DRL_HW1

107081028 常安彥

1. Random Policy Evaluation

a. Implementation

First, I initialized the value function as 4 * 4 zero matrix, and concluded action space as [[0, 1], [-1, 0], [0, -1], [1, 0]], which represents moving right, up, left, down respectively. Then, the converging factor, theta, is set as 0.0001 and gamma is set as required for different problems.

```
V = np.zeros((4, 4))
action_space = [[0, 1], [-1, 0], [0, -1], [1, 0]]                    # {right, up, left, down}
```

In the iteration, the function calculates the accumulation of rewards of every action through all states, implemented with three for-loops. For each state, if it is in terminate state, reward is 0; if it is out of map after action, don't do action with reward -1; otherwise reward is -1 with action done. Afterwards, reward is accumulated as the value function, "p(action) * (reward + gamma * V[state']."

```
while(delta >= theta):
    iter += 1
    delta = 0
    V_copy = copy.deepcopy(V)
    for i in range(4):
        for j in range(4):
            rewards = 0
            for action in action_space:
                reward = 0
                action_i = 0
                action_j = 0
                # terminate
                if (i, j) == (0, 0) or (i, j) == (3, 3):
                    reward = 0
                # out of map -> stay initial state
                elif (i + action[0] == 4) or (i + action[0] == -1) or (j + action[1] == 4) or (j + action[1] == -1):
                    reward = -1
                # do action
                else:
                    reward = -1
                    action_i = action[0]
                    action_j = action[1]
                rewards += (1/len(action_space)) * (reward + gamma * V_copy[i + action_i, j + action_j])
            delta = max(delta, abs(V_copy[i, j] - rewards))
            V[i, j] = rewards
```

Finally, calculate the difference between the value of state and state' and check if delta is smaller than theta, deciding whether to terminate.

b. Results

Gamma = 0.9

```
60
[[ 0.    -5.28 -7.13 -7.65]
 [-5.28 -6.61 -7.18 -7.13]
 [-7.13 -7.18 -6.61 -5.28]
 [-7.65 -7.13 -5.28  0.  ]]
```

Gamma = 0.1

```
5
[[ 0.   -1.08 -1.11 -1.11]
 [-1.08 -1.11 -1.11 -1.11]
 [-1.11 -1.11 -1.11 -1.08]
 [-1.11 -1.11 -1.08  0.  ]]
```

(with theta = 0.0001) When discount factor is 0.9, the function iterates 60 times to converge; when it is 0.1, the iteration ran only 5 times to converge.

c. Discussion

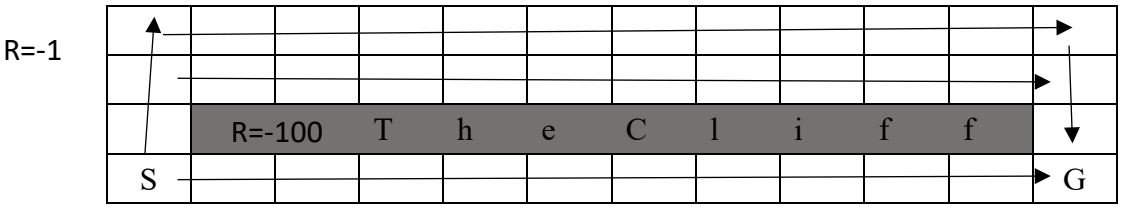
The iteration stops until delta is smaller than theta, which means that the function reaches convergence. In the policy evaluation algorithm, it calculated delta to reflect the difference the value of state changes respect to the previous value. Delta decays through iterations and reaches near 0 eventually.

Discounting factor, gamma, is considering of immediate rewards are more important than future rewards. We can see that through the iterations, states far from terminate states are affected later than those nearby. This is how adjusting gamma may influence. Furthermore, the lower gamma is, the faster the iteration reaches convergence. This is obvious because the discounting makes the difference decreases, and the more it decreases, the earlier it converges.

2. Compare Q-Learning with SARSA

a. Implementation

- Environment (4, 12)



In my design, the cliff is lifted up to the row above the row it was before in the example.

