CS 695: Programming assignment (P5)

Md. Ridwan Hossain Talukder
Department of Computer Science
George Mason University
Fairfax, USA
mtalukd@gmu.edu

I. Deep Reinforcement Learning

The Cartpole problem has a maximum total reward of 200. The random policy assessed by compute avg return has a value of 24.2 that varies with each call to a random value that the random policy may attain.

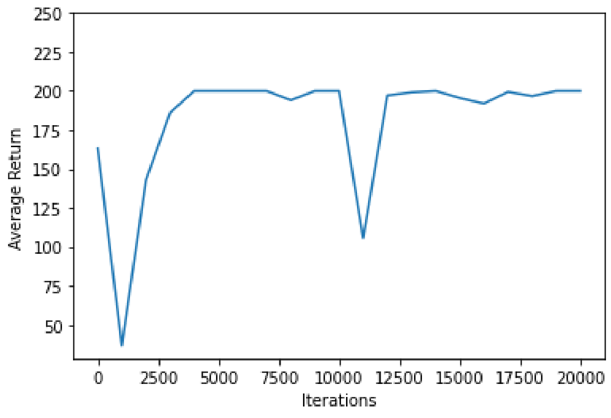


Fig. 1: Average Return vs. Iterations [lr=1e-3]

The algorithm works quite well. It reaches the maximum before the first 5000 steps. The final average return and the maximum possible value are both 200.

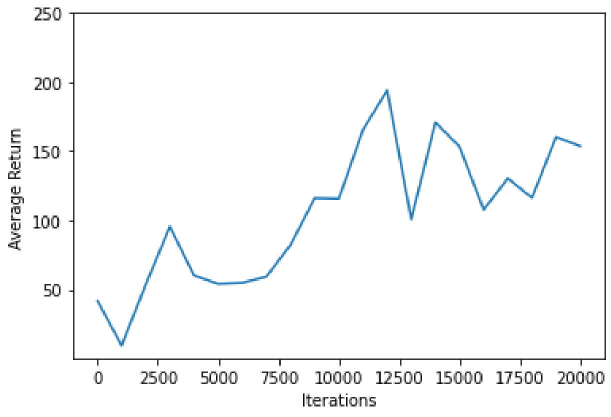


Fig. 2: Average Return vs. Iterations [lr=1e-4]

Because the variations in learning rate are too small to converge, it is never learning to maximize the return. Yes, the final average return has changed to 153.70, and the rate of improvement has decreased and is unable to converge with the modified learning rate's predefined steps.

II. Bandit Algorithms

A. Epsilon Greedy Bandits

```
def bandit_epsilon_greedy(bandits, num_steps=2000,
    epsilon=0.0, seed=695):
    """
    returns (average reward over time,
             proportion of pulls for each bandit by
    end)
    """
    random.seed(seed)
    num_pulls_per_bandit = np.zeros(len(bandits))
    tot_reward_per_bandit = np.zeros(len(bandits))
    all_rewards = []  # List of each reward returned
     per pull.

    for ii in range(num_steps):
        # Pick one of the bandits.
        temp = list()
        for idx, bandit in enumerate(bandits):
            if num_pulls_per_bandit[idx] > 0:
                temp.append(tot_reward_per_bandit[
    idx]/num_pulls_per_bandit[idx])
            else:
                temp.append(tot_reward_per_bandit[
    idx])
        bandit_ind = np.argmax(temp)
        if random.random() <= epsilon:
            bandit_ind = random.randint(0, len(
    bandits)-1)

        # Data Storage
        reward = bandits[bandit_ind].pull_arm()
        tot_reward_per_bandit[bandit_ind] += reward
        num_pulls_per_bandit[bandit_ind] += 1
        all_rewards.append(reward)

    return (np.cumsum(all_rewards) / (np.arange(
    num_steps) + 1),
            num_pulls_per_bandit / num_steps)
```

