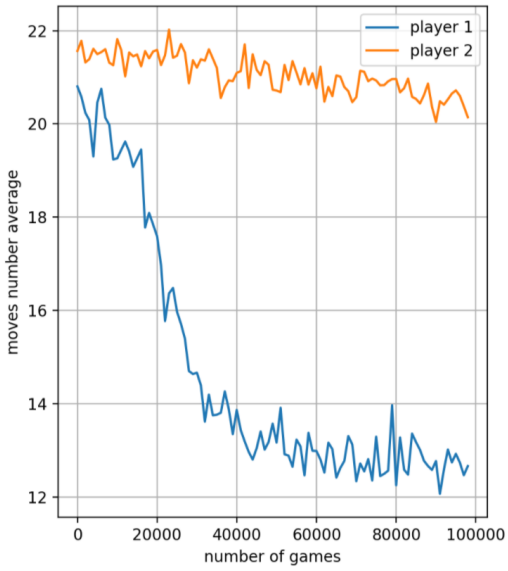
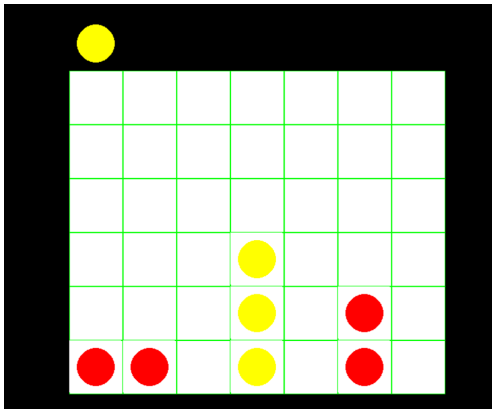


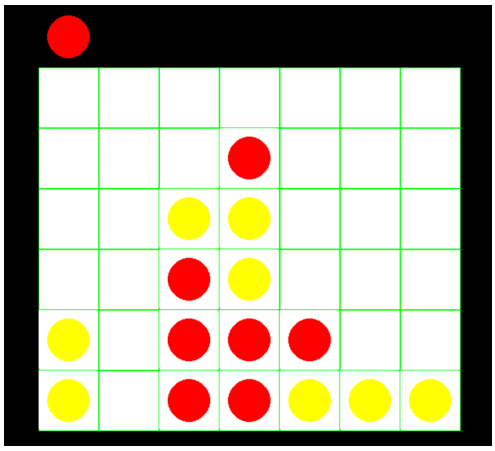
Fig2

- Fig1 is a graph plotted between winning rate of player1 vs winning rate of player2 which are montecarlo player and random player respectively.
- As depicted in the graph we find that using monte carlo tree search based algo , player1's winning rate has increased significantly.
- Fig2 is a graph plotted between the average number of moves taken to win connect4 game for player1 and player2 which are montecarlo player and random player respectively.
- As depicted in the Fig2 we find that monte carlo tree search based algo, makes the player1 win faster in less number of moves.

Experiments conducted:

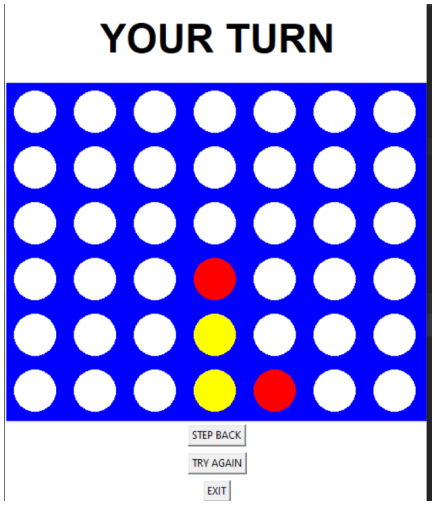


For random policy , even though we have piled up three same colored coins , the agent hasn't dropped a coin in the middle column but rather has dropped in the first column in this particular experiment.
(Here red is computer)



For the Q-learning agent , we have done the same steps that is piling up three same colored coins in one of the columns but this time as it's trained beforehand and also a q-learner , it immediately recognized the stack and dropped a coin in the same column to prevent the loss.

(Here yellow is computer)



Meanwhile while we have done the same experiment for the monte carlo method , it resulted that immediately after piling up two coins it dropped a coin in my column but it need not be the same all time . It depends on the number of moves till then we had and other column fillings.

Findings and other Accomplishments:

- According to our findings, the agent using the Monte Carlo Tree Search Algorithm consistently had a marginally higher win rate as compared to the agent using the Q learning algorithm.
- The agent using MCTS takes more episodes to learn than the Q learning agent, but the learning is faster.
- Although Connect 4 has more than 10^4 possible states, which means that probably every game ever played is unique, the agents learn to generalized and play well. The **Q agent player learns to generalize by learning similarities between different states and the MCTS agent learns to take the most promising path**.

Table of Work Division:

Name	ID	WorkDone
SRIKAR SASHANK MUSHUNURI	2019A7PS0160H	Report writing and implementation of Q learning policy using policy improvement theorem(change-1).
SHIVA HARSHITH GUNDLAPALLI	2019A7PS0030H	Report Writing/ Monte Carlo Tree Search.
DURGAVARAPU SRI KRISHNA KARTHIK	2019A7PS0189H	Report Writing and mainly involved in coding random policy implementation and helped UI in Monte carlo.
GAGAN REDDY KONANI	2019A7PS0169H	Report Writing/ Monte Carlo Tree Search.
DONI AKHIL LOHITH	2019A7PS0026H	Report writing and coded in the implementation of the Human Player class and mainly in the UI development.

The GitHub link for the project repository :

https://github.com/BENTENNYSON-5/RL_project_Connect4