- Alpha = 0.5 (learning rate)
- Gamma = 1.0
- Epsilon greedy = 0.05
- reward = -1 for transition, -100 for falling off cliff, 0 for terminate
- Episode = 500

• Q-Learning

```
def q_learning():
    reward_plot = list()
    qlr_table = np.zeros((4, 12, 4))
    for i in range(500):
        terminate = False
        current_environment = np.zeros((4, 12))
        current_environment[3][0] = 1
        current_row = 3
        current_col = 0
        reward_accumulatived = 0
        step = 1
        while(terminate == False):
            step += 1
            rng = np.random.random()
            action = 3
            if(rng < 0.05):
              action = np.random.choice(4)
            else:
               action = np.argmax(qlr_table[current_row, current_col, :])
            # action = np.argmax(qlr_table[current_row, current_col, :])
            next_row = current_row
            next_col = current_col
            if(action == 0 and current_row > 0):
                next_row = current_row - 1
            elif(action == 1 and current_col > 0):
                next_col = current_col - 1
            elif(action == 2 and current_col < 11):
                next_col = current_col + 1
            elif(action == 3 and current_row < 3):
                next_row = current_row + 1
            current_environment[next_row][next_col] = step
            max_value = np.max(qlr_table[next_row, next_col, :])
            reward = -1
            if(next_row == 3 and next_col == 11):
                terminate = True
                reward = 0
            elif(next_row == 2 and next_col > 0 and next_col < 11):
                terminate = True
                reward = -100
            reward_accumulatived += reward
            next_qlr_value = qlr_table[current_row, current_col, action] + \
                alpha * (reward + gamma * max_value - qlr_table[current_row, current_col, action])
            qlr_table[current_row, current_col, action] = next_qlr_value
            current_row = next_row
            current_col = next_col
        reward_plot.append(reward_accumulatived)
```

For each step of episode, choose action from state using policy. Then, take action to observe reward and state'. Finally, update the q board with the function and update new state with state'.

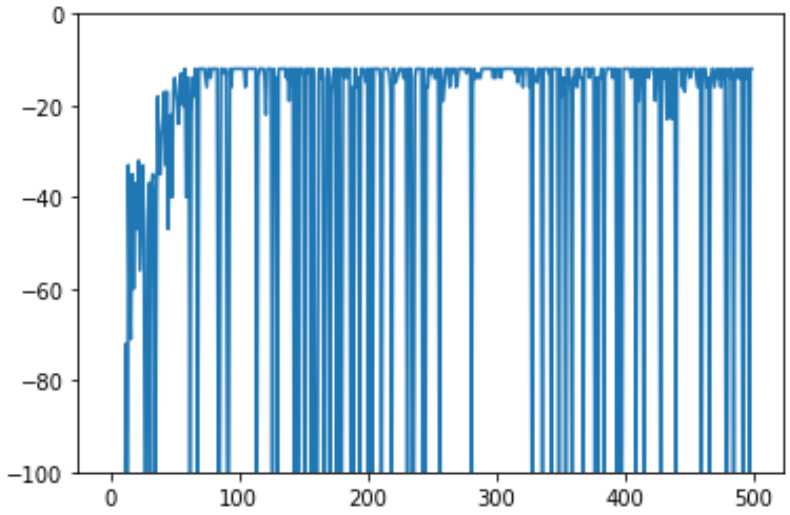
• SARSA

```
def sarsa():
    reward_plot = list()
    sarsa_table = np.zeros((4, 12, 4))
    for i in range(500):
        terminate = False
        current_environment = np.zeros((4, 12))
        current_row = 3
        current_col = 0
        reward_accumulatived = 0
        action = np.argmax(sarsa_table[current_row, current_col, :])
        step = 1
        while(terminate == False):
            step += 1
            next_row = current_row
            next_col = current_col
            if(action == 0 and current_row > 0):
                next_row = current_row - 1
            if(action == 1 and current_col > 0):
                next_col = current_col - 1
            if(action == 2 and current_col < 11):
                next_col = current_col + 1
            if(action == 3 and current_row < 3):
                next_row = current_row + 1
            current_environment[current_row][current_col] = step
            reward = -1
            if(next_row == 3 and next_col == 11):
                terminate = True
                reward = 0
            elif(next_row == 2 and next_col > 0 and next_col < 11):
                terminate = True
                reward = -100
            reward_accumulatived += reward
            rng = np.random.random()
            next_action = 3
            if(rng < 0.05):
               next_action = np.random.choice(4)
            else:
               next_action = np.argmax(sarsa_table[next_row, next_col, :])
            # next_action = np.argmax(sarsa_table[next_row, next_col, :])
            value = sarsa_table[next_row, next_col, next_action]
            next_sarsa_value = sarsa_table[current_row, current_col, action] + \
                alpha * (reward + gamma * value - sarsa_table[current_row, current_col, action])
            sarsa_table[current_row, current_col, action] = next_sarsa_value
            current_row = next_row
            current_col = next_col
            action = next_action
        reward_plot.append(reward_accumulatived)
```