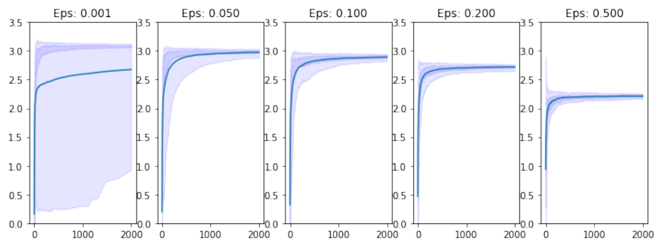


Fig. 3: Epsilon Greedy Bandit Evaluation

```
Eps = 0.001 : Optimal Pulls =  78.59\%
Eps = 0.050 : Optimal Pulls =  84.60\%
Eps = 0.100 : Optimal Pulls =  75.75\%
Eps = 0.200 : Optimal Pulls =  64.00\%
Eps = 0.500 : Optimal Pulls =  45.12\%
```

1) Answer: We get the peak ”optimal pull percentage” for Eps = 0.05. The percentage of optimum pulls falls as epsilon increases, since we choose a random bandit to pull with a higher likelihood than the optimal bandit with a larger value of epsilon.

B. UCB Bandits

```python
import math
def bandit_ucb(bandits, num_steps=2000, c=4.0, seed
    =695):
    """
    returns (average reward over time,
             proportion of pulls for each bandit by
    end)
    """
    random.seed(seed)
    num_pulls_per_bandit = np.zeros(len(bandits))
    tot_reward_per_bandit = np.zeros(len(bandits))
    all_rewards = []  # List of each reward returned
     per pull.

    for ii in range(num_steps):
        # Pick one of the bandits.
        temp = list()
        for idx, bandit in enumerate(bandits):
            if num_pulls_per_bandit[idx] > 0:
                temp.append(tot_reward_per_bandit[
    idx]/num_pulls_per_bandit[idx]+
                           (c*math.sqrt(np.log(ii)/
    num_pulls_per_bandit[idx])))
            else:
                temp.append(tot_reward_per_bandit[
    idx] + c)
        bandit_ind = np.argmax(temp)

        # Data Storage
        reward = bandits[bandit_ind].pull_arm()
        tot_reward_per_bandit[bandit_ind] += reward
        num_pulls_per_bandit[bandit_ind] += 1
        all_rewards.append(reward)

    return (np.cumsum(all_rewards) / (np.arange(
    num_steps) + 1),
            num_pulls_per_bandit / num_steps)
```

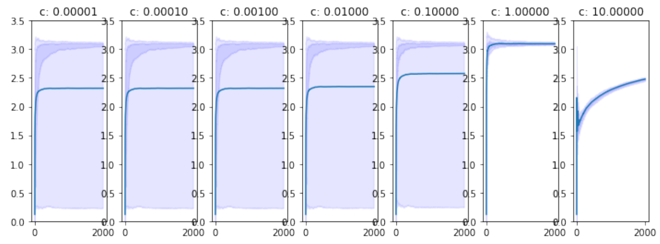


Fig. 4: UCB Bandit Evaluation

```
c = 0.00001 : Optimal Pulls =  72.86\%
c = 0.00010 : Optimal Pulls =  72.86\%
c = 0.00100 : Optimal Pulls =  72.86\%
c = 0.01000 : Optimal Pulls =  73.86\%
c = 0.10000 : Optimal Pulls =  81.80\%
c = 1.00000 : Optimal Pulls =  99.89\%
c = 10.00000 : Optimal Pulls =  31.81\%
```

1) Answer: For C=1, we have the maximum ”optimal pull percentage.” It outperforms the epsilon-greedy bandit’s best epsilon with 0.05. Because at UCB, we are only selecting actions that are better than the reward’s mean + upper confidence bound, rather than selecting actions at random from the whole action pool. As a result, we are no longer deciding at random and are instead selecting the best action, resulting in a higher optimal pull percentage.