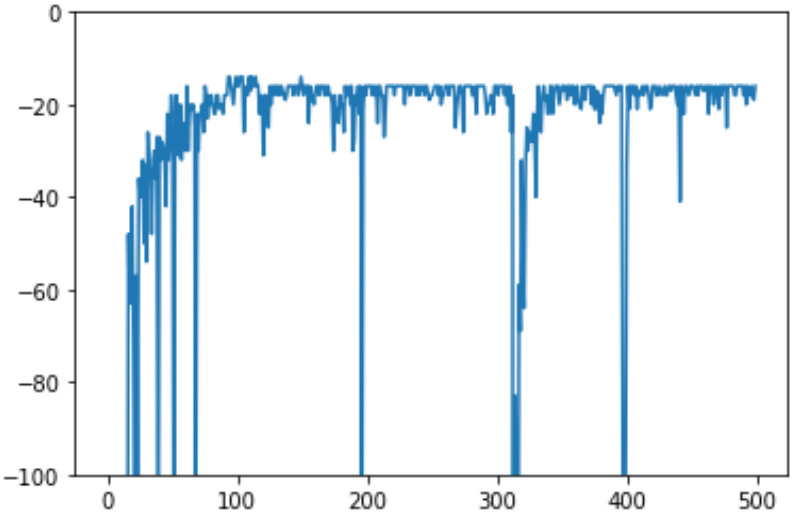
For each step of episode, take action and observe reward and state'. Then, choose action' from state' using policy. Finally, update q board with function and also update state and action.

b. Result

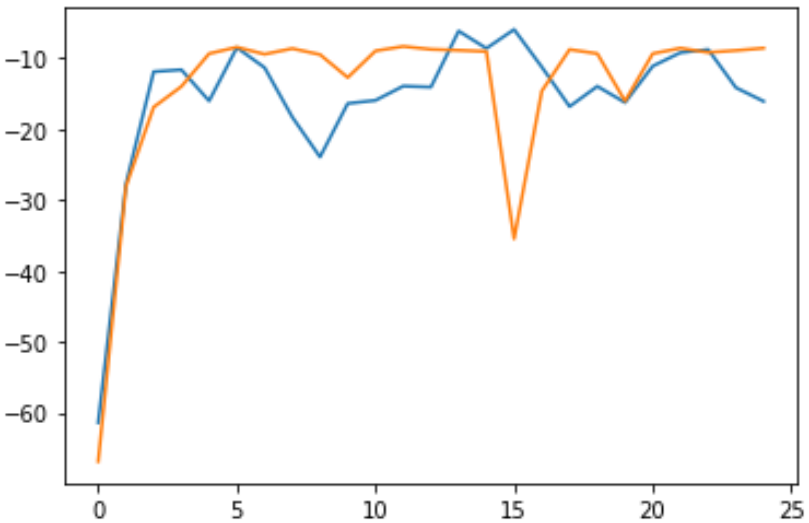
Q-learning:



SARSA:



Both: plot with normalization {norm = (x – mean) / std}



Blue: Q-learning Yellow: SARSA

c. Discussion

The obvious difference between the two algorithm is:

Q-learning goes the path near the cliff in order not to do too many transitions, which cause -1 reward for each.

SARSA avoids paths nearby the cliff in order not to fall off the cliff, which cause -100 reward when occurs.

From the plots above, these results were shown. Q-learning occasionally falls off the cliff due to the epsilon greedy function, and receives -100 as termination; SARSA seldom receive -100 as reward, but does more transitions than Q-learning does, receiving many -1.

Finally, the figure below shows the path Q-learning and SARSA often goes:

Q-Learning

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  2.  3.  4.  5.  7.  8.  9. 10. 11. 12. 13.]]
```

SARSA

```
[[ 6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.]
 [ 5.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 18.]
 [ 4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 19.]
 [ 2.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

3. Tic-tac-toe

a. Implementation

- States: 3 ^ 9 (19683) representing each board, [-1,0,1] arranged in 9 frames
- Action space: 0 to 8 representing each frame in board
- Reward: 1 for a win of cross, 0.5 for a draw, 0 for a win of circle
- Value table: numpy.ones((19683)) * 0.5
- Epsilon: 0.4 for first 200000 episodes, 0.8 for rest 100000 episodes
- Alpha: 0.01 (learning rate)
- Episodes: 300000