2) Answer: When the exploration parameter C is set to a low value, the agent focuses on executing the available action rather than exploring. As a result, it misses out on actions that may have resulted in greater rewards. As the number of trials grows larger, it becomes irrelevant whether we explore or exploit first, and the optimal pulls percentage for any c can become the same.

3) Answer: When the exploration parameter C is set to a high value, the agent is more interested in exploring than than executing actions to receive reward. As a result, it spends more time exploring rather than executing actions that may have resulted in higher rewards. As the number of trials grows larger, it becomes irrelevant whether we explore or exploit first, and the optimal pulls percentage for any c becomes the same.

III. Connect Four and MCTS

A. Minimax

If there is no winner at maximum depth, the evaluation function evaluates the game as a draw. If the current player wins, it returns +1; if the other player wins, it returns -1. This evaluation method is only as good as the player’s depth.

```
Total Plays: 25
Depth 5 Wins: 19
Depth 3 Wins: 5
Draws: 1
```

Sometimes the depth-3 minimax search wins against the depth-5 minimax search because the goal lies within depth 3 and having the advantage to look ahead upto depth 5 does not give the player any advantage in the game.

B. Monte-Carlo Tree Search

```python
def monte_carlo_tree_search(start_state,
    num_iterations=1000):
    """MCTS core loop"""
    # Start by creating the root of the tree.
    root = Tree(start_state=start_state)
    # Loop through MCTS iterations.
    for _ in range(num_iterations):
        # One step of MCTS iteration
        leaf = traverse(root)
        simulation_result = rollout(leaf,
    start_state)
        backpropagate(leaf, simulation_result)
    # When done iterating, return the ’best’ child
    of the root node.
    return best_child(root)
```

```python
def backpropagate(node, simulation_result):
    """Update the node and its parent (via recursion
    )."""
    if node is None:
        return
    node.n += 1
    node.values.append(simulation_result)
    if node.parent:
        backpropagate(node.parent, simulation_result)
```

```
def best_child(node):
    """When done sampling, pick the child visited the most."""
    m = -100
    for child in node.children:
        if child.n > m:
            m = child.n
            child_to_return = child
    return child_to_return.move
```

```
def best_uct(node, C=100):
    """Pick the best action according to the UCB/UCT algorithm"""
    temp = list()
    m = -1000
    for child in node.children:
        q = sum(child.values) / child.n
        mu = C*math.sqrt(np.log(node.n)/child.n)
        added = q + mu
        if added > m:
            m = added
            child_to_return = child
    return child_to_return
```

1) Answer: For the given configuration (1000 iterations, C=5), MCTS wins more often (20 out of 25 times).

```
Total Plays: 25
MiniMax Wins: 3
MCTS Wins: 20
Draws: 2
```

2) Answer: For the given configuration (1000 iterations, C=0.1), MiniMax wins more often (15 out of 25 times).

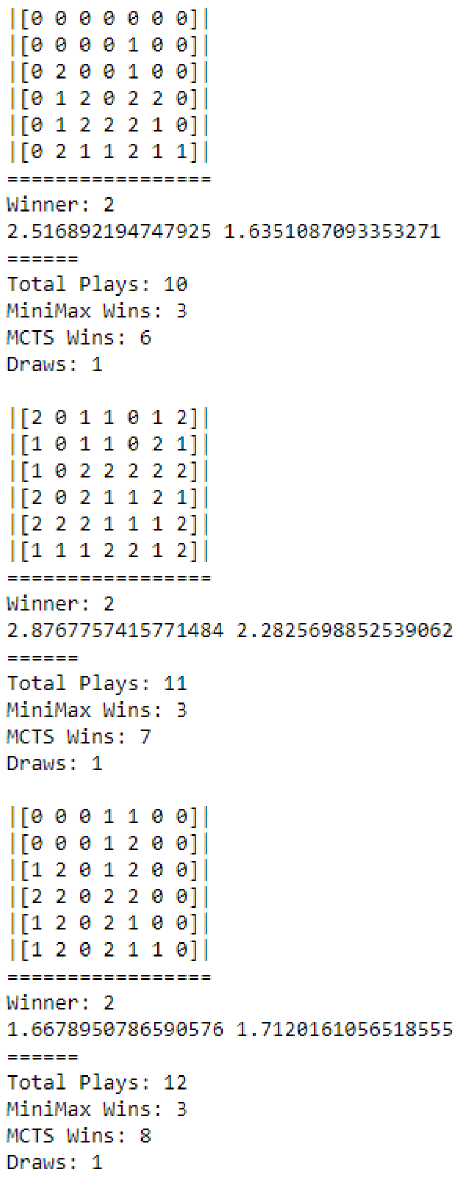
```
Total Plays: 25
MiniMax Wins: 15
MCTS Wins: 9
Draws: 1
```

3) Answer: For the given configuration (1000 iterations, C=25), MCTS wins more often (21 out of 25 times).

```
Total Plays: 25
MiniMax Wins: 2
MCTS Wins: 21
Draws: 2
```

4) Answer: For C=0.1 the win rate for MiniMax is 60% and for MCTS is 16%. 4% of the games were drawn in this case. Again, for C=25 the win rate for MiniMax is 8% and for MCTS is 84%. 8% of the games were drawn in this case. MCTS performs better with the increase of C.

5) Answer: The C parameter acts as exploration tuning parameter for MCTS. When C is increased, MCTS spends more time exploring and acquiring knowledge from the environment before exploiting, resulting in greater win rates; meanwhile, when C is reduced, MCTS spends more time exploiting than exploring, resulting in a lower game win rate. The influence may also be seen in the computation time. We raised calculation time for bigger C values and decreased computation time for lower C values since exploring more requires more time.

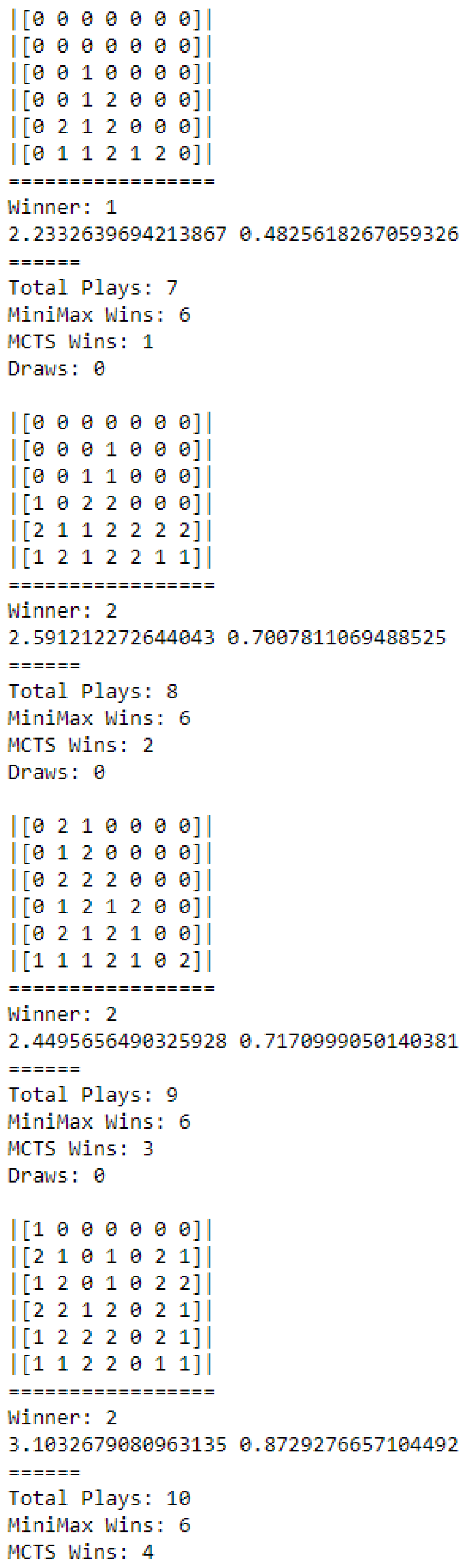


```
|[0 0 0 0 0 0 0]|
|[0 0 0 0 1 0 0]|
|[0 2 0 0 1 0 0]|
|[0 1 2 0 2 2 0]|
|[0 1 2 2 2 1 0]|
|[0 2 1 1 2 1 1]|
=================
Winner: 2
2.516892194747925 1.6351087093353271
======
Total Plays: 10
MiniMax Wins: 3
MCTS Wins: 6
Draws: 1

|[2 0 1 1 0 1 2]|
|[1 0 1 1 0 2 1]|
|[1 0 2 2 2 2 2]|
|[2 0 2 1 1 2 1]|
|[2 2 2 1 1 1 2]|
|[1 1 1 2 2 1 2]|
=================
Winner: 2
2.8767757415771484 2.2825698852539062
======
Total Plays: 11
MiniMax Wins: 3
MCTS Wins: 7
Draws: 1

|[0 0 0 1 1 0 0]|
|[0 0 0 1 2 0 0]|
|[1 2 0 1 2 0 0]|
|[2 2 0 2 2 0 0]|
|[1 2 0 2 1 0 0]|
|[1 2 0 2 1 1 0]|
=================
Winner: 2
1.6678950786590576 1.7120161056518555
======
Total Plays: 12
MiniMax Wins: 3
MCTS Wins: 8
Draws: 1
```