In every episode, randomly choose which player to go first and initialize the useful variables

```
for _ in range(300000):
    state = [0 for i in range(9)]
    terminate = False
    action_list = [i for i in range(9)]
    state_index = states.index(state)
    O_X = random.choice([-1, 1])
```

While the game hasn't terminated, choose action of current state from the available action list based on the value table, and later on remove the chosen action from available action list.

```
while terminate == False:
    next_state_index, value, action = Model1.choose_action(state_index, action_list, O_X)
    action_list.remove(action)
```

Below shows choosing action with epsilon greedy, randomly choose action or choose action based on the best of value table by comparing the value of each possible next states.

```
def next_state_remain(self, state, actions, O_X):
    state_actions = []
    for action in actions:
        current = states[state].copy()
        current[action] = O_X
        state_index = self.states.index(current)
        state_actions.append((state_index, action))
    return state_actions

def choose_action(self, state, actions, O_X):
    if(np.random.uniform() >= self.epsilon):
        action = np.random.choice(actions)
        current = states[state].copy()
        current[action] = O_X
        state_index = self.states.index(current)
        return state_index, self.table[state_index], action
    else:
        state_actions = self.next_state_remain(state, actions, O_X)
        if O_X == 1:
            value = -sys.float_info.max
            chosen_action = -1
            for (indices, action) in state_actions:
                if self.table[indices] > value:
                    value = self.table[indices]
                    state_index = indices
                    chosen_action = action
            return state_index, value, chosen_action

        else:
            value = sys.float_info.max
            chosen_action = -1
            for (indices, action) in state_actions:
                if self.table[indices] < value:
                    value = self.table[indices]
                    state_index = indices
                    chosen_action = action
            return state_index, value, chosen_action
```

Another important implementation is min-max algorithm, here circle player chooses based on minimum and cross player chooses based on maximum. Also, the win of circle player receives 0 while the win of cross player receives 1 in order to match the min-max algorithm.

Next, operate the action and observe the reward and check if terminate or not (1). The termination detection is implemented by summing the possible winning situation and sees if the summation matches 3 or -3 (2). If termination occurs, update the terminate state of value table immediately (3) to affect the current state in later on update.

```
terminate, reward = make_action(next_state_index, O_X)
if terminate:
    Model1.terminate(next_state_index, reward)
    if reward == 1:
        win_cross += 1
    elif reward == 0.5:
        tie += 1
    else:
        win_circle += 1
```

(1)

```
def rewarding(state):
    ending_board = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), \
        (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)]
    for(i, j, k) in ending_board:
        if(sum([state[i], state[j], state[k]]) == -3):
            return circle_reward
        if(sum([state[i], state[j], state[k]]) == 3):
            return cross_reward
    if 0 not in state:
        return 0.5
    return -1

def make_action(state_index, O_X):
    reward = rewarding(states[state_index])
    terminate = False
    if reward == 1 or reward == 0 or reward == 0.5:
        terminate = True
    return terminate, reward
```

(2) ↑

```
def terminate(self, state, reward):
    self.table[state] = reward
```

(3) ↑

Finally, update the value of current state in the value table with the value function. Update new state and exchange turns.

```
Model1.learn(state_index, next_state_index, action)
state_index = next_state_index
O_X *= -1
```

```
def learn(self, state_index, next_state_index, action):
    value = self.table[state_index]
    next = self.table[next_state_index]
    self.table[state_index] += 0.01 * (next - value)
```

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

(value function)

b. Result

Some actions chosen from designed states:

```
X
? X O
O X ?
? ? ?
1 2

X
O X O
? X X
? O ?
0 1

O
O X O
? X X
? O ?
0 1
```

```
X
? O ?
X X ?
O X O
2 1

X
? ? ?
? ? ?
? ? ?
1 1

O
? ? ?
? ? ?
? ? ?
1 1
```

```
O
O X ?
? O X
? ? ?
2 2

X
O X ?
? O X
? ? ?
2 2

X
O X O
X X ?
O O ?
2 1
```

(the actions are all optimal)

After 300000 episodes, let the model compete with itself without epsilon greedy, so that it always operates actions due to the value table. The figure below shows that every game ends with a draw, and only two optimal models ends with a draw for every game.

```
win of cross: 110173
win of circle: 109760
tie: 79068
[0, 0, 0, 1, 1, 1, 0, -1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
Draw
[1, -1, -1, -1, -1, 1, 1, 1, -1]
```

c. Discussion

- Implementation of environment

  I firstly listed all arrangements of the boards and tag their indices. Every time looking up the table, I check the index of the current state through this list.