Fig. 5: Some of the final board states with C=5



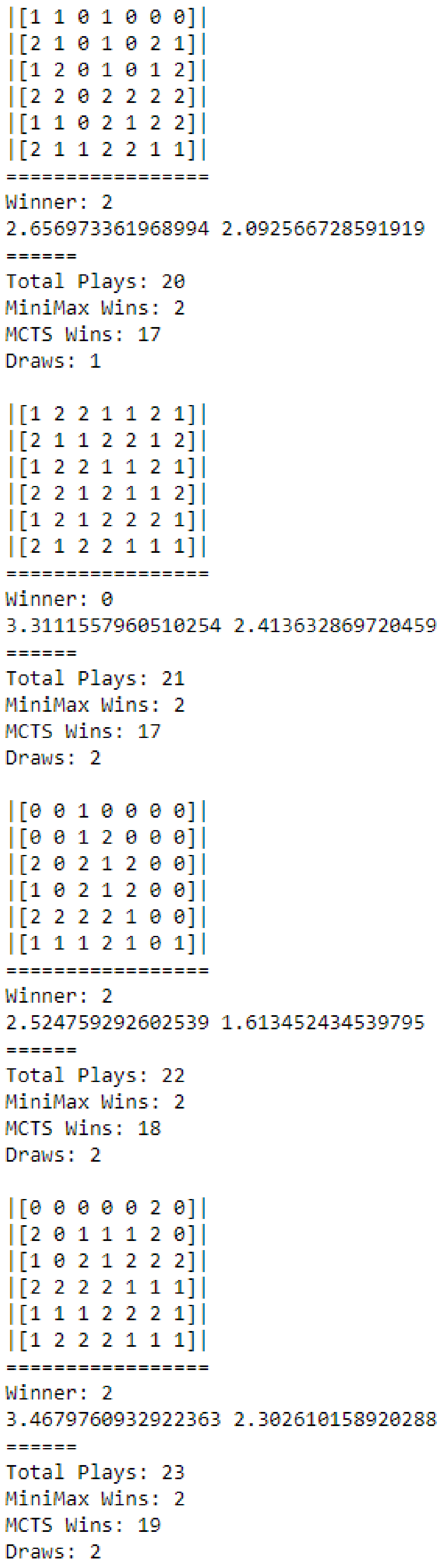
```
|[0 0 0 0 0 0 0]|
|[0 0 0 0 0 0 0]|
|[0 0 1 0 0 0 0]|
|[0 0 1 2 0 0 0]|
|[0 2 1 2 0 0 0]|
|[0 1 1 2 1 2 0]|
=================
Winner: 1
2.2332639694213867 0.4825618267059326
======
Total Plays: 7
MiniMax Wins: 6
MCTS Wins: 1
Draws: 0

|[0 0 0 0 0 0 0]|
|[0 0 0 1 0 0 0]|
|[0 0 1 1 0 0 0]|
|[1 0 2 2 0 0 0]|
|[2 1 1 2 2 2 2]|
|[1 2 1 2 2 1 1]|
=================
Winner: 2
2.591212272644043 0.7007811069488525
======
Total Plays: 8
MiniMax Wins: 6
MCTS Wins: 2
Draws: 0

|[0 2 1 0 0 0 0]|
|[0 1 2 0 0 0 0]|
|[0 2 2 2 0 0 0]|
|[0 1 2 1 2 0 0]|
|[0 2 1 2 1 0 0]|
|[1 1 1 2 1 0 2]|
=================
Winner: 2
2.4495656490325928 0.7170999050140381
======
Total Plays: 9
MiniMax Wins: 6
MCTS Wins: 3
Draws: 0

|[1 0 0 0 0 0 0]|
|[2 1 0 1 0 2 1]|
|[1 2 0 1 0 2 2]|
|[2 2 1 2 0 2 1]|
|[1 2 2 2 0 2 1]|
|[1 1 2 2 0 1 1]|
=================
Winner: 2
3.1032679080963135 0.8729276657104492
======
Total Plays: 10
MiniMax Wins: 6
MCTS Wins: 4
```