- Algorithm

  I tried Q-learning at first, but I couldn't properly pass the effect of termination back to the initial state. Therefore, I tried Q-learning with back propagation, and meet another problem: which state to set as the next state respect to the back propagation and self-competing model.

  I implemented min-max algorithm to deal with the self-competing problem, and later on think of the necessary of discount factor. Whether winning early or lately in tic-tac-toe doesn't matter, so I decided to simply implement value policy and store only the state values in the value table.

4. Connect X

a. Implementation

- State

  - get available action: return list of available columns to make action
  - check terminate: check if current state board terminates the game
    If it terminates, update the winner of this board
  - next state: make action of the state boars respect to the input action

```
class State:
    def __init__(self, board, player, winner = 0):
        self.board = board.copy()
        self.player = player
        self.winner = winner

    def get_available_actions(self):
        return [act for act in range(configuration.columns) if self.board[act] == 0]

    def check_terminate(self, action, has_played = True):
        columns = configuration.columns
        rows = configuration.rows
        inarow = configuration.inarow - 1
        sum = []
        for r in range(rows):
          sum.append(self.board[action + r * columns])
        if not opponent(self.player) in sum:
          has_played = False
        row = (
            min([r for r in range(rows) if self.board[action + (r * columns)] == opponent(self.player)])
            if has_played
            else max([r for r in range(rows) if self.board[action + (r * columns)] == EMPTY])
        )

        def count(offset_row, offset_column):
            for i in range(1, inarow + 1):
                r = row + offset_row * i
                c = action + offset_column * i
                if (
                    r < 0
                    or r >= rows
                    or c < 0
                    or c >= columns
                    or self.board[c + (r * columns)] != opponent(self.player)
                ):
                    return i - 1
            return inarow

        if (
            count(1, 0) >= inarow  # vertical.
            or (count(0, 1) + count(0, -1)) >= inarow  # horizontal.
            or (count(-1, -1) + count(1, 1)) >= inarow  # top left diagonal.
            or (count(-1, 1) + count(1, -1)) >= inarow  # top right diagonal.
        ):
            self.winner = opponent(self.player)
```

```
def next_state(self, action):
    for i in range(configuration.rows - 1, -1, -1):
        if(self.board[i * configuration.columns + action] == 0):
            self.board[i * configuration.columns + action] = self.player
            self.player = opponent(self.player)
            break
```

• Node

-UCB function: calculate the upper confidence bound value of the node

$$\frac{W_i}{N_i} + \sqrt{\frac{C \times lnN}{N_i}}$$

- random action: return a random choice among the available actions
- select best: select the child node with the highest UCB value and return the key of child node (the action to reach this child state)
- expand: pop one action from available actions of current node, and do the action. Also, initialize a new node with the new state and new mark of player as current node's child node. Return the child node
- not root: if current node not root node return true by checking whether current node has a parent node or not
- update: when some nodes reach terminal state, start to update the reward due to the winner upwards to the root
- roll out: simulate the whole process of the game afterwards by randomly choosing the rest actions, and return the winner after terminates.

```python
class Node:
    def __init__(self, state: State, parent = None):
        self.state = deepcopy(state)
        self.available_actions = state.get_available_actions()
        self.parent = parent  # parent is input
        self.children = {}    # value is child node, key is action
        self.Q = 0
        self.visited = 0

    def UCB_func(self, c_param = 1.4):
        if self.visited == 0:
            return math.inf
        return -self.Q / self.visited + c_param * np.sqrt(2 * np.log(self.parent.visited) / self.visited)

    def random_action(self, available_actions):
        return available_actions[np.random.choice(range(len(available_actions)))]

    def select_best(self, c_param = 1.4):
        """use list of UCBs to select best"""
        Max = -1
        action  = None
        for child in self.children.keys():
            temp = self.children[child].UCB_func(c_param)
            if temp > Max:
                Max = temp
                action = child
        return action, self.children[action]
```

```
def expand(self):
    action = self.available_actions.pop()
    current = self.state.player
    next_board = self.state.board.copy()
    for i in range(configuration.rows - 1, -1, -1):
        if(next_board[i * configuration.columns + action] == 0):
            next_board[i * configuration.columns + action] = current
            break
    next_player = opponent(current)
    state = State(next_board, next_player)
    state.check_terminate(action)
    child = Node(state, self)
    self.children[action] = child
    return child

def not_root(self):
    return False if self.parent == None else True

def update(self, champian):
    self.visited += 1
    oppo = opponent(self.state.player)
    if champian == self.state.player:
        self.Q += 1
    elif champian == oppo:
        self.Q -= 1

    if self.not_root():
        self.parent.update(champian)

def roll_out(self):
    current_state = deepcopy(self.state)
    while True:
        if current_state.winner:
            return current_state.winner
        available_actions = current_state.get_available_actions()
        action = self.random_action(available_actions)
        current_state.next_state(action)
        current_state.check_terminate(action)

def fully_expanded(self):
    return len(self.available_actions) == 0
```