Fig. 6: Some of the final board states with C=0.1



```
|[1 1 0 1 0 0 0]|
|[2 1 0 1 0 2 1]|
|[1 2 0 1 0 1 2]|
|[2 2 0 2 2 2 2]|
|[1 1 0 2 1 2 2]|
|[2 1 1 2 2 1 1]|
=================
Winner: 2
2.656973361968994 2.092566728591919
======
Total Plays: 20
MiniMax Wins: 2
MCTS Wins: 17
Draws: 1

|[1 2 2 1 1 2 1]|
|[2 1 1 2 2 1 2]|
|[1 2 2 1 1 2 1]|
|[2 2 1 2 1 1 2]|
|[1 2 1 2 2 2 1]|
|[2 1 2 2 1 1 1]|
=================
Winner: 0
3.3111557960510254 2.413632869720459
======
Total Plays: 21
MiniMax Wins: 2
MCTS Wins: 17
Draws: 2

|[0 0 1 0 0 0 0]|
|[0 0 1 2 0 0 0]|
|[2 0 2 1 2 0 0]|
|[1 0 2 1 2 0 0]|
|[2 2 2 2 1 0 0]|
|[1 1 1 2 1 0 1]|
=================
Winner: 2
2.524759292602539 1.613452434539795
======
Total Plays: 22
MiniMax Wins: 2
MCTS Wins: 18
Draws: 2

|[0 0 0 0 0 2 0]|
|[2 0 1 1 1 2 0]|
|[1 0 2 1 2 2 2]|
|[2 2 2 2 1 1 1]|
|[1 1 1 2 2 2 1]|
|[1 2 2 2 1 1 1]|
=================
Winner: 2
3.4679760932922363 2.302610158920288
======
Total Plays: 23
MiniMax Wins: 2
MCTS Wins: 19
Draws: 2
```

Fig. 7: Some of the final board states with C=25