• simulation: find the best leaf node to roll out for, and simulate until terminate in order to update values

• simulation policy: if current state is terminal state return this state, otherwise expand the node if still action available, the last choice is selecting the best child for current node

While still have time to simulate, do simulation. Finally, select the best among available actions of current node.

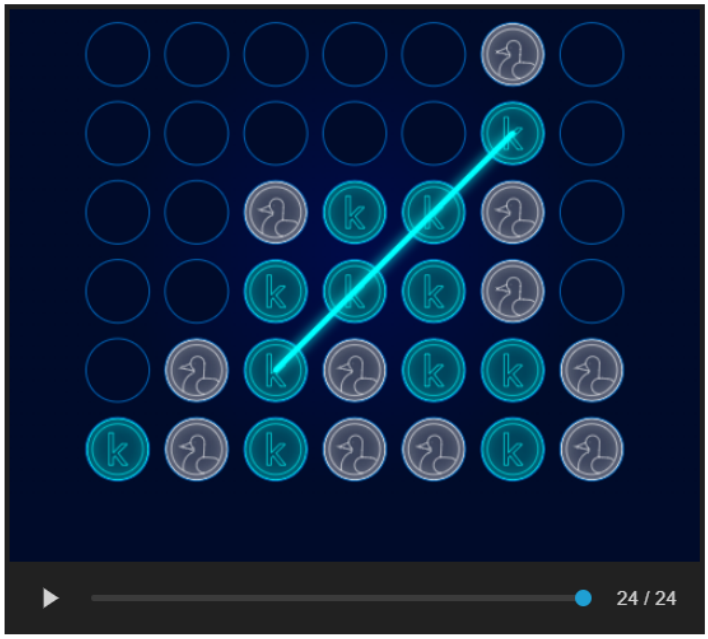
```python
def simulation():
    leaf = simulation_policy()
    winner = leaf.roll_out()
    leaf.update(winner)

def simulation_policy():
    current = current_s
    while True:
        if current.state.winner:
            break
        if current.fully_expanded():
            _, current = current.select_best()
        else:
            return current.expand()
    leaf = current
    return leaf

CurrentState = State(observation.board, observation.mark, 0)
current_s = Node(CurrentState)
while time.time() - init_time <= T_max:
    simulation()
action, next_ = current_s.select_best()
return action
```

Results (Chang an yan) ← Kaggle name

b. Results

My agent (MCTS) vs “negamax” from Kaggle:

```python
env.reset()
# Play as the first agent against default "random" agent.
env.run([MCTS, "negamax"])
env.render(mode="ipython", width=500, height=450)
```



My agent (MCTS) vs random / negamax:

```
def mean_reward(rewards):
        return sum(r[0] for r in rewards) / len(rewards)

# Run multiple episodes to estimate its performance.
print("My Agent vs Random Agent:", mean_reward(evaluate("connectx", [MCTS, "random"], num_episodes=2)))
print("Random Agent vs My Agent:", mean_reward(evaluate("connectx", ["random", MCTS], num_episodes=2)))
print("My Agent vs Negamax Agent:", mean_reward(evaluate("connectx", [MCTS, "negamax"], num_episodes=5)))
print("Negamax Agent vs My Agent:", mean_reward(evaluate("connectx", ["negamax", MCTS], num_episodes=5)))
```

```
My Agent vs Random Agent: 1.0
Random Agent vs My Agent: -1.0
My Agent vs Negamax Agent: 1.0
Negamax Agent vs My Agent: -1.0
```

c. Discussion

In my implementation, the main idea is to simulate as much as my agent can before my playing time runs out. During the simulation, expand as much as it can, and whenever expands, roll out randomly to the end of simulated game and return the winner reward. If the state is fully expanded, simply choose the best child node with the highest upper confidence bound value, and replace the current simulation node with the child.

Finally, when acting time is almost up, choose best action from input board and return.